universität wien

DISSERTATION / DOCTORAL THESIS

Titel der Dissertation /Title of the Doctoral Thesis

„Supporting Architecture Evolution in Microservice-Based Systems and Infrastructure-as-Code Based Deployments“

verfasst von / submitted by

Evangelos Ntentos

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Doktor der technischen Wissenschaften (Dr. techn.)

Wien, 2023 / Vienna 2023

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 786 880
Dissertationsgebiet lt. Studienblatt / field of study as it appears on the student record sheet:	Informatik
Betreut von / Supervisor:	Univ.-Prof. Dr. Uwe Zdun

Acknowledgements

I would like to express my deepest gratitude to my advisor Uwe Zdun, for his invaluable guidance, encouragement, and support throughout my doctoral research. His insightful feedback and constructive criticism were instrumental in shaping my research and academic growth. I am grateful to all my co-authors, and especially to Konstantinos Plakidas, who provided me with their expertise and resources, and without whom my research would not have been possible. Their contributions were critical to the success of my research. I extend my heartfelt thanks to my family, Stavros, Vasiliki and Georgios Ntentos, for their unconditional love, support, and encouragement throughout my academic journey. Their unwavering belief in me has been a constant source of strength and motivation. Lastly, I would like to thank the faculty of Computer Science of University of Vienna, for providing me with the resources and infrastructure necessary for conducting my research. Their support has been invaluable in enabling me to achieve my academic goals.

Thank you all for your unwavering support and encouragement throughout my doctoral journey.

Abstract

Microservice systems are a modern approach to software architecture that involves the decomposition of a large, monolithic application into smaller, independently deployable services. The traditional approach to software development involved building large, monolithic applications that included all the functionality needed for the system. While this approach made it easier to develop and deploy applications, it also led to several challenges, such as increased complexity, reduced scalability, and difficulty in maintaining and updating the application.

In contrast, microservice systems offer a number of benefits over monolithic applications. For example, each microservice can be developed, tested, and deployed independently, allowing for faster and more efficient development cycles. This also allows for easier scaling of individual services, as only the services that require additional resources need to be scaled, rather than the entire application. Another key advantage of microservice systems is increased resilience. Because each service is deployed independently, if one service experiences a failure, the other services in the system can continue to operate normally. This creates more robust systems that can continue to operate even in the face of individual service failures. In addition to these benefits, microservice systems also offer improved scalability, as services can be deployed across multiple nodes or locations to meet changing demands. This allows the system to handle increased load and traffic, while maintaining high levels of performance. One of the key challenges of microservice systems is managing the communication and integration between services. To address this challenge, microservice systems use well-defined APIs to enable services to communicate with each other in a consistent and predictable manner. This facilitates the development of more complex systems that can be managed and updated more easily.

Another challenge in microservice systems is ensuring that services are deployed and managed consistently across the system. To address this challenge, many organizations use DevOps practices, such as continuous integration and delivery, to automate the deployment and management of services. This helps to ensure that services are deployed and managed consistently across the system, while also reducing the risk of manual errors and improving the speed and efficiency of the deployment process. Although there are established patterns and best practices documented, actual implementations often deviate from them and it is difficult to assess this manually for large-scale systems. One of the main challenges in software architecture is avoiding architecture model drift and erosion, which is exacerbated in microservice-based systems due to frequent changes and the use of multiple technologies. Existing solutions for reconstructing architecture models fall short in addressing these challenges as they struggle with continuous evolution, accuracy, and polyglot settings. Spotting violations in complex or large systems can be hard and tedious.

The complexity of developing and deploying microservice-based systems has increased and become a challenge due to the requirement of frequent releases and the substantial number of infrastructure nodes necessary for system operation. Infrastructure as code (IaC) is a practice that

involves the use of configuration files, scripts, and other code artifacts to manage and provision the underlying infrastructure for a software system. This allows for the deployment and management of infrastructure to be automated and repeatable, which is crucial for ensuring the stability and reliability of microservice systems. When deploying microservices in a production environment, it is important to have a well-designed infrastructure that supports the requirements of the services. This includes having sufficient computing resources, network connectivity, storage, and other resources needed to run the services effectively. IaC automates and standardizes infrastructure management, enabling efficient deployment of new services, scaling of existing services, and updates. Defining infrastructure as code allows for versioning, testing, and collaboration, improving reliability and repeatability of the deployment process. IaC can be applied to various types of infrastructure, including virtual machines, containers, and cloud resources. In the context of microservice systems, IaC can be particularly valuable, as it allows for the deployment and management of the infrastructure in a way that is tailored to the requirements of the microservices. Despite its widespread use, there is a lack of methods for evaluating architectural conformance and addressing architectural issues, such as loose coupling, in IaC-based deployments.

In summary, IaC and microservices are complementary technologies that can be used together to deploy and manage software systems in a scalable, flexible, and reliable way. By automating the deployment and management of infrastructure, IaC helps to reduce the risk of manual errors and improve the overall speed and efficiency of the deployment process.

In this thesis, we conducted a thorough and qualitative examination of best practices and patterns in microservice data management architecture through practitioner descriptions. By following a model-based qualitative research approach, we arrived at a formal architecture decision model. Upon comparing the completeness of our model to an existing pattern catalog, we found that it greatly reduces the effort and uncertainty involved in comprehending microservice data management decisions. We designed an architecture model abstraction approach that would allow for a better understanding of the architecture models of complex systems that evolve over time. We have identified two possible approaches and have evaluated them through an empirical case study: an opportunistic approach and a reusable semi-automatic detector-based approach. Furthermore, we developed an automated method for evaluating microservice architecture conformance to established patterns and practices, and IaC-based deployments, by using a microservice system's component model. This involved establishing a ground truth for all possible microservice architectures and determining relevant metrics to assess microservice and IaC principles, regardless of the technology used. Additionally, we have created an automated assistance approach to aid architects during the development of microservice systems and IaC deployments. Specifically, our focus was on providing a foundation for automated architecture reconstruction, assessing conformance to microservice-specific patterns and practices, and identifying potential violations. As a result, architects can use our approach to enhance architecture compliance through a continuous feedback process, offering practical solutions for improving the overall quality of microservice systems and IaC deployments.

This thesis aims to develop a systematic and reusable architectural design decision model for microservices and create an accurate approach for reconstructing component architecture models. It also focuses on devising a method for automatically assessing architecture conformance to microservice patterns and practices, along with determining suitable metrics for the assessment.

The combination of these techniques with runtime and dynamic features of software systems can improve problem detection and assessment, reducing the decision space to a manageable set of actionable options.

Kurzfassung

Microservice-Systeme sind ein moderner Ansatz für die Softwarearchitektur, bei dem eine große, monolithische Anwendung in kleinere, unabhängig einsetzbare Dienste zerlegt wird. Diese Dienste kommunizieren über genau definierte APIs miteinander und ermöglichen so eine höhere Skalierbarkeit, Flexibilität und Belastbarkeit. Der traditionelle Ansatz der Softwareentwicklung bestand darin, große, monolithische Anwendungen zu erstellen, die alle für das System benötigten Funktionen enthielten. Dieser Ansatz erleichterte zwar die Entwicklung und Bereitstellung von Anwendungen, führte aber auch zu einer Reihe von Problemen, wie z. B. erhöhter Komplexität, geringerer Skalierbarkeit und Schwierigkeiten bei der Wartung und Aktualisierung der Anwendung.

Im Gegensatz dazu bieten Microservice-Systeme eine Reihe von Vorteilen gegenüber monolithischen Anwendungen. So kann beispielsweise jeder Microservice unabhängig entwickelt, getestet und bereitgestellt werden, was schnellere und effizientere Entwicklungszyklen ermöglicht. Dies ermöglicht auch eine einfachere Skalierung einzelner Dienste, da nur die Dienste, die zusätzliche Ressourcen benötigen, skaliert werden müssen, und nicht die gesamte Anwendung. Ein weiterer entscheidender Vorteil von Microservice-Systemen ist die erhöhte Ausfallsicherheit. Da jeder Dienst unabhängig eingesetzt wird, können bei einem Ausfall eines Dienstes die anderen Dienste im System normal weiterarbeiten. Dies ermöglicht robustere Systeme, die auch bei einem Ausfall einzelner Dienste weiterarbeiten können. Zusätzlich zu diesen Vorteilen bieten Microservice-Systeme auch eine bessere Skalierbarkeit, da die Dienste über mehrere Knoten oder Standorte verteilt werden können, um wechselnden Anforderungen gerecht zu werden. Auf diese Weise kann das System eine höhere Last und einen größeren Datenverkehr bewältigen und gleichzeitig ein hohes Leistungsniveau beibehalten. Eine der größten Herausforderungen von Microservice-Systemen ist die Verwaltung der Kommunikation und Integration zwischen den Diensten. Um diese Herausforderung zu bewältigen, verwenden Microservice-Systeme genau definierte APIs, damit die Dienste auf konsistente und vorhersehbare Weise miteinander kommunizieren können. Dies ermöglicht die Entwicklung komplexerer Systeme, die sich leichter verwalten und aktualisieren lassen.

Eine weitere Herausforderung bei Microservice-Systemen besteht darin, sicherzustellen, dass die Dienste systemweit einheitlich bereitgestellt und verwaltet werden. Um diese Herausforderung zu bewältigen, setzen viele Unternehmen DevOps-Verfahren ein, z. B. kontinuierliche Integration und Bereitstellung, um die Bereitstellung und Verwaltung von Diensten zu automatisieren. Auf diese Weise wird sichergestellt, dass die Dienste im gesamten System einheitlich bereitgestellt und verwaltet werden, während gleichzeitig das Risiko manueller Fehler verringert und die Geschwindigkeit und Effizienz des Bereitstellungsprozesses verbessert wird. Obwohl es etablierte Muster und dokumentierte Best Practices gibt, weichen die tatsächlichen Implementierungen oft davon ab, und es ist schwierig, dies bei großen Systemen manuell zu beurteilen. Eine der größten Herausforderungen in der Software-Architektur ist die Vermeidung von Drift und

Erosion des Architekturmodells, was sich bei Microservice-basierten Systemen aufgrund häufiger Änderungen und der Verwendung mehrerer Technologien noch verschärft. Bestehende Lösungen zur Rekonstruktion von Architekturmodellen können diese Herausforderungen nicht bewältigen, da sie mit der kontinuierlichen Entwicklung, der Genauigkeit und den polyglotten Einstellungen zu kämpfen haben. Das Aufspüren von Verstößen in komplexen oder großen Systemen kann schwierig und mühsam sein.

Die Komplexität der Entwicklung und des Einsatzes von Microservice-basierten Systemen hat zugenommen und ist zu einer Herausforderung geworden, da häufige Releases erforderlich sind und eine beträchtliche Anzahl von Infrastrukturknoten für den Systembetrieb notwendig ist. Infrastructure as Code (IaC) ist eine Praxis, die die Verwendung von Konfigurationsdateien, Skripten und anderen Code-Artefakten zur Verwaltung und Bereitstellung der zugrunde liegenden Infrastruktur für ein Softwaresystem beinhaltet. Dies ermöglicht eine automatisierte und wiederholbare Bereitstellung und Verwaltung der Infrastruktur, was für die Gewährleistung der Stabilität und Zuverlässigkeit von Microservice-Systemen entscheidend ist. Bei der Bereitstellung von Microservices in einer Produktionsumgebung ist es wichtig, eine gut konzipierte Infrastruktur zu haben, die die Anforderungen der Services unterstützt. Dazu gehören ausreichende Rechenressourcen, Netzwerkkonnektivität, Speicherplatz und andere Ressourcen, die für die effektive Ausführung der Dienste erforderlich sind. IaC automatisiert und standardisiert die Verwaltung der Infrastruktur und ermöglicht die effiziente Bereitstellung neuer Dienste, die Skalierung bestehender Dienste und Updates. Die Definition der Infrastruktur als Code ermöglicht die Versionierung, das Testen und die Zusammenarbeit und verbessert die Zuverlässigkeit und Wiederholbarkeit des Bereitstellungsprozesses. IaC kann auf verschiedene Arten von Infrastruktur angewendet werden, darunter virtuelle Maschinen, Container und Cloud-Ressourcen. Im Kontext von Microservice-Systemen kann IaC besonders wertvoll sein, da es die Bereitstellung und Verwaltung der Infrastruktur auf eine Weise ermöglicht, die auf die Anforderungen der Microservices zugeschnitten ist. Trotz der weiten Verbreitung von IaC fehlt es an Methoden zur Bewertung der Architekturkonformität und zur Behandlung von Architekturproblemen, wie z. B. der losen Kopplung, in IaC-basierten Implementierungen.

Zusammenfassend lässt sich sagen, dass IaC und Microservices komplementäre Technologien sind, die zusammen verwendet werden können, um Softwaresysteme auf skalierbare, flexible und zuverlässige Weise bereitzustellen und zu verwalten. Durch die Automatisierung der Bereitstellung und Verwaltung der Infrastruktur hilft IaC, das Risiko manueller Fehler zu verringern und die Geschwindigkeit und Effizienz des Bereitstellungsprozesses insgesamt zu verbessern.

In dieser Arbeit haben wir eine gründliche und qualitative Untersuchung von Best Practices und Mustern in der Microservice-Datenmanagement-Architektur anhand von Beschreibungen von Praktiker*innen durchgeführt. Mit Hilfe eines modellbasierten qualitativen Forschungsansatzes haben wir ein formales Architekturentscheidungsmodell entwickelt. Beim Vergleich der Vollständigkeit unseres Modells mit einem bestehenden Musterkatalog stellten wir fest, dass es den Aufwand und die Ungewissheit, die mit dem Verstehen von Microservice-Datenmanagement-Entscheidungen verbunden sind, erheblich reduziert. Wir haben einen Ansatz zur Abstraktion von Architekturmodellen entwickelt, der ein besseres Verständnis der Architekturmodelle komplexer Systeme, die sich im Laufe der Zeit weiterentwickeln, ermöglichen würde. Wir haben zwei mögliche Ansätze identifiziert und anhand einer empirischen Fallstudie evaluiert: einen

opportunistischen Ansatz und einen wiederverwendbaren halbautomatischen detektorbasierten Ansatz. Darüber hinaus haben wir eine automatisierte Methode entwickelt, um die Konformität von Microservice-Architekturen mit etablierten Mustern und Praktiken sowie IaC-basierte Implementierungen anhand des Komponentenmodells eines Microservice-Systems zu bewerten. Dies beinhaltete die Erstellung einer Basiswahrheit für alle möglichen Microservice-Architekturen und die Bestimmung relevanter Metriken zur Bewertung von Microservice- und IaC-Prinzipien, unabhängig von der verwendeten Technologie. Darüber hinaus haben wir einen automatisierten Assistenzansatz entwickelt, um Architekt*innen bei der Entwicklung von Microservice-Systemen und IaC-Implementierungen zu unterstützen. Insbesondere haben wir uns darauf konzentriert, eine Grundlage für die automatische Rekonstruktion der Architektur zu schaffen, die Konformität mit Microservice-spezifischen Mustern und Praktiken zu bewerten und potenzielle Verstöße zu identifizieren. Als Ergebnis können Architekt*innen unseren Ansatz nutzen, um die Konformität der Architektur durch einen kontinuierlichen Feedback-Prozess zu verbessern und praktische Lösungen zur Verbesserung der Gesamtqualität von Microservice-Systemen und IaC-Implementierungen anzubieten.

Diese Arbeit zielt darauf ab, ein systematisches und wiederverwendbares Architekturentscheidungsmodell für Microservices zu entwickeln und einen genauen Ansatz für die Rekonstruktion von Komponentenarchitekturmodellen zu schaffen. Ein weiterer Schwerpunkt ist die Entwicklung einer Methode zur automatischen Bewertung der Konformität der Architektur mit Microservice-Mustern und -Praktiken sowie die Bestimmung geeigneter Metriken für die Bewertung. Die Kombination dieser Techniken mit Laufzeit- und dynamischen Merkmalen von Softwaresystemen kann die Problemerkennung und -bewertung verbessern und den Entscheidungsraum auf eine überschaubare Menge von Handlungsoptionen reduzieren.

Contents

List of Tables

List of Figures

Listings

Part I

Foundations and Research Overview

1 Introduction

1.1 Thesis Subject and Motivation

The subject of this thesis concerns the study of *microservice architectures* and *IaC-based deployments*. The objective is to provide automated assistance for architecting during the evolution of microservice-based systems. Specifically, we strive to establish the groundwork for an automated process of reconstructing the architecture, assessing conformance to patterns and practices specific to microservice architectures, detect possible violations and provide actionable options to architects for improving architecture conformance as part of a feedback mechanism.

Microservice architectures [New15, Zim17] have emerged from established practices in service-oriented computing (cf. [PJ16, Ric17, ZKL+09]). The microservices approach emphasizes business capability-based and domain-driven design, development in independent teams, cloud-native technologies and architectures, polyglot technology stacks including polyglot persistence, lightweight containers, loosely coupled service dependencies, and continuous delivery (cf. [LF04, New15, Zim17]). Some of these tenets introduce substantial challenges for such architectures. Notably, it is usually advised to decentralize all data management concerns. Such an architecture requires, in addition to the already existing non-trivial design challenges intrinsic to distributed systems, sophisticated solutions for data integrity, data querying, transaction management, and consistency management [New15, Zim17, PJ16, Ric17]. Many authors have written about microservice data management and various attempts to document microservice patterns and best practices exist [Ric17, Gup17, LF04, PJ16]. Nevertheless, most of the established practices in the industry are only reported in the so-called "grey literature", consisting of practitioner blogs, experience reports, system documentations, etc. In most cases, each of these sources documents a few existing practices well, but usually they do not provide systematic architectural guidance. Instead, the reported practices are largely based on personal experience, often inconsistent, and, when studied on their own, incomplete. This creates considerable uncertainty and risk in architecting microservices, which can be reduced either through substantial personal experience or by a careful study of a large set of knowledge sources. The fact that microservice-based systems are complex and polyglot means that an automatic or semi-automatic assessment of their conformance to best patterns and practices is difficult: real-world systems feature combinations of patterns, and different degrees of violations of the same; and different technologies in different parts of the system implement the patterns in different ways, making the automatic parsing of code and identification of the patterns a haphazard process. Making matters even more challenging, a high level of automation is required for complex systems. While a manual assessment of small-scale systems of a few services by an expert is probably as quick and as accurate as an automated one, that is not true for industrial-scale systems of several hundred or more services, which are being developed by different teams or companies, evolving at different paces. In that

case, manual assessment is laborious and inaccurate, and a more automated method would vastly improve cost-effectiveness.

Architecture reconstruction is also a challenging topic in microservices. Several architecture reconstruction approaches have been proposed to automatically or semi-automatically produce architecture models from the source code [DP09, MNS95, MMW02]. Unfortunately, these approaches are usually not designed for supporting continuous evolution, meaning a substantial effort is needed to either manually maintain the reconstructed architecture model or redo the reconstruction after the system has evolved (see [HZ14]). In addition, automated approaches have only low accuracies (see [GIM13]), meaning much manual effort is needed for correcting and augmenting their results. Most reconstruction approaches focus on a very limited number of programming languages and technologies (see [DP09]), meaning they are hard to apply to systems such as today's microservice systems which use polyglot programming, persistence and technologies, often in their latest incarnations.

The development and deployment of microservice-based systems have become increasingly challenging and complex due to the need for frequent releases and the large number of infrastructure nodes required to keep the system running. Furthermore, cloud computing has caused a significant increase in the number of infrastructure nodes required by microservice systems [Nyg07]. Additionally, modern systems are frequently released to production, sometimes multiple times per day, leading to more frequent infrastructure changes [HF10, Nyg07]. To address these challenges, organizations have adopted IaC, which involves using reusable scripts to manage and provision infrastructure [Mor15]. IaC ensures that a provisioned environment remains consistent with the same configuration, and configuration files containing infrastructure specifications can be easily edited and distributed [Mor15, ABDN+17]. Implementing IaC practices can also improve consistency, security, and reduce errors and manual configuration effort [ABDN+17]. Without IaC, it becomes increasingly difficult to manage the scale of current systems' components and infrastructure [Mor15]. IaC technologies (e.g. Ansible, Terraform) allow to provision, deploy, and configure arbitrary application architectures. Thus, developers and operators, respectively, can model any desired deployment. This freedom quickly results in problems if security-related aspects are not taken into account. For example, vulnerabilities in IaC deployment models (e.g. weak access and traffic control) could allow attackers to access procedures and run code to hack the application. Furthermore, the deployment infrastructure can be structured using infrastructure stacks. An infrastructure stack is a collection of infrastructure elements/resources that are defined, provisioned, and updated as a unit [Mor15]. An incorrect structure can result in issues if coupling-related aspects are not considered. For instance, defining all the system deployment artifacts as only one unit in one infrastructure stack can significantly impact the dependencies of system parts and teams as well as the independent deployability of system services. This, combined with the fact that industry-scale systems support multiple such architectural practices at once and also different implementations of them, makes it difficult to assess whether an IaC deployment model that implicitly describes also the application's architecture is conforming to recommended best practices or not. In modern cloud-based architectures, such as microservice architectures [New15] and other frequently released systems [Nyg07], an automatic assessment method would produce more accurate results. For instance, this could be applied in the context of continuous delivery practices employed in these systems requiring the

automated setup of infrastructures in usually every run of the delivery pipeline [HF10].

1.2 Thesis Structure

This doctoral thesis is structured as follows:

- In **Part I**, we discuss the state of the art that constitutes the basis of the thesis and point out the novelty of our research work in comparison to existing approaches. In addition, we analyze the main research problems addressed and the research methods applied in the context of this thesis.
- In **Part II**, we study all the patterns and practices related to data management in microservices to create a systematic, reusable architectural design decision model. This model aims to complement the existing literature on individual practices by practitioners with a consistent and unbiased overview of industry practices. The design decision model provides a comprehensive approach to microservice architecture.
- In **Part III**, we suggest an approach for evaluating microservice architecture conformity to patterns and practices through an automatic assessment using the component model of a microservice system. To accomplish this, we establish a standard or "ground truth" for each possible microservice architecture and determine appropriate metrics to assess all relevant microservice principles, regardless of the technology used. Moreover, we report on a research study aiming to design a highly accurate architecture model abstraction approach for comprehending component architecture models of highly polyglot systems that can cope with continuous evolution.
- In **Part IV**, we focus on automated assistance for architects during the ongoing development of microservice systems. Specifically, we focus on the establishment of a foundation for automated architecture reconstruction, evaluation of conformity to microservice-specific patterns and practices, and detection of potential violations. By doing so, we provide architects with practical solutions for enhancing architecture compliance through a continuous feedback process.
- Finally, in **Part V**, we summarize the main contributions of the thesis.

2 State of the Art

In this chapter, we give the context of our research by discussing the related work. In this discussion, we also define some challenges and the motivation behind the research work. First, we briefly introduce the main context of this doctoral thesis. In addition, we present the related work on the existing approaches in the areas of microservices, IaC best practices, frameworks for detecting violations, and architecture conformance checking. The discussion of the related work, however, does not end here. Each chapter introduces additional related works and a comparison with the respective approach.

2.1 Research Context

Microservices typically do not share their data with other services, are commonly deployed in lightweight containers or virtualized environments and communicate with other services through message-based remote APIs that are loosely coupled. Microservices utilize multiple programming languages and data storage systems, along with DevOps practices like continuous delivery and end-to-end monitoring (see e.g. [Zim17, PZA+17]). While microservices are one of many service-based architecture decomposition approaches (see e.g. [PJ16, PW09, Ric17, ZGK+07]), they, like others, do not effectively address the software architecture problems of architecture drift and erosion. These problems occur when changes are made to the source code that violate or are not reflected in the architecture model, causing the architecture models to become out of sync with the code during system evolution (see e.g. [PW92, ZZGL08]). The principle of strong coupling goes against some of the fundamental principles of microservices. Specifically, the ability to release microservices independently is a crucial aspect of modern systems that adopt DevOps practices. Strong coupling between microservices makes it challenging to achieve releasability, as it hinders the independent and swift release of individual microservices. Additionally, the development of independent teams becomes more challenging due to strong dependencies between microservices, and the autonomous deployment of individual microservices in lightweight containers becomes difficult for the same reason.

The complexity of developing and deploying microservice-based systems has increased due to the demand for quick releases and the extensive number of infrastructure nodes necessary to operate the system [HF10, Nyg07]. To manage and provision IT infrastructure effectively, IaC architecture utilizes software development techniques like version control, automated testing, and continuous integration and delivery (CI/CD) [Mor15]. In an IaC architecture, infrastructure is described using code, typically in a declarative language such as YAML or JSON, which specifies the desired state of the infrastructure. The code is versioned using a source control system, such as Git, which allows teams to collaborate on infrastructure changes and track changes over time. Continuous integration and delivery (CI/CD) pipelines are used to automate the testing, building,

and deployment of infrastructure code changes. The pipeline can be configured to run tests automatically and to deploy the code changes to production or staging environments when they pass. By treating infrastructure as code, it becomes easier to manage infrastructure changes and track changes over time. Additionally, IaC architecture can help improve security by providing visibility into infrastructure changes and enabling teams to identify and fix security vulnerabilities quickly and reduce coupling in deployment by structuring the infrastructure elements based on certain responsibilities.

In this thesis, as an initial step, we investigate microservice-related patterns and practices in order to provide a systematic and consistent, reusable architectural design decision model which can complement the rich literature of detailed descriptions of individual practices by practitioners. The design decision model aims to provide an unbiased and more complete treatment of industry practices. Another necessary component of this work is the creation of a highly accurate architecture model abstraction approach for reconstructing component architecture models of highly polyglot systems, that can cope with continuous evolution. The next step focuses on devising a method for automatically assessing architecture conformance to patterns and practices in microservice architectures and IaC-based deployments based on a microservice system's component model. To do this, we also need to establish a 'ground truth' about the state of each possible microservice architecture. Consequently, we need to determine suitable metrics for the assessment of all relevant microservice and IaC tenets independent of the technology employed. By integrating these techniques with well-established runtime and dynamic features of software systems, it is possible to improve the accuracy of problem detection and assessment. Ideally, this combination would identify the exact cause of the problem, thereby reducing the number of potential solutions (reducing the decision space) to a manageable set of actionable options.

2.2 Related Work

This section provides an overview of the related works that were utilized in our studies. Moreover, each chapter expands on and builds upon these works, including a comparison with the approach that is presented.

2.2.1 Related Works on Best Practices and Patterns in Microservices

Various studies have extensively examined best practices for microservices. Richardson [Ric17] has published a set of microservice patterns and practices, while Skowronski [Sko19] has published another set of practices specific to event-driven microservice architectures. Zimmermann et al. [ZSZ+19] have introduced patterns that are relevant to microservice APIs. Fowler and Lewis [LF04] have discussed the basics and best practices of microservices, and Pahl and Jamshidi [PJ16] have summarized many of them in a mapping study. Taibi and Lenarduzzi [TL18] have studied "bad smells" in microservices. Chapters 4, 6, and 7 provide a comprehensive review of related works on patterns and practices in microservices and compare them with our approaches.

2.2.2 Related Works on Frameworks and Metrics in Microservices

Nowadays, many studies on service metrics concentrate on runtime properties (e.g. [PRG18]). Several studies have assessed microservice-based software architectures using metrics, such as those proposed in [PW09, ZNL17, BWZ17]. However, each of these studies only focuses on a narrow set of architecture-relevant tenets (e.g. loose coupling), and there is no overall approach for assessing different microservice tenets. For instance, Pautasso and Wilde [PW09] proposed a composite metric based on facets to evaluate loose coupling in service-oriented systems. Zdun et al. [ZNL17] investigated the independent deployment of microservices and defined metrics to assess architecture conformance to microservice patterns in terms of two aspects: independent deployment and shared dependencies of services. Engel et al. [ELBH18] proposed a method based on real-time system communication traces to extract metrics that conform to recommended microservice design principles, such as loose coupling and small service size. These studies treat microservice architectures as a combination of components and connectors, taking into account the technologies employed, and producing assessments that aggregate different evaluation parameters (i.e. metrics). If automatically collected, these metrics could be utilized as part of larger assessment models or frameworks during the design and development phases. Chapters 6, 7, 10, and 11 discuss these related works, among others.

2.2.3 Related Works on Best Practices and Patterns in IaC-based Deployments

As IaC practices gain more popularity and acceptance in the industry, there is a growing body of scientific research focused on collecting and systematizing IaC-related patterns and practices. Kumara et al. [KGR+21] present a comprehensive catalog of language-agnostic and language-specific best and bad practices that address implementation issues, design issues, and violations of essential IaC principles. Morris [Mor15] provides guidance on how to manage IaC and describes technologies related to IaC-based practices, along with a broad catalog of patterns and practices classified into several categories. Sharma et al. [SFS16] present a catalog of design and implementation smells specific to Puppet, while Schwarz et al. [SSL18] provide a catalog of smells for Chef. Our work also adheres to IaC-specific recommendations outlined in AWS [AWS21], OWASP [OWA21a, OWA21c, OWA21b], and the Cloud Security Alliance [Clo18, Clo21]. Chapters 8 and 9 discuss additional related works on patterns and practices in IaC-based deployments.

2.2.4 Related Works on Frameworks and Metrics in IaC-based Deployments

Numerous studies have proposed various tools and metrics to evaluate and enhance the quality of IaC deployment models. For instance, Dalla Palma et al. [DDPT20, DDT20] offer a catalog of 46 quality metrics that concentrate on Ansible scripts to identify IaC-related characteristics and demonstrate how to use them to analyze IaC scripts. Wurster et al. [WBH+20] present TOSCA Lightning, an integrated toolchain that specifies multi-service applications with TOSCA Light and transforms them into different deployment technologies that are production-ready.

Kumara et al. [KVM+20] suggest a tool-based approach for identifying smells in TOSCA models. Sotiropoulos et al. [SMS20] develop a tool-based approach that detects dependencies-related issues by analyzing Puppet manifests and their system call trace. Van der Bent et al. [vdBHVG18] define metrics that reflect best practices for assessing Puppet code quality. Pendleton et al. [PGLCX16] provide a comprehensive survey on security metrics that focuses on how a system security state can evolve as an outcome of cyber-attack defence interactions. They also propose a security metrics framework for measuring system-level security. Frameworks and metrics related to IaC-based deployments are further described in Chapters 8 and 9.

2.2.5 Software Architecture Conformance Checking in IaC-based Deployments

In [FBKL17, KBKL18], an approach is presented for automatically verifying compliance of declarative deployment models during design time. The approach involves modeling compliance rules as a pair of deployment model fragments, where one fragment serves as a detector subgraph that determines whether the rule applies to a given deployment model, and the second fragment determines a desired structure that the deployment model must contain. However, this approach is generic and does not introduce specific compliance rules, such as for the security domain, and assumes that the rule modeler can translate best practices into compliance rules of the expected format. Other approaches related to architecture conformance checking, such as those based on automated extraction techniques [GAK99, VDHK+04], can be used to check conformance to architecture patterns [GAK99, HZ14] or other architectural rules [VDHK+04]. Chapters 8 and 9 provide additional works and comparisons in this area.

3 Problem Analysis and Research Approach

In this chapter, we formulate the main research problems that are addressed in the context of this doctoral thesis. Each research problem leads us to consider one or more research questions. Furthermore, we discuss the research methodology as well as the research methods that have been applied to evaluate the proposed approaches that address the research questions under study.

3.1 Research Topics and Aims

The aim was to contribute to a more robust, comprehensive, and evidence-based understanding of architecting microservice-based systems and deployments. To do this, it is necessary to collect, model, analyze, assess, and understand both the extant practices and the theoretical underpinnings of the microservice domain. Specifically, this requires us to:

1. investigate existing practices in the industry and extant scientific and practitioner literature to acquire a comprehensive overview of the state of the field,

2. systematically categorize and model the extant practices and recommendations in a unified framework for modelling and analyzing microservice-based systems and deployments,

3. establish an automatic approach to assess architecture conformance to recommended patterns and practices of real-world microservice-based systems and deployments,

4. automatically calculate and provide actionable options to architects for improving architecture conformance as part of a feedback loop.

Based on the information derived from this process, we aim to answer a series of research questions on the subject of microservice-based systems and IaC-based deployments evolution.

3.2 Research Questions

Based on the motivation described in 1.1, the purpose of this thesis is to investigate methods for understanding the present practices in the industry and literature that relate to microservice-based systems and IaC-based deployments. Furthermore, it endeavours to assess how well a system complies with the recommended best practices and patterns and to suggest improvements to enhance architecture conformance. We state four primary research questions (RQs), and every chapter comprises several subsidiary questions that pertain to the following RQs:

- **RQ1:** What are the commonly used patterns and practices in architecting microservice-based systems and IaC-based deployments?
- **RQ2:** How can you reconstruct the component model from the source of an extant microservice-based system?
- **RQ3:** How can you automatically assess conformance to recommended patterns and practices in microservice-based systems and IaC-based deployments?
- **RQ4:** How can you provide actionable options for improving the system architecture as part of a feedback loop?

3.3 Research Problems

In order to support the architecture evolution of microservice-based systems and IaC-based deployments, it is essential to tackle a range of issues pertaining to gathering, modeling, analyzing, assessing, refactoring, and comprehending both the existing practices and the theoretical foundations of the microservice domain. The following research problems (P) are addressed in this doctoral thesis:

- **P1** *Lack of codification of architecture knowledge in form of a reusable architectural design decision model.*

  The problem is related to RQ1. Numerous patterns and practices for managing data in microservice architectures have been suggested. However, these concepts are mostly discussed informally in "grey literature" such as practitioner blogs, experience reports, and system documentations. Consequently, the knowledge related to microservice architecture is spread across several sources, which are often based on individual experiences, lack consistency, and when examined independently, do not offer a complete understanding.

- **P2** *Lack of accurate architecture model abstraction of systems that are highly polyglot.*

  The problem is related to RQ2. Preventing architecture model drift and erosion is a major challenge in software architecture, especially in complex software systems. Microservice-based systems introduce additional challenges as they frequently use a wide range of technologies in their latest version and undergo frequent changes and releases. The existing solutions for reconstructing architecture models do not adequately address these new challenges, as they are not suited to continuous evolution, have low accuracy, and lack support for highly polyglot environments.

- **P3** *Lack of assessing architecture conformance to best patterns and practices in microservices and IaC-based deployments.*

  The problem is related to RQ3 and RQ4. One of the fundamental principles for a thriving microservice architecture is the strong autonomy of individual microservices, i.e. loose coupling. While several patterns and best practices are established in the literature, most microservice-based systems, either entirely or partially, do not conform to them. Evaluating this conformity manually is not a practical option for large-scale systems.

- **P4** *Lack of a unified framework for modelling and analyzing microservice-based systems and IaC-based deployments.*

  This problem is related to RQ1, RQ2, and RQ3 and concerns the key components needed for establishing a meta-model for modeling microservice-based systems. It is also related to the essential elements required for reconstructing existing microservice-based systems, such as connector and component types, relationships, and associated technologies.

- **P5** *Lack of automatic violation detection and fix suggestion in microservice-based architectures and IaC-based deployments.*

  This problem pertains to RQ3 and RQ4 and concerns potential architecture violations resulting from design decisions, which can be detected through automatic means. It also concerns methods of guiding architects in rectifying these violations via continuous feedback while retaining flexibility to consider other architecture trade-offs. Moreover, it concerns the possible solutions to these violations and ways to support architects in selecting and implementing the appropriate fixes.

3.4 Research Contributions

This section offers a summary of the contributions presented in this doctoral thesis, connected to their respective research problems outlined in Section 3.3. The contributions were published in conferences and scientific journals such as International Conference on Software Architecture (ICSA), European Conference on Software Architecture (ECSA), International Conference on Services Computing (SCC), International Conference on Cloud Computing (CLOUD), International Conference on Service-Oriented Computing (ICSOC) and the Computing journal. The contributions (C) published in the course of this doctoral thesis are:

- **C1** *Reusable architectural design decision model.*

  The contribution addresses P1 by providing a pattern catalog with all the related data management patterns and practices, their relations and the corresponding impact/drivers (Chapter 4).

- **C2** *Architecture reconstruction of polyglot systems from the source code.*

  The contribution is part of P2 and P4 and introduces a number of detectors that continuously parse relevant parts of the source code and create model abstractions from it. The detectors are designed to address the polyglot nature of microservice-based systems (Chapter 5).

- **C3** *Automatic assessment based on support or violation of patterns/practices of modeled systems.*

  The contribution addresses P3 by providing an automatic metric-based approach for measuring whether a microservice-based system supports the best patterns and practices and to what degree it does so (Chapters 6, 7, 8 and 9).

- **C4** *Modeling microservice-based systems and IaC-based deployments.*

  The contribution is part of P3 and P4 by performing an iterative study of a variety of microservice-related knowledge sources, in order to gradually refine a meta-model, and modeling a number of model instances (microservice-based systems and IaC-based deployments) in order to investigate the ontology, and to evaluate the meta-model's efficiency (Chapters 5, 6, 7, 8 and 9).

- **C5** *Automatic decision-based violation detection.*

  The contribution addresses P3 and P5 by introducing detectors that are able to localize a decision-based violation and in combination with the metrics introduced to address P3, the detectors return the elements (components and connectors) that are involved in a specific violation (Chapters 10, 11 and 12).

- **C6** *Automatic model generation based on the recommended fixes.*

  The contribution addresses P5 by providing a number of actionable options (fixes) for each violation to improve the system architecture as part of a feedback loop (Chapters 10, 11 and 12).

3.5 List of Publications

The content of this thesis is primarily derived from research that has either been published in various scientific outlets such as workshops, conferences, and journals, or is currently under review for submission to a research venue. We associate the publications with the corresponding chapter. The following publications were used in its composition:

- Evangelos Ntentos, Uwe Zdun, Ghareeb Falazi, Uwe Breitenbücher, Frank Leymann "Detecting and Resolving Coupling-Related Infrastructure as Code Based Architecture Smells in Microservice Deployments" IEEE CLOUD 2023, 2-8 July 2022, Chicago, Illinois USA (2023)
  - DOI: https://doi.org/10.5281/zenodo.7737931 (Chapter 12)
- Evangelos Ntentos, Uwe Zdun, Jacopo Soldani, Antonio Brogi "Assessing Architecture Conformance to Coupling-Related Infrastructure-as-Code Best Practices: Metrics and Case Studies" 16th European Conference on Software Architecture, 19.09.2022 - 23.09.2022, Prague, Czech Republic (2022)
  - DOI: https://doi.org/10.5281/zenodo.6801247 (Chapter 8)
- Evangelos Ntentos, Uwe Zdun, Ghareeb Falazi, Uwe Breitenbücher, Frank Leymann "Assessing Architecture Conformance to Security-Related Practices in Infrastructure as Code Based Deployments" IEEE International Conference on Services Computing (SCC 2022), 11-16 July 2022, Barcelona, Spain (2022)
  - DOI: https://doi.org/10.5281/zenodo.6694962 (Chapter 9)

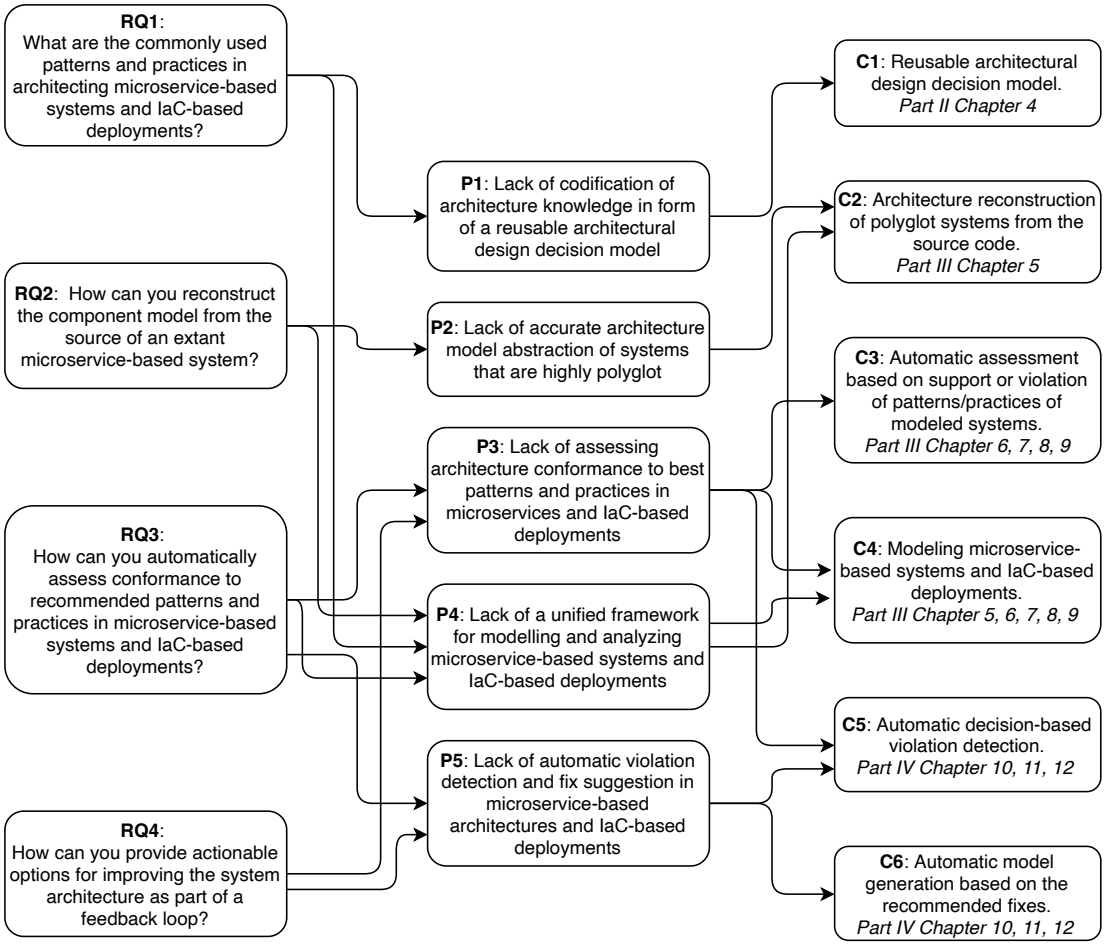

Figure 3.1: Research Overview

- Ghareeb Falazi, Uwe Breitenbücher, Frank Leymann, Miles Stötzner, Evangelos Ntentos, Uwe Zdun, Martin Becker, Elena Heldwein "On Unifying the Compliance Management of Applications Based on IaC Automation" 1st International Workshop on the Foundations of Infrastructure Specification and Testing, 12 March 2022, Virtual (2022)
  - DOI: https://doi.org/10.5281/zenodo.7143512
- Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Sebastian Geiger "Evaluating and Improving Microservice Architecture Conformance to Architectural Design Decisions" Service-Oriented Computing - 19th International Conference, ICSOC 2021, November 22-25, Dubai, United Arab Emirates (2021)
  - DOI: https://doi.org/10.5281/zenodo.7124916 (Chapter 11)
- Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Patric Genfer, Sebastian Geiger, Sebastian Meixner, Wilhelm Hasselbring "Detector-based component model abstraction for

microservice-based systems" Computing, 103 pp. 2521-2551 ISSN 0010-485X Springer (2021)

  - DOI: https://doi.org/10.5281/zenodo.5235931 (Chapters 5)

- Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Sebastian Geiger "Semi-automatic Feedback for Improving Architecture Conformance to Microservice Patterns and Practices" 18th IEEE International Conference on Software Architecture (ICSA 2021), 22 - 26 March, Stuttgart, Germany (2021)

  - DOI: https://doi.org/10.5281/zenodo.5724082 (Chapter 10)

- Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Sebastian Meixner, Sebastian Geiger "Metrics for Assessing Architecture Conformance to Microservice Architecture Patterns and Practices" 18th International Conference on Service Oriented Computing (ICSOC 2020), 14-17 Dec 2020, Dubai (2020)

  - DOI: https://doi.org/10.5281/zenodo.4551448 (Chapter 7)

- Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Sebastian Meixner, Sebastian Geiger "Assessing Architecture Conformance to Coupling-Related Patterns and Practices in Microservices" 14th European Conference on Software Architecture (ECSA), 2020, 14-18 Sep 2020, L'Aquila, Italy (2020)

  - DOI: https://doi.org/10.5281/zenodo.4551524 (Chapter 6)

- Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Daniel Schall, Fei Li, Sebastian Meixner "Supporting Architectural Decision Making on Data Management in Microservice Architectures" 13th European Conference on Software Architecture (ECSA) - 2019, 9-13 September 2019, Paris, France (2019)

  - DOI: https://doi.org/10.5281/zenodo.3484435 (Chapter 4)

- Uwe Zdun, Evangelos Ntentos, Konstantinos Plakidas, Amine El Malki, Daniel Schall, Fei Li "On the Design and Architecture of Deployment Pipelines in Cloud- and Service-Based Computing – A Model-Based Qualitative Study" 2019 IEEE International Conference on Services Computing (SCC 2019), 8-13 July 2019, Milan, Italy (2019)

- Christoph Czepa, Amirali Amiri, Evangelos Ntentos, Uwe Zdun "Modeling compliance specifications in linear temporal logic, event processing language and property specification patterns: a controlled experiment on understandability" Software and Systems Modeling, 18 pp. 3331-3371 ISSN 1619-1366 Springer (2019)

  - DOI: 10.1007/s10270-019-00721-4

3.6 Architecture Evaluation and Optimization

As illustrated in Figure 3.3, our approach consists of one main role and 3 major phases as well as other additional factors. Initially, we investigate methods and techniques related to *system*

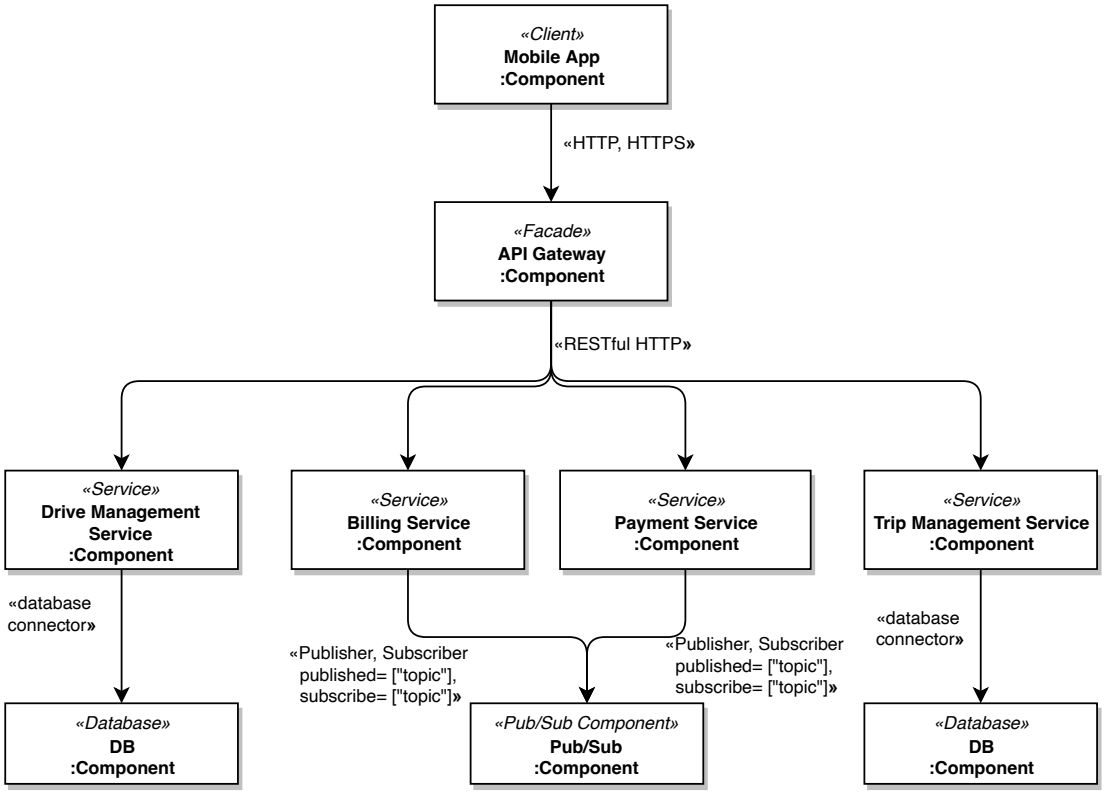

Figure 3.2: Example of Microservice Component Model

architecture abstraction. For this process we utilize real systems, as well as the generic *component meta-model*, which contains all the necessary stereotypes to reconstruct system architecture and generate a *component model* such as the one in Figure 3.2. Additionally, the architect manually specifies the architecture abstraction specification for a software system which is used also as an input for the *system architecture abstraction* phase (see Chapter 5). In the *architecture conformance assessment* phase, based on the generated system *component model*, an automatic assessment of conformance to microservice patterns and practices, outlined in Chapter 4, is performed, using a set of generic, technology independent metrics detailed in Chapters 6, 7, 8, and 9, as well as a detection of violations to recommended microservice and IaC patterns and practices. This returns a feedback to the architect concerning the system architecture quality and the possible violations. Subsequently, a semi-automatic *architecture refactoring* phase is performed (see Chapters 10, 11 and 12). The architect selects an optimal provided solution based on a catalog of possible fixes for decisions, decision options, and metric violations. Then, a new model with the applied fixes is generated and can follow the same process.

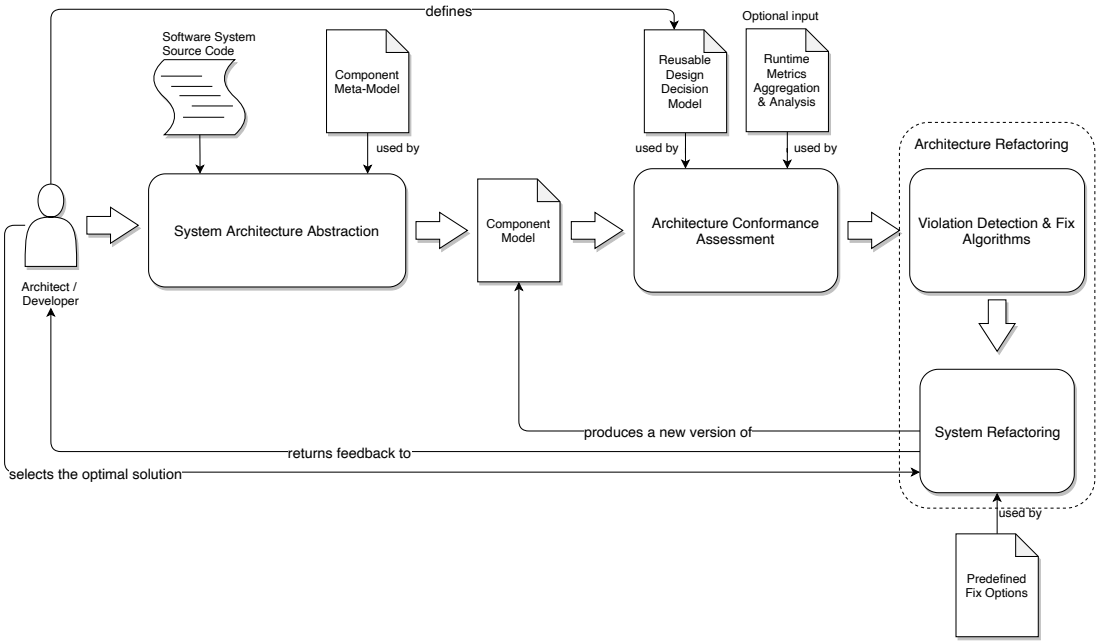

Figure 3.3: Architecture Evaluation and Optimization Workflow

3.7 Research Methods

In order to answer RQ1, we conducted a systematic study of established practices in the field of architecting microservice-based systems. This study combines the established Grounded Theory (GT) [GS67] qualitative research approach with techniques for studying established practices, such as pattern mining [Cop96] and its combination with GT [HZHD15]. The GT approach involves a systematic and iterative process of data collection and analysis, with the goal of developing a theory that reflects the experiences and perspectives of the study population. The data collected through this method is analyzed through various steps, including open coding, axial coding, and selective coding, to identify patterns and relationships and generate a theoretical framework that helps explain the dynamics and processes involved in the development and adoption of microservice-based systems. We began with the collection of descriptions of established practices from the authors' own experiences. We then searched for a limited number of credible, technically-detailed sources from the so-called "grey literature" (e.g. practitioner reports, system documentations, practitioner blogs, etc.). These sources provided unbiased descriptions of established practices in the field and served as the basis for our further analysis. Furthermore, we followed many of the principles of the classic GT approach, with a few key differences. Similar to GT, we involved a thorough examination of each knowledge source and a systematic coding process, as well as a constant comparison procedure to develop a model. In classic GT, the data analysis process primarily involves the use of textual analysis to identify patterns and relationships in the data. In our study, while textual codes were used initially, they were then transformed into formal software models. This distinction is what sets our method

apart from classic GT and gives it its "model-based" character.

To answer RQ2 we used design science research to analyze the design of artifacts in a specific context. The research process involves several design and engineering cycles, each comprising four steps: problem investigation, treatment design, treatment validation, and design implementation. If the evaluation of a cycle is unsatisfactory, subsequent cycles may be conducted to improve the design. Design implementation is optional, and our study did not carry it out. Empirical evaluation based on a case study was used to validate the treatment, which is one of the various empirical methods recommended by Wieringa et al. [Wie14].

To answer RQ3, it was necessary to collect a sufficient number of microservice-based systems that have been developed by practitioners and published in public repositories and practitioners' blogs. We use these systems as case studies to evaluate the respective approaches. Case studies are often regarded as "typical research" since they observe and analyze activities, processes, and other aspects of real-world projects. Yin [Yin02] defines case studies as empirical investigations of current phenomena in their authentic settings, particularly when it's unclear where the boundary between the phenomenon and context lies. The primary objective of conducting case studies is to comprehend the hows and whys of specific phenomena within a defined time frame and to identify the mechanisms that establish various cause-and-effect relationships [WHH03]. Table 3.1 shows the case studies that served as the main dataset for the approaches in Chapters 6, 7, 10 and 11.

To create the system component models, the decision models and the meta-model, we used our existing modeling tool CodeableModels[1], a Python implementation for the precise specification of meta-models, models, and model instances in code. CodeableModels allows for the specification of meta-models for components, connectors, and relationships, making it an ideal tool for our study. The next step in the process was to manually create model instances for each of the collected systems. This involved utilizing automated constraint checkers and PlantUML code generators to generate graphical visualizations of all meta-models and models. This process was iterative, allowing for refinement and testing of the meta-model's ontology until no further additions were necessary. The collection of microservice-based systems, along with the use of CodeableModels, allowed us to create models and a meta-model that accurately reflected the existing practices in the field. The iterative refinement process ensured that the meta-model's ontology was comprehensive and accurate, making it a valuable tool for future research in the field.

For the theoretical basis of RQ3 and RQ4, we conducted a comprehensive multi-vocal literature study. This involved exploring all possible sources of knowledge, including web resources, public repositories, and scientific papers, as outlined in [GFM17]. Additionally, we employed an iterative trial-and-error process to create automatic tools that would support the specific tasks outlined in the RQs. This involved continual refinement and testing until the tools were optimized for the tasks at hand. Where statistical evaluation of our results was necessary, we utilized robust statistical methods and relevant R packages to ensure the validity and reliability of our findings. By combining the multi-vocal literature study with the use of automatic tools and robust statistical methods, we were able to provide a comprehensive and rigorous examination of the theoretical basis of microservice-based systems and IaC-based deployments.

[1] https://github.com/uzdun/CodeableModels

Model ID	*Model Size*	*Description / Source*
BM1	10 components 14 connectors	Banking-related application based on CQRS and event sourcing (from https://github.com/cer/event-sourcing-examples).
BM2	8 components 9 connectors	Variant of BM1 which uses direct RESTful completely synchronous service invocations instead of event-based communication.
BM3	8 components 9 connectors	Variant of BM1 which uses direct RESTful completely asynchronous service invocations instead of event-based communication.
CO1	8 components 9 connectors	The common component model E-shop application implemented as microservices directly accessed by a Web frontend (from https://github.com/cocome-community-case-study/cocome-cloud-jee-microservices-rest).
CO2	11 components 17 connectors	Variant of CO1 using a SAGA orchestrator on the order service with a message broker. Added support for Open Tracing. Added an API gateway.
CO3	9 components 13 connectors	Variant of CO1 where the reports service does not use inter-service communication, but a shared database for accessing product and store data. Added support for Open Tracing.
CI1	11 components 12 connectors	Cinema booking application using RESTful HTTP invocations, databases per service, and an API gateway (from https://codeburst.io/build-a-nodejs-cinema-api-gateway-and-deploying-it-to-docker-\part-4-703c2b0dd269).
CI2	11 components 12 connectors	Variant of CI1 routing all interservice communication via the API gateway.
CI3	10 components 11 connectors	Variant of CI1 using direct client to service invocations instead of the API gateway.
CI4	11 components 12 connectors	Variant of CI1 with a subsystem exposing services directly to the client and another subsystem routing all traffic via the API gateway.
EC1	10 components 14 connectors	E-commerce application with a Web UI directly accessing microservices and an API gateway for service-based API (from https://microservices.io/patterns/microservices.html).
EC2	11 components 14 connectors	Variant of EC1 using event-based communication and event sourcing internally.
EC3	8 components 11 connectors	Variant of EC1 with a shared database used to handle all but one service interaction.
ES1	20 components 36 connectors	E-shop application using pub/sub communication for event-based interaction, a middleware-triggered identity service, databases per service (4 SQL DBs, 1 Mongo DB, and 1 Redis DB), and backends for frontends for two Web app types and one mobile app type (from https://github.com/dotnet-architecture/eShopOnContainers).
ES2	14 components 35 connectors	Variant of ES1 using RESTful communication via the API gateway instead of event-based communication and one shared SQL DB for all 6 of the services using DBs. No service interaction via the shared database occurs.
ES3	16 components 35 connectors	Variant of ES1 using RESTful communication via the API gateway instead of event-based communication and one shared database for all 4 of the services using SQL DB in ES1. However, no service interaction via the shared database occurs.
FM1	15 components 24 connectors	Simple food ordering application based on entity services directly linked to a Web UI (from https://github.com/jferrater/Tap-And-Eat-MicroServices).
FM2	14 components 21 connectors	Variant of FM1 which uses the store service as an API composition and asynchronous interservice communication. Added Jaeger-based tracing per service.
FM3	13 components 15 connectors	Variant of FM1 which demonstrates a cyclic dependency case, uses the store service as an API composition and asynchronous inter-service communication.
HM1	13 components 25 connectors	Hipster shop application using GRPC interservice connection and OpenCensus monitoring & tracing for all but one service as well as on the gateway (from https://github.com/GoogleCloudPlatform/microservices-demo).
HM2	14 components 26 connectors	Variant of HM1 that uses publish/subscribe interaction with event sourcing, except for one service, and realizes the tracing on all services.
RM1	11 components 18 connectors	Restaurant order management application based on SAGA messaging and domain event interactions. Rudimentary tracing support (from https://github.com/microservices-patterns/ftgo-application).
RM2	14 components 14 connectors	Variant of RM1 which contains transitively shared services, API Gateway for client services communication, database per service and direct communication between service.
RM3	14 components 15 connectors	Variant of RM1 which demonstrates a cyclic dependency case, API Gateway for client services communication, database per service and direct communication between service.
RS	18 components 29 connectors	Robot shop application with various kinds of service interconnections, data stores, and Instana tracing on most services (from https://github.com/instana/robot-shop).
TH1	14 components 16 connectors	Taxi hailing application with multiple frontends and databases per services from (https://www.nginx.com/blog/introduction-to-microservices/).
TH2	15 components 18 connectors	Variant of TH1 that uses publish/subscribe interaction with event sourcing for all but one service interaction.

Table 3.1: Overview of modelled systems used in our studies (size, details, and sources)

Part II

Decision-Making Support for Microservice Architectures

4 Supporting Architectural Decision Making on Data Management in Microservice Architectures

As described in the previous chapter in Section 1.1 managing data in microservice architectures is crucial and requires careful consideration. Although many patterns and practices for microservice data management have been proposed, they are often informally discussed in practitioner blogs, experience reports, and system documentation. In this chapter we present a qualitative study of 35 practitioner descriptions of best practices and patterns in microservice data management architectures. Using a model-based qualitative research method, we developed a formal architecture decision model with 325 elements and relations.

4.1 Introduction

Microservice architectures [New15, Zim17] have emerged from established practices in service-oriented computing (cf. [PJ16, Ric17, ZKL+09]). The microservices approach emphasizes business capability-based and domain-driven design, development in independent teams, cloud-native technologies and architectures, polyglot technology stacks including polyglot persistence, lightweight containers, loosely coupled service dependencies, and continuous delivery (cf. [LF04, New15, Zim17]). Some of these tenets introduce substantial challenges for the data management architecture. Notably, it is usually advised to decentralize all data management concerns. Such an architecture requires, in addition to the already existing non-trivial design challenges intrinsic in distributed systems, sophisticated solutions for data integrity, data querying, transaction management, and consistency management [New15, Zim17, PJ16, Ric17].

Many authors have written about microservice data management and various attempts to document microservice patterns and best practices exist [Ric17, Gup17, LF04, PJ16]. Nevertheless, most of the established practices in industry are only reported in the so-called "grey literature", consisting of practitioner blogs, experience reports, system documentations, etc. In most cases, each of these sources documents a few existing practices well, but usually they do not provide systematic architectural guidance. Instead the reported practices are largely based on personal experience, often inconsistent, and, when studied on their own, incomplete. This creates considerable uncertainty and risk in architecting microservice data management, which can be reduced either through substantial personal experience or by a careful study of a large set of knowledge sources. Our aim is to complement such knowledge sources with an unbiased, consistent, and more complete view of the current industrial practices than readily available today.

To reach this goal, we have performed a qualitative, in-depth study of 35 microservice data practice descriptions by practitioners containing informal descriptions of established practices

and patterns in this field. We have based our study on the model-based qualitative research method described in [ZSZ+18]. It uses such practitioner sources as rather unbiased (from our perspective) knowledge sources and systematically codes them through established coding and constant comparison methods [GS67], combined with precise software modeling, in order to develop a rigorously specified software model of established practices, patterns, and their relations. This work aims to study the following research questions:

- **RQ1.1** What are the patterns and practices currently used by practitioners for supporting data management in a microservice architecture?
- **RQ1.2** How are the current microservice data management patterns and practices related? In particular, which architectural design decisions (ADDs) are relevant when architecting microservice data management?
- **RQ1.3** What are the influencing factors (i.e., decision drivers) in architecting microservice data management in the eye of the practitioner today?

This work makes three major contributions. First, we gathered knowledge about established industrial practices and patterns, their relations, and their decision drivers in the form of a *qualitative study on microservice data management architectures*, which included 35 knowledge sources in total. Our second contribution is the codification of this knowledge in form of a *reusable architectural design decision (ADD) model* in which we formally modeled the decisions based on a UML2 meta-model. In total we documented 9 decisions with 30 decision options and 34 decision drivers. Finally, we *evaluated the level of detail and completeness of our model* to support our claim that the chosen research method leads to a more complete treatment of the established practices than methods like informal pattern mining. For this we compared to the by far most complete of our pool of sources, the *microservices.io* patterns catalog [Ric17], and are able to show that our ADD model captures 210% more elements and relations.

The remainder of this chapter is organized as follows: In Section 4.2 we compare to the related work. Section 4.3 explains the research methods we have applied in our study and summarizes the knowledge sources. Section 4.4 describes our reusable ADD model on microservice data management. Section 4.5 compares our study with *microservices.io* in terms of completeness. Finally, Section 4.6 discusses the threats to validity of our study and Section 4.7 summarizes our findings.

4.2 Related Work

A number of approaches that study microservice patterns and best practices exist: The *microservices.io* collection by Richardson [Ric17] addresses microservice design and architecture practices. As the work contains a category on data management, many of them are included in our study. Another set of patterns on microservice architecture structures has been published by Gupta [Gup17], but those are not focused on data management. Microservice best practices are discussed in [LF04], and similar approaches are summarized in a recent mapping study [PJ16]. So far, none of those approaches has been combined with a formal model; our ADD model complements these works in this sense.

Decision documentation models that promise to improve the situation exist (e.g. for service-oriented solutions [ZKL+09], service-based platform integration [LSZ12], REST vs. SOAP [PZL08], and big data repositories [GKN15]). However, this kind of research does not yet encompass microservice architectures, apart from our own prior study on microservice API quality [ZSZ+18]. The model developed in our study can be classified as a reusable ADD model, which can provide guidance on the application of patterns [ZKL+09]. Other authors have combined decision models with formal view models [vHAH12]. We apply such techniques in our work, but also extend them with a formal modeling approach based on a qualitative research method.

4.3 Research Method and Modelling Tool

Research Method. This work aims to systematically study the established practices in the field of architecting data management in microservice architectures. We follow the model-based qualitative research method described in [ZSZ+18]. It is based on the established Grounded Theory (GT) [GS67] qualitative research method, in combination with methods for studying established practices like pattern mining (see e.g. [Cop96]) and their combination with GT [HZHD15]. The method uses descriptions of established practices from the authors' own experiences as a starting point to search for a limited number of well-fitting, technically detailed sources from the so-called "grey literature" (e.g., practitioner reports, system documentations, practitioner blogs, etc.). These sources are then used as unbiased descriptions of established practices in the further analysis. Like GT, the method studies each knowledge source in depth. It also follows a similar coding process, as well as a constant comparison procedure to derive a model. In contrast to classic GT, the research begins with an initial research question, as in Charmaz's constructivist GT [Cha14]. Whereas GT typically uses textual analysis, the method uses textual codes only initially and then transfers them into formal software models (hence it is model-based).

The knowledge-mining procedure is applied in many iterations: we searched for new knowledge sources, applied open and axial coding [GS67] to identify candidate categories for model elements and decision drivers, and continuously compared the new codes with the model designed so far to incrementally improve it. A crucial question in qualitative methods is when to stop this process. Theoretical saturation [GS67] has attained widespread acceptance for this purpose. We stopped our analysis when 10 additional knowledge sources did not add anything new to our understanding of the research topic. While this is a rather conservative operationalisation of theoretical saturation (i.e., most qualitative research saturates with far fewer knowledge sources that add nothing new), our study converged already after 25 knowledge sources. The sources included in the study are summarized in Table 4.1. Our search for sources was based on our own experience, i.e., tools, methods, patterns and practices we have access to, worked with, or studied before. We also used major search engines (e.g., Google, Bing) and topic portals (e.g., InfoQ) to find more sources.

Modelling Tool Implementation. To create our decision model, we used the Python modeling library CodeableModels described in Chapter 3.7.

Table 4.1: Knowledge Sources Included in the Study

Name	Description	Reference
S1 [2]	Intro to Microservices: Dependencies and Data Sharing	https://bit.ly/2YTnolQ
S2 [1]	Pattern: Shared database	https://bit.ly/30L1PW2
S3 [4]	Enterprise Integration Patterns	https://bit.ly/2WrlOHC
S4 [2]	Design Patterns for Microservices	https://bit.ly/2EBmIcQ
S5 [2]	6 Data Management Patterns for Microservices	https://bit.ly/2K3YMTb
S6 [1]	Pattern: Database per service	https://bit.ly/2EDDici
S7 [2]	Transaction Management in Microservices	https://bit.ly/2XSKhWL
S8 [2]	A Guide to Transactions Across Microservices	https://bit.ly/2WpQN9j
S9 [2]	Saga Pattern – How to implement business transactions using Microservices	https://bit.ly/2WpRBuR
S10 [2]	Saga Pattern and Microservices architecture	https://bit.ly/2HF6G3G
S11 [2]	Patterns for distributed transactions within a microservices architecture	https://bit.ly/2QqZqUx
S12 [2]	Data Consistency in Microservices Architecture	https://bit.ly/2K5G79y
S13 [2]	Event-Driven Data Management for Microservices	https://bit.ly/2WlSKUs
S14 [1]	Pattern: Saga	https://bit.ly/2WpS549
S15 [2]	Managing Data in Microservices	https://bit.ly/2HYIvvY
S16 [2]	Event Sourcing, Event Logging – An essential Microservice Pattern	https://bit.ly/2QusIcb
S17 [1]	Pattern: Event sourcing	https://bit.ly/2K62TOn
S18 [2]	Microservices With CQRS and Event Sourcing	https://bit.ly/2JK2IZQ
S19 [2]	Microservices Communication: How to Share Data Between Microservices	https://bit.ly/2HCR94u
S20 [2]	Building Microservices: Inter-Process Communication in a Microservices Architecture	https://bit.ly/30OVB7U
S21 [1]	Pattern: Command Query Responsibility Segregation (CQRS)	https://bit.ly/2X80LcM
S22 [3]	Data considerations for microservices	https://bit.ly/2WrLeav
S23 [2]	Preventing Tight Data Coupling Between Microservices	https://bit.ly/2WptQmJ
S24 [3]	Challenges and solutions for distributed data management	https://bit.ly/2wp5YkO
S25 [3]	Communication in a microservice architecture	https://bit.ly/2X7UDkT
S26 [2]	Microservices: Asynchronous Request Response Pattern	https://bit.ly/2WjAFqb
S27 [2]	Patterns for Microservices — Sync vs. Async	https://bit.ly/2Ezhsqg
S28 [2]	Building Microservices: Using an API Gateway	https://bit.ly/2EA3AfA
S29 [2]	Microservice Architecture: API Gateway Considerations	https://bit.ly/2YUKWqr
S30 [1]	Pattern: API Composition	https://bit.ly/2WlVqS0
S31 [1]	Pattern: Backends For Frontends	https://bit.ly/2X9I3kQ
S32 [3]	Command and Query Responsibility Segregation (CQRS) pattern	https://bit.ly/2wltdMq
S33 [2]	Introduction to CQRS	https://bit.ly/2HY0sLm
S34 [2]	CQRS	https://bit.ly/2JKI2Rz
S35 [2]	Publisher-Subscriber pattern	https://bit.ly/2JGtqCx

[1] denotes a source taken from *microservices.io*
[2] practitioner blog
[3] Microsoft technical guide
[4] book chapter

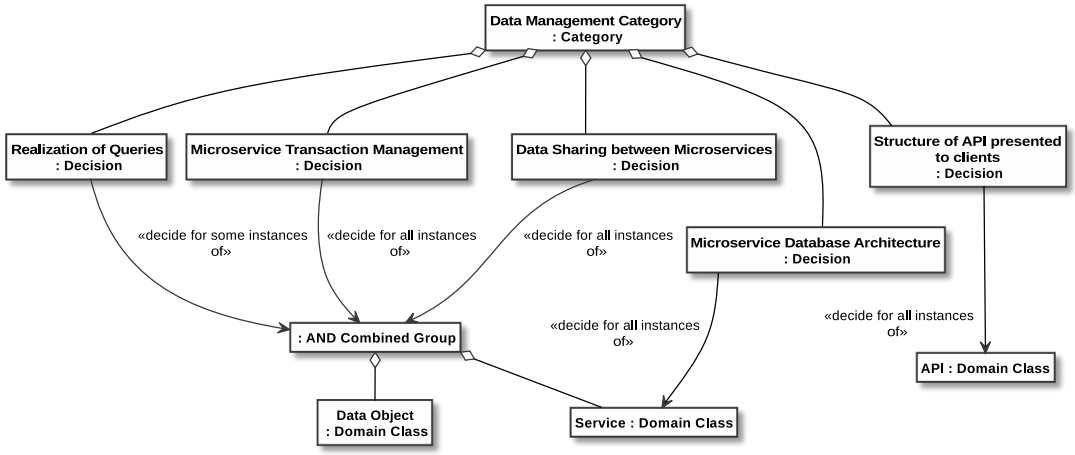

Figure 4.1: Reusable ADD Model on Microservice Data Management: Overview

4.4 Reusable ADD model for data management in microservice architectures

In this section, we describe the reusable ADD model derived from our study [1]. All elements of the model are *emphasized* and all decision drivers derived from our sources in Table 4.1 are *slanted*. It contains one decision category, *Data Management Category*, relating five top-level decisions, as illustrated in Fig. 4.1. These decisions need to be taken for the decision contexts *all instances of* an *API*, *Service* instances, or the combination of *Data Object*s and *Service* instances, respectively. Note that all elements of our model are instances of a meta-model (with meta-classes such as *Decision*, *Category*, *Pattern*, *AND Combined Group*, etc.), which appear in the model descriptions. Each of them is described in detail below (some elements may be relevant for more than one decision, but this has been omitted from the figures for ease of presentation).

Microservice Database Architecture (Fig. 4.2).

Since most software relies on efficient data management, database architecture is a central decision in the design of a microservice architecture. Quality attributes such as performance, reliability, coupling, and scalability, need to be carefully considered in the decision making process. The simplest decision option is to choose *service stores no persistent data*, which is applicable only for services whose functions are performed solely on transient data, like pure calculations or simple routing functions. By definition, a microservice should be autonomous, loosely coupled and able to be developed, deployed, and scaled independently [LF04]. This is ensured by the *Database per Service* pattern [Ric17], which we encountered, either directly or implicitly, in 33 out of 35 sources: each microservice manages its own data, and data exchange and communications with other services are realized only through a set of well-defined APIs. When choosing this option, transaction management between services becomes more difficult,

[1] Replication package can be found at: https://bit.ly/2EKyTnL

as the data is distributed across the services; for the same reason making queries could become a challenge, too. Thus the optional next decisions on *Microservice Transaction Management* (see sources [S7, S8, S11]) and *Realization of Queries* [Ric17] should be considered (both explained below). The use of this pattern may also require a next decision on the *Need for Data Composition, Transformation, or Management*. Another option, which is recommended only for special cases (e.g., when a group of services always needed to share a data object), is to use a *Shared Database* [Ric17](see sources [S1, S19]): all involved services persist data in one and the same database.

There are a number of criteria that determine the outcome of this decision. Applying the *Database per Service* pattern in a system results in more *loosely coupled* microservices. This leads to better *scalability* than a *Shared Database* closer to the service with only transient data, since microservices can scale up individually. Especially for low loads this can reduce *performance*, as additional distributed calls are needed to get data from other services and establish *data consistency*. The pattern's impact on *performance* is not always negative: for high loads a *Shared Database* can become a bottleneck, or database replication is needed. On the other hand, *Shared Database* makes it easier to *manage transactions* and *implement queries and joins*; hence the follow-on decisions for *Database per Service* mentioned above. Furthermore, *Database per Service* facilitates *polyglot persistence*. The *Shared Database* option could be viable only if the *integration complexity* or related challenges of *Database per Service*-based services become too difficult to handle; also, operating a single *Shared Database* is simpler. Though *Shared Database* ensures *data consistency* (since any changes to the data made in a single service are made available to all services at the time of the database commit), it would appear to completely eliminate the targeted benefits of *loose coupling*. This negatively affects both the *development and runtime coupling* and the potential *scalability*.

Structure of API Presented to Clients (Fig. 4.3).

When software is decomposed into microservices, many major challenges lie in the structure of the API. This topic has been extensively studied in our prior and ongoing work on API patterns [ZSZ+18]; here we concentrate only on those decision options relevant to data management. Many issues in microservice design are resolved at the API level, such as routing requests to the appropriate microservice, the distribution of multiple services, and the aggregation of results. The simplest option for structuring a system is *Clients Access Microservices Directly*: all microservices are entry points of the system, and clients can directly request the service they need (each service offers its own API endpoint to clients). However, all studied sources recommend or assume the use of the *API Gateway* pattern [Ric17] as a common entry point for the system, through which all requests are routed. An alternative solution, for servicing different types of clients (e.g., mobile vs. desktop clients) is the *Backends for Frontends* pattern variant [Ric17], which offers a fine-grained API for each specific type of client. An *API Gateway* could also be realized as an *API Composition Service* [Ric17], that is a service which invokes other microservices. Furthermore an *API Gateway* can have *Additional centralized data-related functions* (shown in Fig. 4.3 and discussed below as decision drivers).

The main driver affecting this decision is that *API Gateways* (and thus *API Composition Service* and *Backends for Frontends* in a more limited capacity) can provide a number of centralized

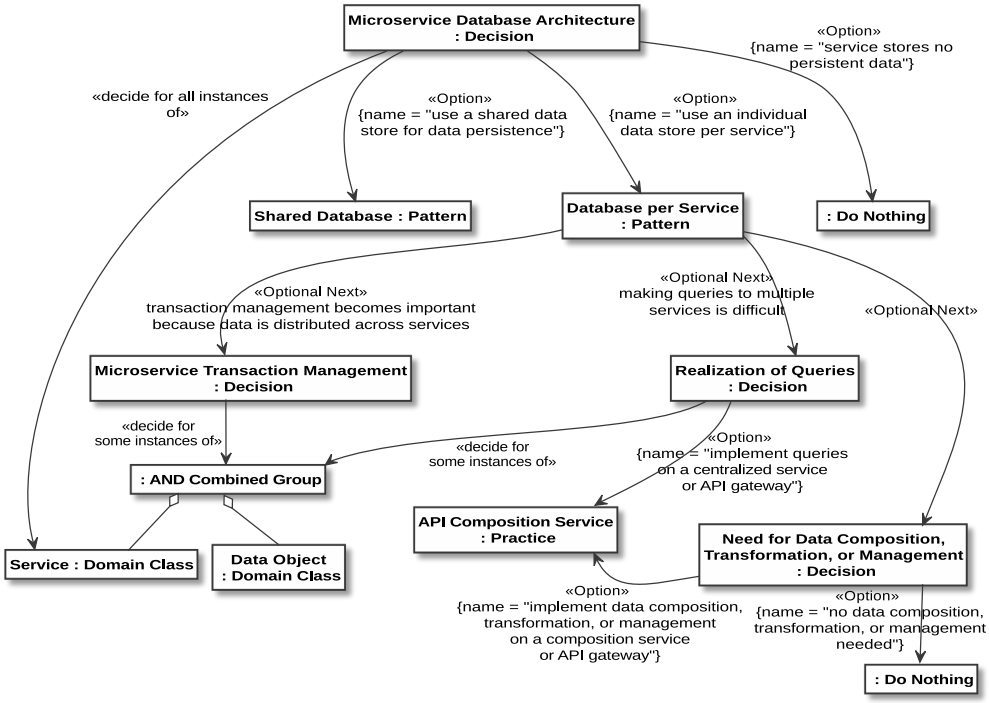

Figure 4.2: Microservice Database Architecture Decision

services. They can work as a proxy service to *route requests* to the appropriate microservice, *convert or transform requests or data* and deliver the *data at the granularity needed by the client*, and provide the *API abstractions for the data needed by the client.* In addition, they can *handle access management* to data (i.e., authentication/authorization), serve as a *data cache*, and *handle partial failures*, e.g. by returning default or cached data. Although its presence increases the overall *complexity* of the architecture since an additional service needs to be developed and deployed, and increases *response time* due to the additional network passes through it, an *API Gateway* is generally considered as an optimal solution in a microservice-based system. *Clients Access Microservices Directly* makes it difficult to realize such centralized functions. A sidecar architecture [sid17] might be a possible solution, but if the service should fail, many functions are impeded, e.g. caching or handling partial failures. The same problem of centralized coordination also applies to a lesser extent to *Backends for Frontends* (centralization in each *API Gateway* is still possible). *Use API Gateway to cache data* reduces the *response time*, returning cached data faster, and increases *data availability*: if a service related to specific data is unavailable, it can return its cached data.

Data Sharing Between Microservices (Fig. 4.4).

Data sharing must be considered for each data object that is shared between at least two microservices. Before deciding how to share data, it is essential to identify the information to be

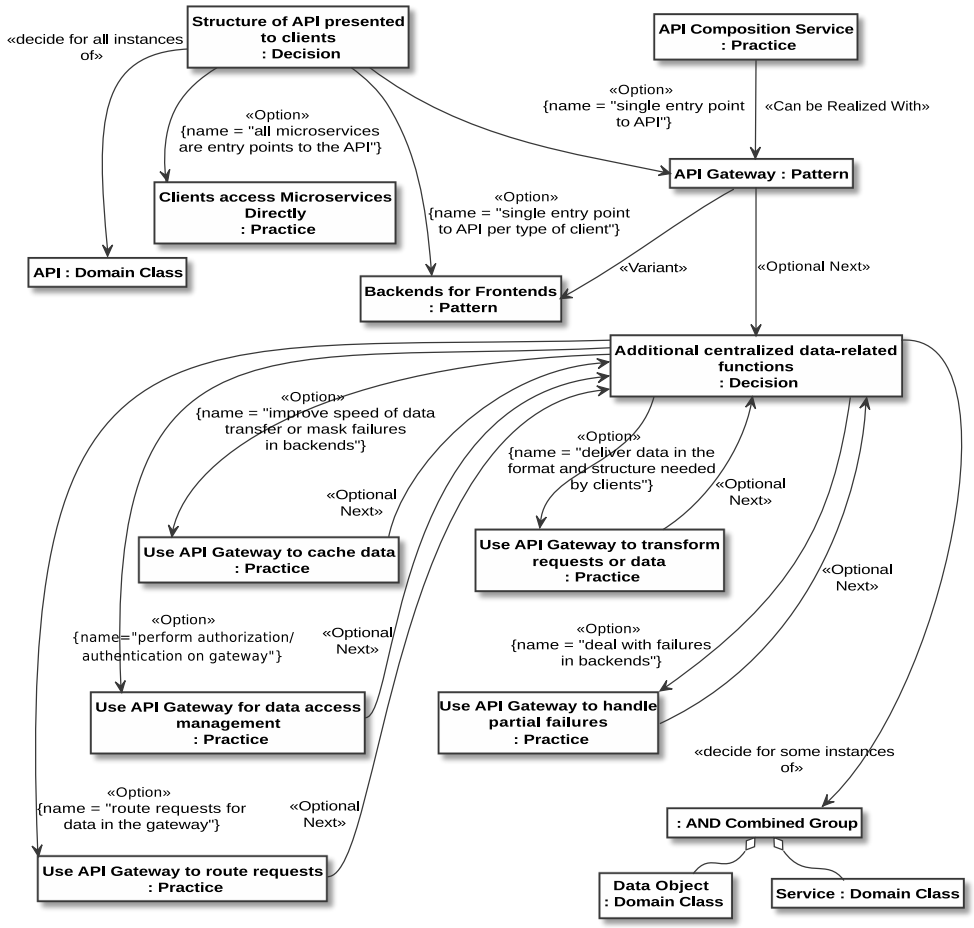

Figure 4.3: Structure of API Presented to Clients Decision

shared, its update frequency, and the primary provider of the data. The decision must ensure that sharing data does not result in tightly coupled services. The simplest option is to choose *services share no data*, which is theoretically optimal in ensuring loose coupling, but is only applicable for rather independent services or those that require only transient data. Another option, already discussed above, is a *Shared Database*. In this solution services share a common database; a service publishes its data, and other services can consume it when required. A number of viable alternatives to the *Shared Database* exist. *Synchronous Invocations-Based Data Exchange* is a simple option for sharing data between microservices. *Request-Response Communication* [HW03b] is a data exchange pattern in which a service sends a request to another service which receives and processes it, ultimately returning a response message. Another typical solution that is well suited to achieving *loose coupling* is to use *Asynchronous Invocations-Based Data Exchange*. Unlike *Request-Response Communication*, it removes the need to wait for a response, thereby decoupling the execution of the communicating services. Implementation of asynchronous communication leads to *Eventual Consistency* [Per17]. There are several possible

Asynchronous Data Exchange Mechanisms: *Publish/Subscribe* [HW03b], in which services can subscribe to an event; use of a *Messaging* [HW03b] middleware; *Data Polling*, in which services periodically poll for data changes in other services; and the *Event Sourcing* [Ric17] pattern that ensures that all changes to application state are stored as a sequence of events.

The choices in this decision are determined by a number of factors. With a *Shared Database*, the system tends to be more *tightly coupled* and *less scalable*. Conversely, an *Asynchronous Data Exchange Mechanism* ensures that the services are more *loosely coupled*, since they interact mostly via events, use message buffering for queuing requests until processed by the consumer, support *flexible* client--service interactions, or provide an explicit inter-process communication mechanism. It has minimal impact on quality attributes related to network interactions, such as *latency* and *performance*. However, *operational complexity* is negatively impacted, since an additional service must be configured and operated. On the other hand, a *Request-Response Communication* mechanism does not require a broker, resulting in a *less complex* system architecture. Despite this, in a *Request-Response Communication*-based system, the communicating services are more *tightly coupled* and the communication is less *reliable*, as they must both be running until the exchange is completed. Applying the *Event Sourcing* pattern increases *reliability*, since events are published whenever state changes, and the system is more *loosely coupled*. Patterns supporting message persistence such as *Messaging*, *Event Sourcing*, and messaging-based *Publish/Subscribe* increase the *reliability* of message transfers and thus the *availability* of the system.

Microservice Transaction Management (Fig. 4.5).

One common problem in microservice-based systems is how to manage distributed transactions across multiple services. As explained above, the *Database per Service* pattern often introduces this need, as the relevant data objects of a transaction are scattered across different services and their databases. Issues concerning transaction atomicity and isolation of user actions for concurrent requests need to be dealt with. One of the easiest and most efficient options to solve the problem of distributed transactions is to completely avoid them. This can be done through a *Shared Database* (with all its drawbacks in a microservice architecture) or by service redesign so that all data objects of the transaction reside in one microservice. If this is not possible, another option is to apply the *Saga Transaction Management* [Ric17] pattern, where each transaction updates data within a single service, in a sequence of local transactions [S9]; every step is triggered only if the previous one has been completed. The implementation requires an additional decision for the *Saga Coordination Architecture*. There are two possible options for implementing this pattern: *Event/Choreography Coordination* and *Command/Orchestration Coordination* [S9]. *Event/Choreography Coordination* is a distributed coordination approach where a service produces and publishes events, that are listened to by other services which then decide their next action. *Command/Orchestration Coordination* is a centralized approach where a dedicated service informs other involved services, through a command/reply mechanism, what operation should be performed. Moreover, *Saga Transaction Management* supports failure analysis and handling using *Event Log* and *Compensation Action* practices [S12]. Implementing this pattern leads also to *Eventual Consistency*. Another typical option for implementing a transaction across different services is to apply the *Two-Phase Commit Protocol* [AhS09] pattern:

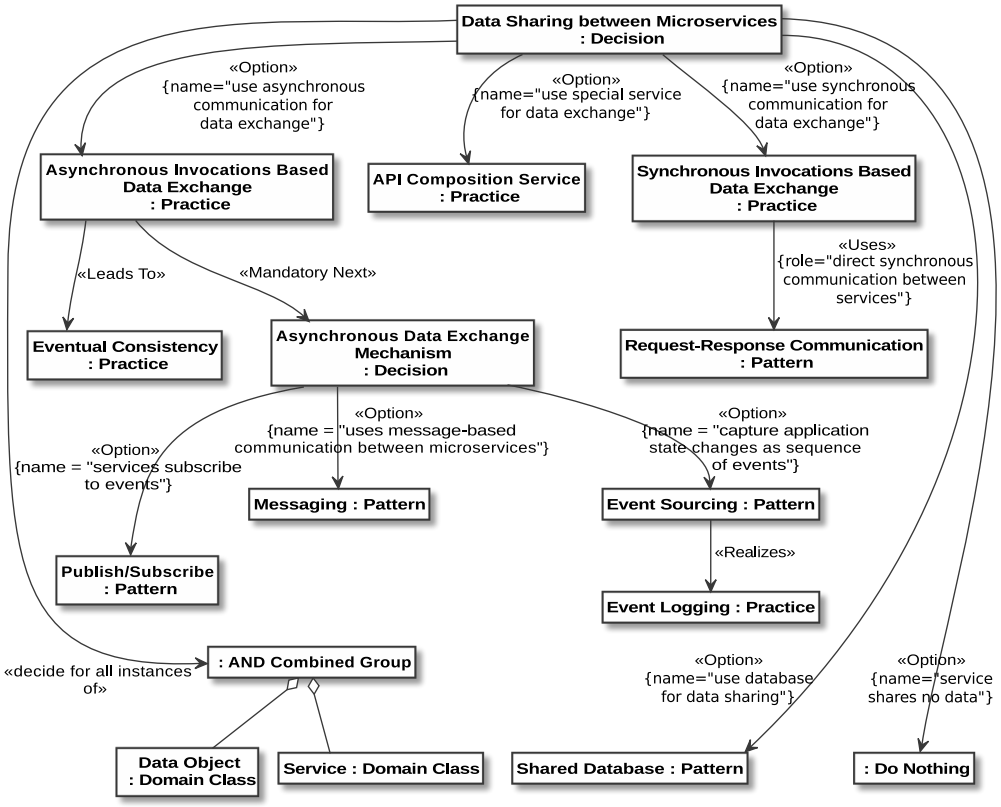

Figure 4.4: Data Sharing Between Microservices Decision

in the first phase, services which are part of the transaction prepare for commit and notify the coordinator that they are ready to complete the transaction; in the second phase, the transaction coordinator issues a commit or a rollback to all involved microservices. Here, the *Rollback* [S7] practice is used for handling failed transactions.

There are a number of criteria that need to be considered in this decision. When implementing the *Saga Transaction Management* pattern, the *Event/Choreography Coordination* option results in a more *loosely coupled* system where the services are more *independent* and *scalable*, as they have no direct knowledge of each other. On the other hand, the *Command/Orchestration Coordination* option has its own advantages: it *avoids cyclic dependencies between services*, *centralizes the orchestration of the distributed transaction*, reduces the participants' *complexity*, and makes rollbacks easier to manage. The *Two-Phase Commit Protocol* pattern is not a typical solution for managing distributed transactions in microservices, but it provides a *strong consistency* protocol, guarantees *atomicity* of transactions, and allows read-write *isolation*. However, it can significantly impair system *performance* in high load scenarios.

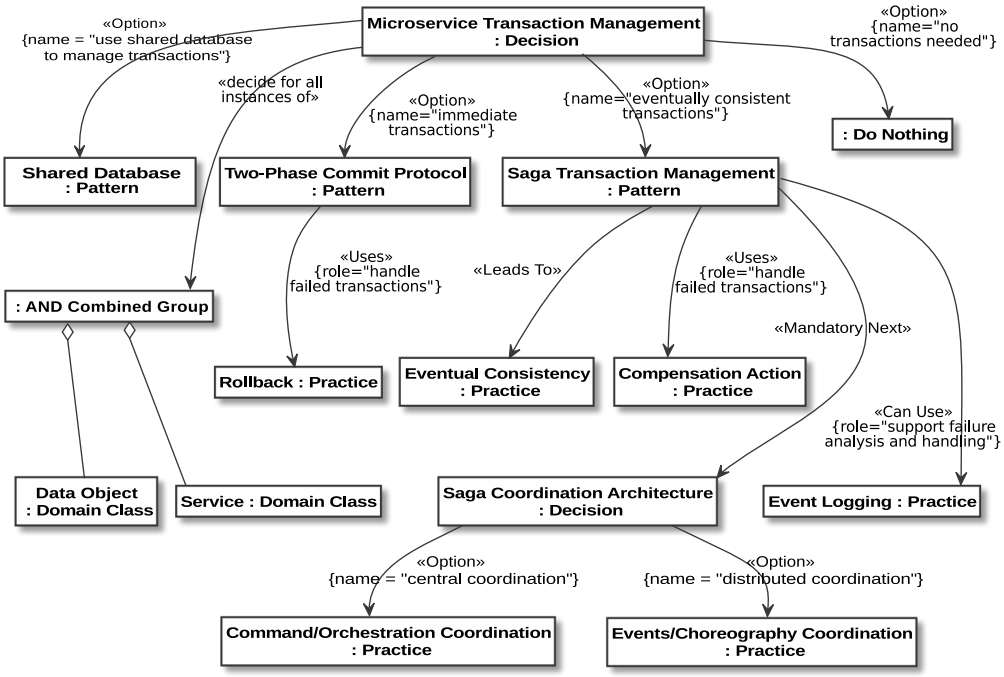

Figure 4.5: Microservice Transaction Management Decision

Realization of Queries (Fig. 4.6).

For every data object and data object combination in a microservice-based system, and its services, it must be considered whether queries are needed. As data objects may reside in different services, e.g., as a consequence of applying *Database per Service*, queries may be more difficult to design and implement than when utilizing a single data source. The simplest option is of course to implement *no queries* in the system, but this is often not realistic. An efficient option for managing queries is to apply the *Command-Query-Responsibility-Segregation* (CQRS) pattern [Fow11]. CQRS is a process of separation between read and write operations into a "command" and a "query" side. The "command" side manages the "create", "update" and "delete" operations; the "query" side segregates the operations that read data from the "update" operation utilizing separated interfaces. This is very efficient if multiple operations are performed in parallel on the same data. The other option is to implement queries in a *API Composition Service* or in the *API Gateway*.

A number of criteria determine the outcome of this decision. The *Command-Query-Responsibility-Segregation* (CQRS) option increases *scalability* since it supports independent horizontal and vertical scaling, improves *security* since the read and write responsibilities are separated. It also increases *availability*: when the "command" side is down the last data update remains available on the "query" side. Despite these benefits, using CQRS has some drawbacks: it adds significant *complexity*, and is not suitable to every system. On the other hand, implementing queries in an *API Composition Service* or *API Gateway* introduces an overhead and decreases *performance*,

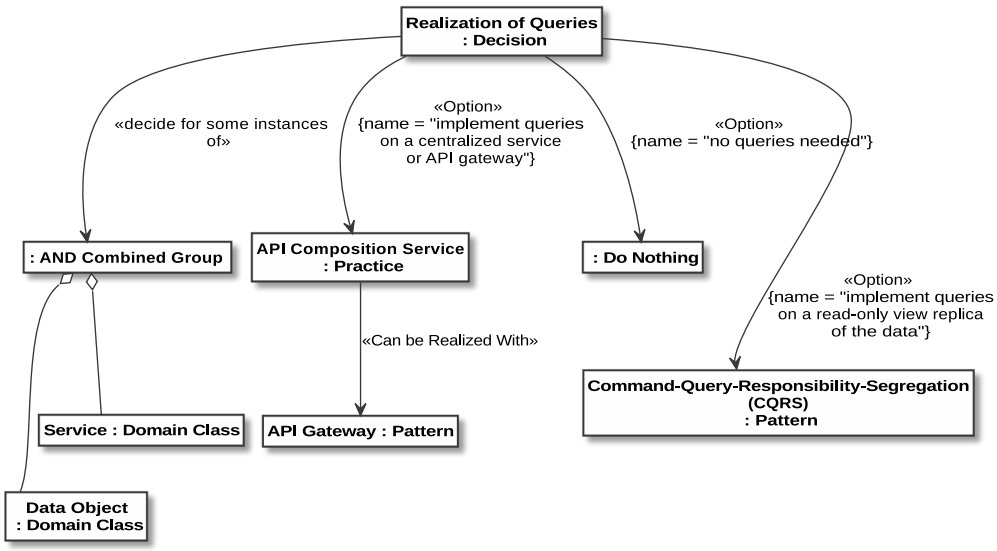

Figure 4.6: Realization of Queries Decision

entails the risk of reduced *availability*, and makes it more difficult to ensure transactional *data consistency*.

4.5 Evaluation

We used our model-based qualitative research method described in Section 4.3 because informal pattern mining, or just reporting the author's own experience in a field (which is the foundation of most of the practitioner sources we encountered), entail the high risk of missing important knowledge elements or relations between them. To evaluate the effect of our method, we measure the improvement yielded by our study compared to the individual sources; specifically *microservices.io* [Ric17], the by far most complete and detailed of our sources. This is an informally collected pattern catalog based on the author's experience and pattern mining. As such, it is a work with similar aims to this study. Of course, our formal model offers the knowledge in a much more systematically structured fashion; whereas in the *microservices.io* texts the knowledge is often scattered throughout the text, requiring careful study of the entire text to find a particular piece of knowledge. For this reason, we believe the formal ADD model to be a useful complement to this type of sources, even if the two contain identical information.

For evaluation of our results, we studied the *microservices.io* texts in detail a second time after completing the initial run of our study, to compare which of the model elements and relations we found are also covered by *microservices.io*. Some parts of this comparison might be unfair in the sense that the *microservices.io* author does not present a decision model and covers the topic in a broad manner, so that some elements or relations may have been excluded on purpose. In addition, there may be some differences in granularity between *microservices.io* and our model, but we tried to maintain consistency with the granularity in the analysis and coding during the

Table 4.2: Comparison of number of found elements and relation types our ADD model and *microservices.io*

Element and Relation Types	**ADD Model**	***microservices.io***	**Improvement**
Domain model elements	4	4	0%
Decisions	9	4	125%
Decision context relations	6	3	100%
Patterns/practices	32	15	113%
Decision to option relations	30	13	131%
Relations between patterns/practices	10	4	150%
Patterns/practices to decision relations	12	4	200%
Categories	1	1	0%
Category to decision relations	5	3	67%
Unique decision drivers	34	17	100%
Decision drivers to patterns/practices relations	182	37	392%
Total number of elements	325	105	210%

GT process. Considering the relatively high similarity of those *microservices.io* parts that overlap with the results of our study, and the general goal of pattern mining of representing the current practice in a field correctly and completely, we nevertheless believe that our assumption that the two studies are comparable is not totally off.

Table 4.2 shows the comparison for all element and relation types in our model. Only 105 of the 325 elements and relations in our model are contained in *microservices.io*: a 210% improvement in completeness has resulted from systematically studying and formally modeling the knowledge in the larger set of knowledge sources summarized in Table 4.1. Apart from the trivial *Categories* element type, most elements and relation types display high improvement, most notably, the *Decision driver to patterns/practices relations*. That is mainly because design options (and consequently their relations) are missing entirely. Apart from *Categories*, only the *Domain model elements* type shows no improvement, because we only considered those domain elements directly connected to our decisions here. In the larger context of our work, we use a large and detailed microservice domain object model, but as there is nothing comparable in the microservice patterns, we only counted the directly related contexts here (else the improvement of our model would be considerably higher).

4.6 Threats to Validity

To increase internal validity we used practitioner reports produced independently of our study. This avoids bias, for example, compared to interviews in which the practitioners would be aware that their answers would be used in a study. This introduces the internal validity threat that some important information might be missing in the reports, which could have been revealed in an interview. We tried to mitigate this threat by looking at many more sources than needed for theoretical saturation, as it is unlikely that all different sources miss the same important information.

The different members of the author team have cross-checked all models independently to minimize researcher bias. The threat to internal validity that the researcher team is biased in some sense remains, however. The same applies to our coding procedure and the formal modeling: other researchers might have coded or modeled differently, leading to different models. As our goal was only to find one model that is able to specify all observed phenomena, and this was achieved, we consider this threat not to be a major issue for our study.

The experience and search-based procedure for finding knowledge sources may have introduced some kind of bias as well. However, this threat is mitigated to a large extent by the chosen research method, which requires just additional sources corresponding to the inclusion and exclusion criteria, not a specific distribution of sources. Note that our procedure is in this regard rather similar to how interview partners are typically found in qualitative research studies in software engineering. The threat remains that our procedures introduced some kind of unconscious exclusion of certain sources; we mitigated this by assembling an author team with many years of experience in the field, and performing very general and broad searches. Due to the many included sources, it is likely our results can be generalized to many kinds of architecture requiring microservice data management. However, the threat to external validity remains that our results are only applicable to similar kinds of microservice architectures. The generalization to novel or unusual microservice architectures might not be possible without modification of our models.

4.7 Conclusion

In this chapter, we have reported on an in-depth qualitative study of existing practices in industry for data management in microservice architectures. The study uses a model-based approach to provide a systematic and consistent, reusable ADD model which can complement the rich literature of detailed descriptions of individual practices by practitioners. It aims to provide an unbiased and more complete treatment of industry practices. To answer RQ1.1 we have found in 32 common patterns and established practices. To answer RQ1.2, we have grouped 5 top-level decisions in the data management category and documented in total 9 *decisions* with 6 *decision context relations*. Further we were able to document 30 *decision to option relations* and 22 (10+12) further relations between patterns and practices and decisions. Finally, to answer RQ1.3, we have found 34 *unique decision drivers* with 182 links to patterns and practices influencing the decisions. The 325 elements in our model represent, according to our rough comparison to *microservices.io*, an 210% improvement in completeness. We can conclude from this that to get a full picture of the possible microservice data management practices, as conveyed in our ADD model, many practical sources need to be studied, in which the knowledge is scattered in substantial amounts of text. Alternatively, substantial personal experiences need to be made to gather the same level of knowledge. Both require a tremendous effort and run the risk that some important decisions, practices, relations, or decision drivers might be missed. Our rough evaluation underlines that the knowledge in microservice data management is complex and scattered, and existing knowledge sources are inconsistent and incomplete, even if they attempt to systematically report best practices (such as *microservices.io*, compared to here). A systematic and unbiased study of many sources, and an integration of those sources via formal modeling, as suggested in this chapter, can help to alleviate such problems and provide a rigorous and unbiased

account of the current practices in a field (like presently on microservice data management practices).

Part III

Assessment of Architecture Conformance

5 Detector-based Component Model Abstraction for Microservice-Based Systems

Software architecture presents challenges in avoiding model drift and erosion in complex systems. These challenges are compounded in microservice-based systems, which frequently use a diverse range of technologies and undergo frequent changes and releases. Existing solutions for reconstructing architecture models struggle to handle continuous evolution, suffer from low accuracy, and struggle in highly polyglot settings. In this chapter we report on a research study aiming to design a highly accurate architecture model abstraction approach for comprehending component architecture models of highly polyglot systems that can cope with continuous evolution.

5.1 Introduction

Microservice-based architectures are a kind of service-oriented architecture that consist of independently deployable, modifiable, and scalable services, each having a single responsibility [New15, LF04]. Microservices typically do not share their data with other services, are deployed in lightweight containers or other virtualized environments, and communicate via message-based remote APIs in a loosely coupled fashion. They feature polyglot programming and polyglot persistence, and are often combined with DevOps practices such as continuous delivery and end-to-end monitoring (see e.g. [Zim17, PZA+17, HS17]). Microservices are one of many service-based architecture decomposition approaches (see e.g. [PJ16, PW09, Ric17, ZGK+07]). Just like other architecture decomposition approaches, they do not address the classical software architecture problems of *architecture drift and erosion* [PW92] well. That is, during system evolution, the architecture models increasingly diverge from the actual software as changes are made in the source code which either violate the architecture model's original specifications, or are not reflected in it, for example through the introduction of new features [ZZGL08].

To address this problem, architecture reconstruction approaches have been proposed to automatically or semi-automatically produce architecture models from the source code [DP09, MNS95, MMW02]. Unfortunately, these approaches usually involve a substantial effort to either manually maintain the reconstructed architecture model, or repeat the reconstruction after the system has evolved (see [HZ14]), meaning that they are not suited for supporting continuous evolution of systems. In addition, automated approaches have low accuracy (see [GIM13]), and much additional, manual effort is needed for correcting and augmenting their results. Finally, most reconstruction approaches focus on a very limited number of programming languages and technologies (see [DP09]), meaning they are hard to use with modern systems, such as microservice-based

systems, which use typically polyglot programming, persistence and technologies, often in their latest iterations.

For these reasons, the prospects for ever developing a one-size-fits-all, generic reconstruction method that can cope well with evolving microservice systems (and similar polyglot systems) look bleak. Fortunately, there is hope in the fact that developers usually know a lot about their projects and thus a generic, fully automated reconstruction may not be necessary. In this chapter, we report on a design science research study [Wie14] in which we aimed to design a new approach to enable the accurate creation and continuous evolution of component architecture models in microservice-based and similar polyglot settings with little extra effort. We set out to answer the following *research questions*:

- **RQ2.1** How to design a 100% accurate architecture model abstraction approach for comprehending component architectures of systems that are highly polyglot?
- **RQ2.2** How to support continuous comprehension of such systems in the context of such an architecture model abstraction approach?
- **RQ2.3** How high is the required time (effort) for creating and maintaining architecture model abstractions in such an approach?

Our study was performed by first defining the design science study in terms of design context, artifacts studied, stakeholders, and their requirements. We then selected a case study for research validation and investigated the case by performing a manual reconstruction of it, used later as a ground truth. We then analyzed the related studies that fulfill our requirements best.

In particular, our work is an extension of the approach taken by [HZ14], which is presented in more detail in Section 5.3. Based on our experience with this work, we designed an opportunistic detector-based approach which is capable of fulfilling all our requirements regarding support for polyglot, continuously evolving systems. Our evaluation of this first approach showed that it could be further refined by making the detectors *reusable*, which we proceeded to accomplish. The result were two approaches which are the key contribution of this work:

Detectors *are software components that continuously parse relevant parts of the source code and create model abstractions from the code.*

Reusable Detectors *are detectors which can be reused across different model abstraction tasks and projects.*

We realized both approaches fully (design, prototype development, validation in the case study), and quantitatively and qualitatively compared the results of the two approaches.

This chapter is organized as follows. Section 5.2 examines related work and explains our study's contributions in the context of the state of the art. Next, in Section 5.3 we explain the background of this study and Section 5.4 explains our research study design and the two detector-based approaches in detail. In Section 5.5 we explain the case study implementation, and in Section 5.6 we report on its evaluation. In Section 5.7 we provide a brief overview of an implementation of the same approach in a different domain, as addition proof of concept. We conclude with a discussion of the treats to validity of our approach in Section 5.8, and our general summary in Section 5.9.

5.2 Related Work

Related Works on Microservice Architectures Microservices [New15, LF04, Ric17] are, among many other things, a way to decompose an architecture based on services [Zim17]. This is an area which has been studied intensively in recent years (see e.g. [PJ16, PW09, ZGK+07]). According to mapping studies [AAE16, FML17], the focus of microservice research is – in contrast to our study – frequently on specific system architectures or applications, often in relation to questions of deployment, monitoring, performance, APIs, scalability, and container technologies. The problems of complexity and service composition – relevant to our study – are addressed often, but the majority of these studies focuses on a variety of qualities (with many focusing on runtime aspects) [FML17]. [GCF+17] provide one of the few existing microservice-specific architecture reconstruction approaches. It statically analyses Docker and Docker Compose files for names and ports, and then the Docker containers and network bridges dynamically, to reconstruct the deployed microservices from the system's communication logs. While having quite a different goal than our study, this approach confirms our thesis that microservices require different approaches to architecture reconstruction than those adopted by the existing literature on the topic. As Granchelli et al. only use information from Docker files and related data, the reconstruction achieved is much more limited than the two approaches reported in our study, but in contrast to our approach it considers information on dynamic behavior as well.

[AAE18a] present an approach that is intended as a groundwork for architecture reconstruction of microservices. From the analysis of 8 open source projects, the approach derived a meta-model and possible mapping rules for microservices. This approach misses the detection component, which is the major focus of our approach, but additionally focuses on a broader set of concerns than just those that can be modeled in component models. [VLS14] report on a study of a microservice-based reference architecture as a starting point for enterprise measurement infrastructures. This can be seen as an alternative to a reconstruction effort, but it requires manual maintenance of the architecture in relation to the reference architecture – which could, e.g., be provided by one of the approaches reported in our paper. [RSZ19] suggest to address the polyglot nature of microservices using an aspect-oriented modeling approach. Again, this approach requires manual effort. It could be used as a modeling extension of our approach, where our approach can deliver the information needed for creating and maintaining the model.

Related Works on Architecture Reconstruction and Abstraction Architecture reconstruction focuses on automatically or semi-automatically producing architecture abstractions from the source code [DP09, MMW02, MNS95, vDB11]. Many approaches focus on identifying components or similar abstractions through automatic clustering [vDB11, CDMS10]. A variety of approaches establish different kinds of abstractions between source code and the architecture level. Some use graph-based techniques [Sar03], while others utilize model-driven techniques [SROV06, MNS01, KMNL06], or logic-oriented programming [MMW02]. Other approaches [GL14] analyze external dependencies to discover architectures and analyze a system's quality attributes. ExplorViz observes the runtime behavior of instrumented software systems and reconstructs their architecture on software landscape and software application level [FKH17].

Unfortunately, these approaches have some major issues in practice: (1) architecture recon-

struction approaches focus on identifying abstractions from code, without considering continuous software evolution. That is, once a reconstruction effort is finished and a few subsequent evolution cycles of the software system have occurred, the reconstructed architecture is once again outdated and a new reconstruction effort would be needed. This would not be a big problem, if the reconstruction approach were low-effort, fast, and largely automated. However, (2) automated reconstruction approaches generally have rather low accuracy, precision, and recall ability. For example, a comparative study of nine approaches reports average accuracy of 31% to 58% [GIM13]. Given the tremendous effort needed to find and correct incorrectly-mapped source code elements in large-scale systems, in practice anything else than close to 100% accuracy is hard to use. In other words, today a substantial manual effort is required to reach close to 100% accuracy when starting off with the results of an automatic reconstruction. Finally, (3) most reconstruction approaches focus on a very limited number of programming languages and technologies (e.g., only considering Java code and even there ignoring special cases such as reflection, libraries that create dynamic dependencies, dependencies injected by an external technology, and so on). Systems such as today's microservice systems use polyglot programming, persistence and technologies, often in their latest iteration; that is, different programming languages are used, and in each of them various libraries, sometimes offering multiple APIs, are used to perform tasks such as client invocations, server programming, publish/subscribe interactions, database access, dependency injection, and so on. This is combined with multiple technologies for persistence, dependency management, CI/CD, containerization, end-to-end monitoring, call tracing, and so on, each coming with their specific configuration or other domain-specific languages. New such libraries and technologies emerge constantly, which are quickly adopted by microservice projects, further complicating the issue.

For this [HZ14] proposed an approach for creating an architecture component view from the source code using a domain-specific language (DSL) for architecture abstraction. In this approach, the architect would specify known facts, such as the names of the major components, filter patterns for the relations of packages or classes to components, and so on. The filter patterns are designed to require little or no change if the source code changes. By studying various cases, it has been shown [HZ14, HNZ17] that this approach requires relatively little effort (compared to program size) and can cope well with the evolution of systems. As this approach seems in a number of ways more promising for practical support of continuous architecture abstraction than the existing reconstruction approaches, we decided to use it as groundwork for our study. Like the other mentioned approaches, however, it too falls short in addressing the polyglot nature of microservice-style systems.

5.3 Background

Our study of related works identified the approach by [HZ14] as close to our research requirements. From an architectural point of view, this approach uses program code in Java, and the abstraction DSL, as inputs, and creates models as output. In a first step, Java source code elements are mapped to an abstract syntax tree, from which then a detailed UML model of the relevant parts of the code is created. Next, this model is interpreted and transformed. In a background analysis we investigated to what extent it was possible to extend this Java parsing-based approach to support

multiple technologies and languages. We designed a similar solution based on ANTLR[1], since that is one of the few polyglot parser frameworks that supports most of the grammars for the languages used in our case study. Unfortunately, parsing with existing grammars for ANTLR frequently failed for our case study (see Section 5.4.1) and other test examples, as many ANTLR parsers do not support all the latest features of all languages used in our case study.

We decided not to pursue this approach further, since it would have required sustained effort to first correct and then maintain a wide variety of grammars, just to be able to parse all language features that we might encounter. We concluded that the approach would require us to maintain a polyglot parser framework or an adapter framework to polyglot parsers.

5.4 Case Study Design

This study employs the design science research method, which supports studying the design of artifacts in a specific context [Wie14]. A design science research study is performed in a number of design and engineering cycles. [Wie14] defines 4 possible steps in such a research cycle: *problem investigation*, *treatment design*, *treatment validation*, and *design implementation*. Evaluation of a cycle might lead to a next cycle for improving the design. The last step of the research cycle, *design implementation*, concerns the technology transfer into the real-world context, and is optional. We have not performed it in our study. For *treatment validation*, several validation methods can be applied, including various empirical methods. In our study, we have opted for an empirical evaluation based on a case study. If empirical methods are applied, Wieringa proposes a nested empirical cycle for performing the empirical study.

5.4.1 Study Definition

Figure 5.1 summarizes the main steps in our research study, which started with a *problem investigation*, followed by a definition of requirements, and a background analysis of the approach by [HZ14] (described in Section 5.3) that our study has revealed as close to our research requirements. In parallel we performed a manual reconstruction of a case study as a ground truth (described in Section 5.4.1). Based on the insights of those research steps, we designed and validated first the opportunistic detector-based approach and then the reusable detector-based approach. Finally, we quantitatively and qualitatively compared the results of the two approaches.

Problem Investigation and Treatment Design

In an initial *problem investigation* and requirements definition phase, we have investigated the problem from a stakeholder and stakeholder goals perspective, followed by a definition of the requirements. In parallel we have defined and studied the case study; this has influenced the problem investigation and requirements definition, and vice versa.

Context The specific *context* of our study is *comprehending microservice architectures*; this context can be generalized to *comprehending the component architectures of polyglot and evolving*

[1] https://www.antlr.org/

software systems.

Artifact As we have argued above, to design a fully automated reconstruction technique that works well in this context is likely infeasible. Using the examples provided below, we will illustrate why this is the case in more detail. Hence, we decided to study instead as an *artifact* a *semi-automatic architecture abstraction method* inspired by the work of [HZ14].

Stakeholders The *stakeholders* of our study are *microservice developers and architects* whose systems are complex enough to make comprehension difficult; in a broader context, any *developers and architects* who are in such a situation are the relevant stakeholders.

Stakeholder Goals and Requirements We first investigated high-level stakeholder goals and then derived the following concrete requirements for our design:

- **R1** The approach should support stakeholders in getting an accurate understanding of the component architecture of a system, in the form of a *complete component model* at a sufficient level of detail, i.e., a model that contains all the possible components and component types, connectors and connector types as well as the related technologies.

- **R2** The approach should lead to architecture component models that are – in the absence of human error – *100% correct*. Our approach enables 100% accuracy since it involves a full and detailed manual reconstruction of the component architecture, which is the ground truth for the development of our detectors. The process ensures that the detector developers will be familiarized with the architecture, if that was not the case previously. The detectors are developed explicitly to cover *at least* the ground truth established by the manually reconstructed architecture, and are thus guaranteed to cover all architecturally relevant elements (per R1). Please note that our approach *could potentially use heuristics as detectors.* We have deliberately not chosen this option in this article's case study. If it was chosen that detection would be less than 100% correct as a downside, but the efforts (R6/R7) could substantially be reduced this way. Investigating this option further is beyond the scope of this article.

- **R3** The approach should be applicable in a microservice setting. That is, it should be possible with a reasonable time (effort) (see Requirements R6 and R7 below) to cope with *polyglot programming* and projects which frequently adopt the *latest technologies*.

- **R4** The approach should support *continuous comprehension*. That is, the time (effort) needed to recreate the architecture model after a change of the system should be minimal, i.e., usually close to zero; and in exceptional cases, a fraction of the time (effort) needed for creating the initial architecture model abstraction.

- **R5** The approach should support *traceability between architecture model abstractions and the source code*.

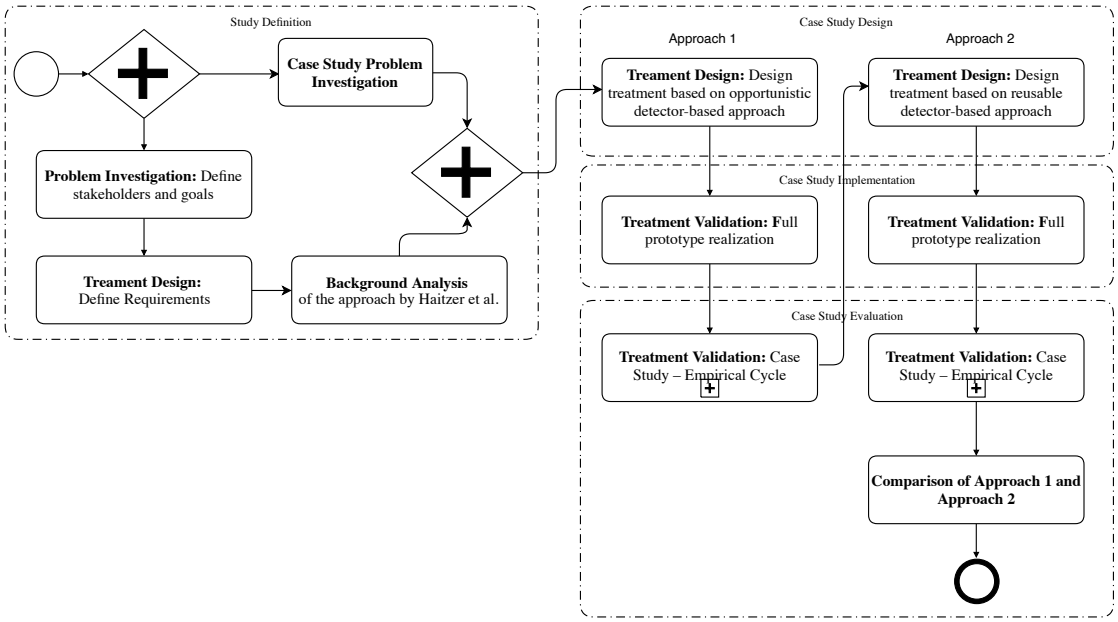

Figure 5.1: Overview of the research study execution steps

- **R6** Compared to the overall time (effort) needed to engineer the system, stakeholders should need to invest only a minimal amount of time (effort) for the manual part of the architecture model abstraction. We estimate that less than 1% of the development time (effort) is acceptable in practice.

- **R7** Compared to the overall time (effort) needed to manually reconstruct an architecture, stakeholders should need to invest only a small amount of time (effort) for the manual part of the architecture model abstraction. We estimate less than 10% of the reconstruction time (effort) is acceptable in practice.

Case Study: Problem Investigation

In order to provide a suitable case study, based on our requirements, we require a highly polyglot microservice-based system that applies a substantial number of different technologies. The case study should have a reasonable size, but not be too large for us to be able to completely implement an architecture abstraction in the scope of a research study, maybe multiple times in each of the research cycles. It should have an industrial background (i.e., be implemented by industry experts, not a toy example by researchers). One option would have been performing an observational case study in industry. But as our study design demands that the design science artifact should substantially evolve within each research cycle, this would have required many implementation iterations performed by industry experts to adapt the case study according to the research progress; this would have made rapid improvements of the method based on intermediate case study results

impossible. For this reason, we decided to perform a so-called *mechanism experiment* [Wie14], i.e. implement the architecture abstraction prototypically for the case ourselves.

We selected an open-source system[2] that was built as a demonstrator for the Instana monitoring technology. We report here on the master branch from 2019-10-23. Overall it consists of 140 files with a total of 5311 lines of code. It was built by industry experts from the company Instana in the timeframe Jan, 2018 — Oct, 2019; hence we believe it to be a good representative example for the current industry practices in microservice-based architectures. The project is highly polyglot: it consists of services written in JavaScript/NodeJS, Java/Spark, Python/Flask, Go, PHP/Apache, RabbitMQ messaging, and Python/Go AMQP messaging. These services use Redis, MongoDB, and MySQL as database technologies, accessed with various APIs for RESTful HTTP communication. AngularJS is used for the web frontend. Nginx is used as an API gateway and web reverse proxy. Docker, Docker Compose, Docker Swarm, and Kubernetes are used for lightweight virtualization and autoscaling. DC/OS and OpenShift are supported. End-to-end monitoring via Instana is supported, and some services have Prometheus metrics endpoints. A load generator is built with Python/Locust. Paypal is used as an external service.

For problem investigation, we performed a full manual reconstruction of the component architecture of the system as a ground truth for the case study. Figure 5.2 shows the result, a detailed component model specifying the component types (e.g., *Services*, *Facades*, and *Databases*), and connector types (e.g., *RESTful HTTP*, *Synchronous/Asynchronous*, *database connectors* etc.). This figure is based on the auto-generated figure created by the prototype implementing our proposed approach, described below. The system consists of 18 components and 29 connectors. More specifically, a *Client*, a *Web UI*, an *API Gateway* as entry point of the system, seven *Services*, three *Databases*, a *Message Broker*, an *External Component (Service)*, two *Monitoring Components*, and a *Tracing Component*. We kept precise time records of the manual reconstruction effort. It took us *2468 minutes (approx. 5 person-days) to perform the reconstruction*.

To study R3, for a very rough comparison of time (effort), we have used the numbers estimated by COCOMO [BCH+95] that would be needed for constructing the case study in an industry setting. We used the online calculator provided by COCOMO [3]. To be on the safe side, we used very conservative parameters for the COCOMO estimations (assuming only nominal values for parameters such as *experiences*, *capabilities*, *developed for reusability*, and so on). In total, the estimated effort for the case study system was *23.7 person-months*. This estimation is inline with the estimation in Code Complete which states: “The industry-average productivity for a software product is about 10 to 50 of lines of delivered code per person per day (including all noncoding overhead).” [McC04]: Assuming 1720 working hours a year, the 23.7 person-months would yield 1698.5 person-hours or 212.31 8-hour person days. This means 31.43 lines of delivered code would be needed for the COCOMOII estimate, which is very close to the average of the Code Complete estimate of 30.

[2] https://github.com/instana/robot-shop
[3] https://csse.usc.edu/tools/COCOMOII.php

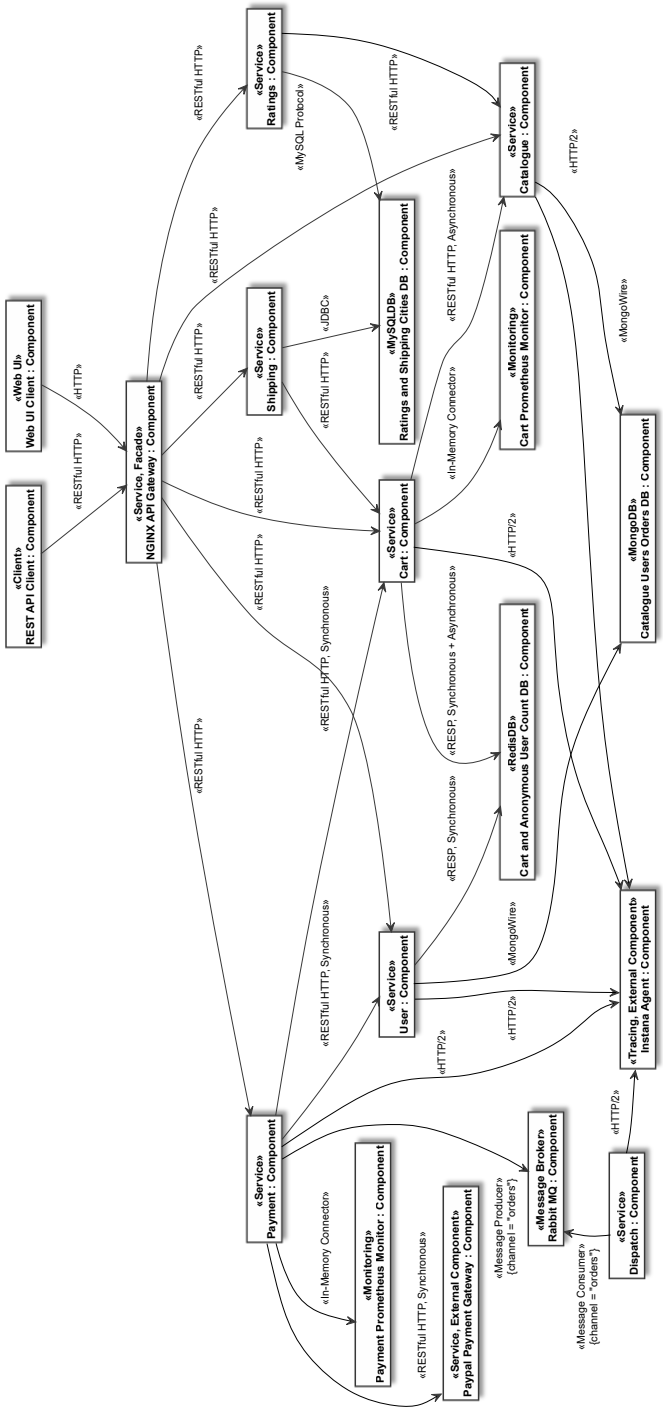

Figure 5.2: Case Study: Overview of the reconstructed component architecture Ground Truth as a UML2 model

5.4.2 Detector-based Architecture Abstraction Approaches

In this section we present and describe in detail the design for *Approach 1* and *Approach 2*. The code and models used in and produced as part of this study have been made available online for reproducibility [4]. Both approaches are based on detectors and aim to address architecture reconstruction challenges introduced by continuous evolution of microservice-based systems and their polyglot nature. Approach 1 is more case-specific and requires custom detectors, while Approach 2 provides detectors that can be reused in multiple cases.

Note that both approaches presuppose that a system expert (architect) has identified the high-level, component-and-connector architecture of the system — with which he should be familiar either way — and modelled it in an execution script that iterates the detectors over each system element. This involves a relatively small per-release effort (removal and addition of services and links between releases, cf. R4, R6, R7 and Section 5.6.2), but can also be obviated altogether by adapting the detector approach to this domain, as shown in Section 5.7.

Approach 1: Opportunistic Detector-based Architecture Model Abstraction

The design used in the study was based on small detectors, one for each feature relevant for detecting one or more architecture abstractions. For example, if code is written in JavaScript/NodeJS, importing the `request` library means that a RESTful HTTP call could possibly be used in the file(s) using this specific technology; if a `request(...);` is present in addition, an HTTP call is actually made in the file. If a manual inspection confirms that a RESTful HTTP call is actually made, we have precise evidence for the presence of a RESTful HTTP call and can establish traceability links to all occurrences of such invocations. Based on such simple detectors, we can correctly detect most evolution scenarios: if changes to other parts of the file are made, it is not possible that the detection of the RESTful HTTP call will fail. If another RESTful HTTP call is added, it will be detected, too. If all RESTful HTTP calls are removed, the detection will fail, as it should, and manual action is required. Only if a RESTful HTTP call with a different technology is made, would a remodeling of the detector be required.

This new approach would not work better than the one from Section 5.3 in terms of parsing, if we followed the same full-fledged parser-based approach. However, some parser frameworks support scanning for the occurrence of parse rules rather than requiring parsing the whole file. One such parser framework is *pyparsing* [5], which we used to only parse the relevant parts of the code. This solved the parsing issues described in Section 5.3. To illustrate the approach, let's consider a simple detection from our case study. Assuming that we have previously detected two components, the *Shipping* and *Cart* services (cf. Figure 5.2), we now want to determine the presence, and the technology, of any connector between the two components. For detecting these two components we used the *JSExpressService* and *JavaSparkService* detectors that return an evidence specifying the corresponding technology types. The detector process will call the detector instances listed in the *DetectInFile* evidence to determine the connections between the two components (see Figure 5.3). In this case, if the *DetectInFile* evidence is successful in detecting (1) a *Main.java* file in the specified directories, and (2) successfully executes the

[4] https://doi.org/10.5281/zenodo.5235931
[5] https://github.com/pyparsing/pyparsing

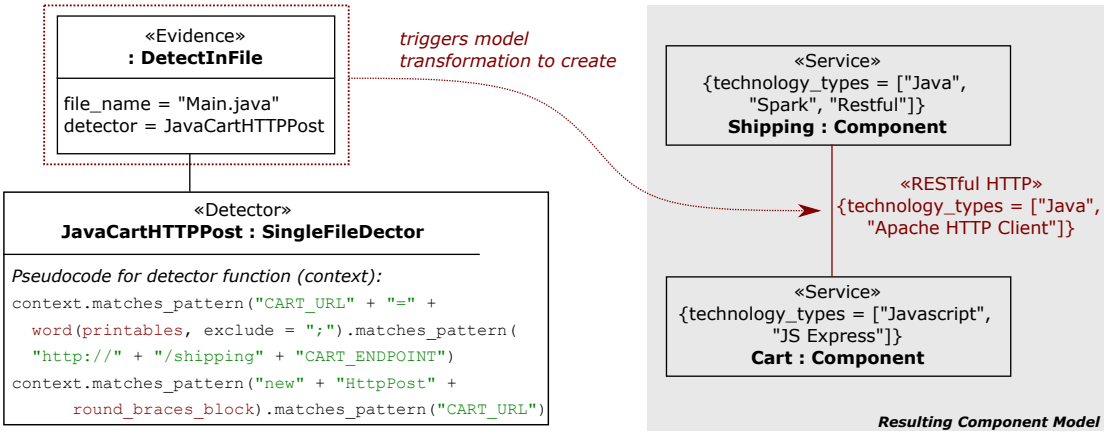

Figure 5.3: Opportunistic detector example: Detecting a Restful HTTP connector

JavaCartHTTPPost detector, it adds a connector of the *restfulHTTP* type between the *Shipping* and *Cart* components to the model. Here, the *DetectInFile()* represents the reusable code of the detector process, and *JavaCartHTTPPost* is a specific detector required for this particular occurrence (i.e., detecting the presence of a Java HTTP call from *Shipping* to *Cart*). As the specific code will be different for other occurrences, we call this the *opportunistic approach*. Please note that usually, the specific code required for a detection is rather small, e.g. in this case two lines of pseudo code are enough.

As we will discuss in more detail below, following this approach, we were able to design a solution that fulfills all the requirements of our study. However, we observed that, as can be seen in the *JavaCartHTTPPost* example, a lot of code is very specific for the particular case at hand, and that many aspects of the detection could be automated to a higher degree, with a higher code reuse. For instance, in the simple example given here, most likely many Java posts, or even other HTTP requests using the same API, could be detected with a more generic detector. If this detector is selected, it is known that it produces *Restful HTTP* connector links; thus a reusable solution could provide this knowledge as default value. Also, it might not be necessary to specify the exact file in which the request occurs, as this could be "guessed" from the directories of the detected components. As a consequence, while the approach described here works well, a reusable detector approach with less specification effort per case and more automation potential might be possible. Consequently, we aimed to design such an approach next.

That is, based on the current state of the art, this approach cannot satisfy Requirement R3 (concerning a "small amount of time (effort) for the manual part of the architecture abstraction") and makes Requirement R4 hard to achieve ("continuous comprehension"). Based on this experience, we designed an opportunistic detector-based approach (*Approach 1*) to cover all challenges that the approach by Haitzer and Zdun cannot address. Based on this, we realized that it is also possible to design a similar, but reusable, detector-based approach (*Approach 2*).

Approach 2: Reusable Detector-based Architecture Model Abstraction

In the reusable detector-based approach, we aimed to reduce the necessary specification in the abstraction model and completely get rid of any case-specific detector code. Instead, all detection should be handled in reusable detectors. We managed to bring almost all specifications of architecture abstractions down to a single line of code per abstraction. To illustrate this approach, let us again consider the previous example. We developed a generic, and hence reusable, detector for the Java Apache HTTP technology, which we provide to the method, along with the IDs of the two components, to create a link between them. The directory in which to search for the link is taken from the component, where it is provided as a top-level directory only (no specific directory or file names are provided anymore).

Given this dramatic reduction in the code size that needs to be written by users, it might seem at first that full automation might be possible. However, this is not correct: Note that the detector specification is meant as an *assertion by a human* that a component or connector link of a certain type was found. With this little extra information – which is much easier to obtain than performing a full manual architecture model reconstruction – we can avoid the issues that make automatic detection hard or even impossible. The problematic part that requires human input in this case is the *Cart* URL and endpoint, which are two specific variable names used in the Java implementation (see pseudo code in Figure 5.3). We could potentially guess them to a certain extent, but developers could find many ways to implement or change them in the future, such as hard-coding the URL directly, obtaining them through a call to an arbitrarily named method, reading them from a file, and so on. By requiring the human identification of the occurrence once, we greatly reduce the possibility of any false positives or negatives.

In our reusable detector, we use some heuristics, and the human user who specifies the case-specific detection must be aware of the heuristics and their limitations. For the example detection between the *Shipping* and *Cart* services we used in Section 5.4.2, the *JavaApacheHTTPLink* reusable detector. It is able to find matches for *get*, *put*, *post*, and *delete* requests, as illustrated in Figure 5.4. For all of these, it checks in the relevant Java files (`file_endings = ["Java"]`) if a respective *new* statement is found. If so, we have previously auto-detected possible aliases for the component names in various places; e.g., in the specific case, these were detected in Docker *env* statements. If one of the matches for the HTTP method links to one of the target component's aliases, we have found a match for calling from source to target using an HTTP method. Otherwise we use the detector method *get_var_assignment_matches_containing_url_alias* to find all Java variable assignments that match one of the target aliases. If one of those is used in a match, it also constitutes a match for calling from source to target component. Both are seen as evidences for a link between the source and target components. This would then trigger the model transformation for creating the Restful HTTP connector in the component model.

With this design, the detector process can tolerate many changes in system evolution (even more than in the design from Approach 1). For example, calling the URL directly instead of using the *CART_URL* variable, moving to a different variable name, moving the call to another method or file, and so on, are examples of possible non-breaking evolution scenarios. The design also fulfills all requirements set for the design study. Thus, we next studied how the two approaches compare with regard to our requirements in the context of our case study to answer the research questions.

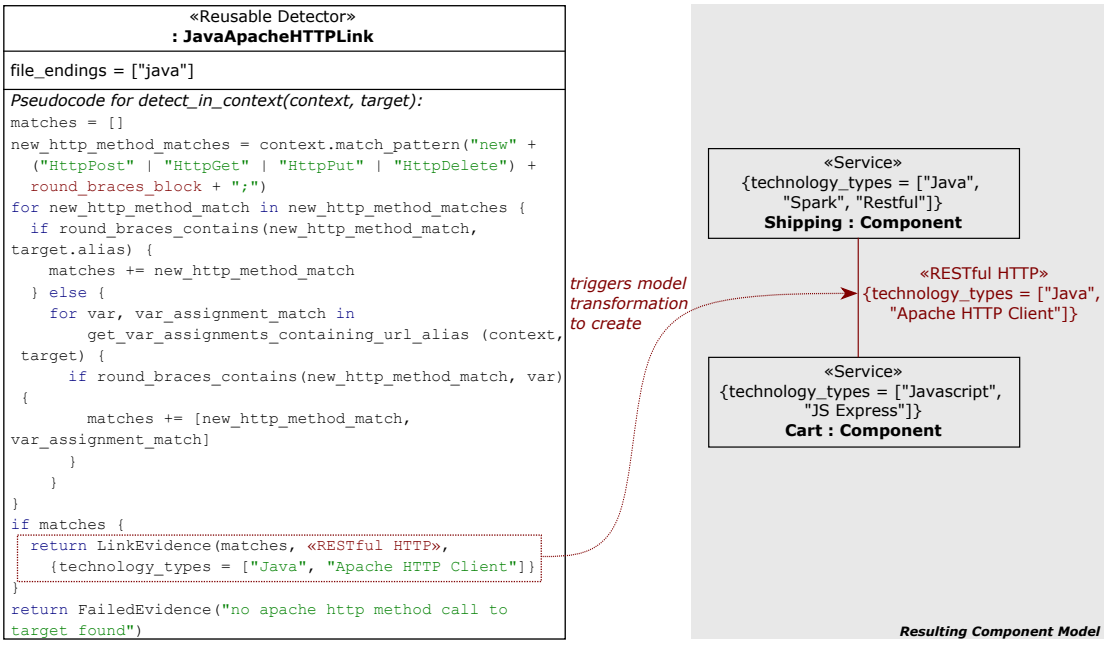

Figure 5.4: Reusable detector example: Detecting a Restful HTTP connector

5.5 Case Study Implementation

5.5.1 Architecture UML Profile

As the generation target of our model transformations we introduced a UML profile. In it, components are extended by the stereotype *component types* (see Figure 5.5), and connectors by the stereotype *connector types* (see Figure 5.6), for introducing the microservice-specific modeling aspects. That is, to be able to apply our two approaches, we first performed an iterative study of a variety of microservice-related knowledge sources, and we refined a meta-model which contains all the required elements to allow an adequate reconstruction of architecture model abstractions in the microservice domain. The resulting stereotypes range from general notions such as the *Service* component type, to very technology-specific classes such as the *RESTful HTTP* connectors. As component types we support, for example, *Service*, *Pub/Sub*, *Message Broker*, *Event Sourcing*, *Steam Processing*, *Client*, *External Component*, *Web UI*, *Monitoring*, *Tracing*, *Logging*, *Saga Orchestrator*, and various kinds of *Databases*. As connector types we support, for example, various *Service Connectors* such as *RESTful HTTP*, *SOAP*, or *GRPC* connectors, various *Web Connectors* such as *HTTP*, *HTTPS*, or *HTTP/2*, various kinds of *Synchronous* and *Asynchronous* connectors, various *Indirect* connections, *In-Memory* connectors, various *Database* connectors, various *Event-based Connectors* such as *Publishers* and *Subscribers*, and various *Messaging Connectors* such as *Message Producers* and *Message Consumers*.

Figure 5.5: Architecture UML Profile: Component Stereotypes

Figure 5.6: Architecture UML Profile: Connector Stereotypes

5.5.2 Detector Framework

One of the contributions of this work is a model for the design of a semi-automatic detector framework for creating architecture abstractions along with traceability links to the source code. The latest design for this part of our study in Approach 2 differs only in details from the one from Approach 1. Thus we report only the latest design here, as it is a few refactoring cycles ahead of our earlier design. As shown in Figure 5.7, a number of detections are abstracted in a *Project*. Each detection traverses the models to detect features of interest. That is, to the project we add architecture abstraction in specifications such as those in Figure 5.4. In those specifications, the *Detectors* that the project should use to detect the abstractions are specified. For example, the 2 components (*Shipping* and *Cart*) in Figure 5.4 use the detectors *JavaSparkService*, *JSExpressService*, and the connector between them uses the detector *JavaApacheHTTPLink*. Two specific subclasses of *Detector* are shown in Figure 5.7: one for detecting at least one matching file and one for detecting matches across multiple files. The former is used as superclass for the majority of our current detectors; the latter is used occasionally. As illustrated in Figure 5.4, detectors use *DetectorContexts*, which implement the scanning and matching methods and contain the text to be parsed. If detectors find matches, a *Match* is used to store the matching text, its position in the parsed text, the file, and the directory; this way, traceability to the source code is established. When all required matches are found for an architecture abstraction, the detector creates an evidence for it (like the *LinkEvidence* in used in Figure 5.4). As shown in Figure 5.7, *Evidence* has three main subclasses, called *FailedEvidence*, *NamedEvidence*, and *LinkEvidence*: *NamedEvidence* has an additional subclass *ComponentEvidence* in which possible link types can be collected on the component if its matches can be considered as enough items for a link (e.g., a Web server, offering the component already, implies possible links to the Web clients). It has also a subclass *ServiceEvidence* to specify the corresponding component type.

The project stores detected components to use them for later detection; e.g., the directory guessing in link detectors explained above works this way. Our design only works for components and connector links so far; for supporting other abstractions, more evidence types and additional functionality on the project would be needed. The project stores all failed evidences, because we do not stop the detection if one failure occurs, but rather let all detectors run through and provide the user with all failures that occurred.

As proof of concept, we realized a process, using Python scripts, that takes the polyglot source code and the detector specification as inputs. It creates an architectural abstraction containing a UML-style component model together with *Evidences* and *Matches*. Models are expressed using the Python modeling library CodeableModels described in Chapter 3.7, a Python implementation for precisely specifying meta-models, models, and model instances in code with a lightweight interface. The process contains a visualization generator for generating PlantUML diagrams such as the one in Figure 5.2.

While the process flow could be changed substantially compared to how we constructed our prototype, the general process flow architecture illustrates how the building blocks interact in order to enact the design explained above (see Figure 5.7). The process aims to generate models for the modeling library CodeableModels. As illustrated in Figure 5.8, the performer of the process is envisaged as a developer or architect. The architect specifies the architecture abstraction specification for a software system, while the developer implements it in source

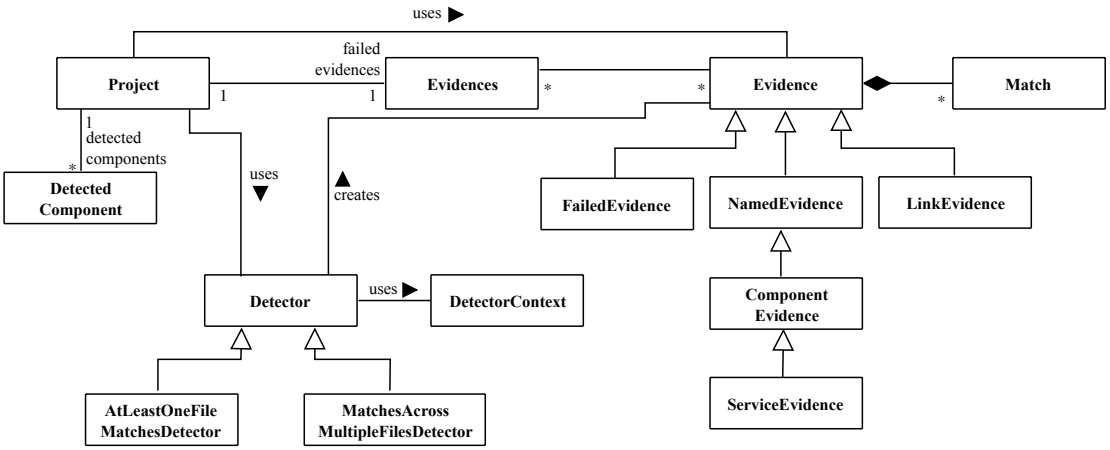

Figure 5.7: Resulting Design: Domain Model of the Detectors

code. Both roles can also be involved in the development of detectors for the two approaches outlined above. The *Detectors Development Process* produces as output either the *Architecture Model Abstraction* based on the *Opportunistic Detector* or the *Reusable Detector* approach, which in turn are used as inputs in the *Detector-based Architecture Abstraction* phase of the *Detectors Implementation Process*. The architecture specification and the system's source code are also inputs of the *Detector-based Architecture Abstraction* phase. The detector performs the architecture abstraction, and when successfully completed, uses a code generator to generate the corresponding *System Component Model* with CodeableModels. The code generator utilizes the *Architecture UML Profile* to instantiate the *System Component Model* from it. Finally, CodeableModels contains a *Visualization Generator*, which uses the *System Component Model* as input, for generating PlantUML diagrams such as the one in Figure 5.2.

5.6 Case Study Evaluation

5.6.1 Effort and Size

In this subsection, we want to give a rough estimation of the effort required to design and implement prototypes for (1) the generic detectors approaches for our two approaches and (2) the case-specific code in both approaches as well as a size comparison in terms of lines of code (LoC). We discuss in Section 5.8 why such a comparison can only provide a rough estimate. More research is needed to generate solid numbers, e.g. for precise effort prediction. However, for the purpose of this study, roughly correct numbers are good enough, as we are interested in understanding the effort and size relations in orders of magnitude. The efforts in minutes reported in Table 5.1 are based on manual time recordings made throughout our project. Lines of code are automatically counted using the VS Code plug-in VC Code Counter, which supports the counting of only the Python code.

As can be seen, the generic code base needed for Approach 2 is substantially larger (64.41%)

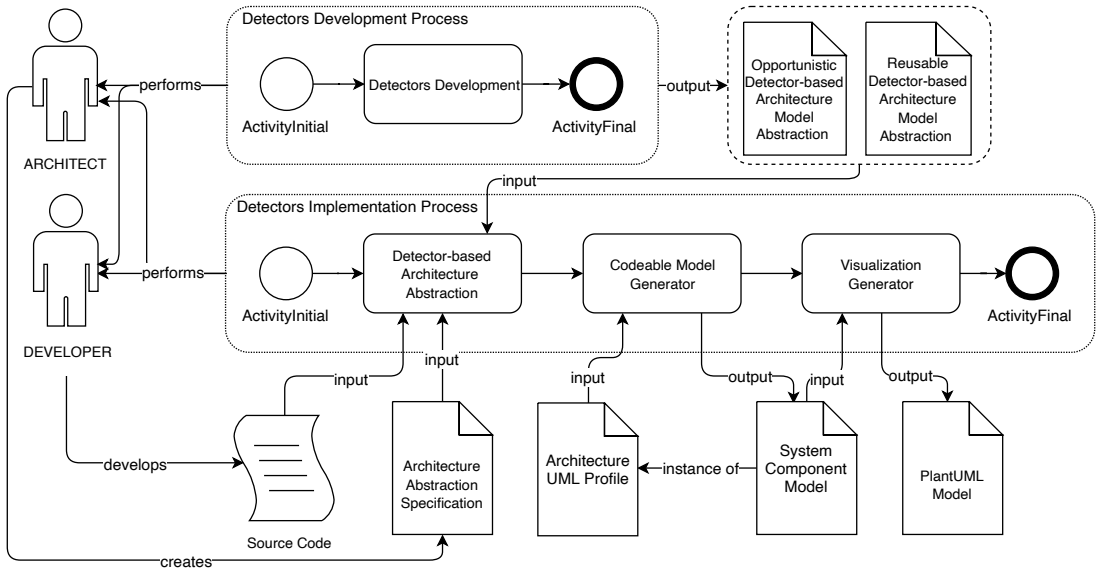

Figure 5.8: Process Flow Architecture of the Prototype

than for Approach 1 and we needed moderately more time (21.62%) for creating it. The effort increase is less than the code size increase, as the generic code base contains a common code base of about 40% of the code, which was created in 1900 minutes. In addition, a learning effect from the experience in the design and implementation of Approach 1 probably also played a role in the reduction. We believe that this learning effect is small, as the non-common code parts turned out to be significantly different. In addition, Approach 2 contains reusable detectors as part of the generic code, which Approach 1 does not; it contains the case-specific detectors instead. Note that for our case study, the number of reusable detectors in Approach 2 is very high because we have studied a highly polyglot case, and started out with zero detectors; for a less polyglot case (as the cases discussed in Section 5.7), or if already extant detectors can be reused (e.g., from an open-source detector repository based on our approach), would dramatically reduce the number of new detectors and the concomitant effort expended. As a consequence of these numbers, the generic code effort is much higher (146.78%) for Approach 2, and the LoC needed for Approach 2 are significantly more (338.89%).

The situation for the case-specific parts is reversed: The case-specific model for Approach 2 is much smaller (-43.69%) than for Approach 1, and requires much less effort (-45.13%). In addition, Approach 1 requires 107 LoC for case-specific detectors constructed in 870 minutes (which are totally absent in Approach 2). Consequently, the case-specific code for Approach 2 is in total significantly smaller (-62.94%) than for Approach 1 and required substantially less effort (-79.13%). That means that realizing a reusable solution can pay off in the long run, when the approach would be applied on many projects. If the approach is needed only once (and for a small-scale case study as performed here), the reusable approach in Approach 2 does not pay off,

as the total effort and LoC comparisons show.

	Approach 1 Effort in Minutes	*Approach 2 Effort in Minutes*	*Diff.*	*Approach 1 Lines of Code*	*Approach 2 Lines of Code*	*Diff.*
Generic Code Base	5329	6481	21.62%	576	947	64.41%
Reusable Detectors	–	6670	N/A	–	1581	N/A
Total Generic Code	5329	13151	146.78%	576	2528	338.89%
Case-specific Detectors	870	–	N/A	107	–	N/A
Case-specific Model	534	293	-45.13%	206	116	-43.69%
Total Case-specific Code	1404	293	-79.13%	313	116	-62.94%
Total	6733	13444	99.67%	889	2644	197.41%

Table 5.1: Effort and Size Comparison

5.6.2 Requirements Fulfillment

For Requirements **R1** and **R2** we can assess that both approaches are able to exactly reproduce the component model from the ground truth. The same result would likely be possible with the approach by Haitzer and Zdun (discussed in Section 5.3), but this has not been fully implemented in our study. Our approach can lead to highly accurate architecture model abstraction since no additional manual effort was needed to correct the resulting model. Moreover, the developers who create the detectors are very familiar with the system, meaning that they hold in depth the system characteristics and requirements. Requirement **R3** is about supporting a highly polyglot setting, as is typical of modern microservices. This seems infeasible for the approach by Haitzer and Zdun with regard to required time and effort, unless the state of the art on polyglot parser approaches and support for multiple well working grammars is significantly improved. Both new Approaches support this requirement well.

Requirement **R4** is about continuous comprehension. In order to test this, we have examined the differences in architecture and technologies used in the system in four additional releases: one prior to our case study, three later ones. We did not only test evolution in later releases, but also in prior ones, in order to be able to test whether our approach works for the removal of features. These are summarized in Table 5.3. Our approaches are able to detect a) the removal of services, b) the removal of system-level capabilities (e.g. Prometheus), and c) the modification of specific technologies (e.g. Go). As can be seen in the table, the additional time effort (in minutes) to the script containing the abstracted architecture model (execution script) to each release is negligible (a few minutes). However, there is a caveat for Approach 1: Due to the many small opportunistic detectors, different implementations for similar concerns and opportunistic code reuse (copy & paste) occurred. It is known that this leads to problems during evolution, such as changes not being carried through for all similar code fragments, or that overly specific code for particular detections can lead to breaking detectors.

Requirement **R5** concerns traceability from code to models. Both approaches establish trace links automatically. Requirement **R6**, i.e., how the amount of time (effort) for creating the architecture abstractions compares to the overall amount of effort for engineering the system, is fulfilled for Approach 1 and Approach 2: Compared to the COCOMO II estimate for delivering the case study project as an industrial solution, the efforts both for Approach 1 and Approach 2 are tiny. For instance, assuming 140 work hours per month, the case study construction would

have consumed, according to COCOMO II, 199080 minutes. That is, the case-specific effort for creating a model and detectors for Approach 1 would be 0.7%, and for Approach 2 0.1%. That is, according to our COCOMO II estimates both approaches are way beyond the set target of 1%. Please note that this does not work out, if generic code base and reusable detectors cannot be reused. Then we observed 4% for Approach 1 and 7% for Approach 2, which might still be acceptable for some projects, but are beyond our 1% target. For more extensive systems these numbers would be much smaller in comparison. Requirement **R7** is about the comparison of the approach to a manual reconstruction. If we compare to the manual reconstruction effort for the case study (see Section 5.4.1), we can see that Approach 1 requires a case-specific effort of 56.89% of the manual reconstruction effort; Approach 2 only requires 11.87% of the case-specific effort. Thus both approaches, when combined with a single manual reconstruction (or with our approach applied permanently from the inception of the project onwards), would require much less effort than periodically repeated manual reconstruction efforts. Assuming the existence of a large repository of reusable detectors (e.g., as an open-source project), Approach 2 would be vastly superior to Approach 1, too; without it, the substantial effort needed to create reusable detectors might eat up much of the benefit compared to Approach 1.

Table 5.2 summarizes and compares *Approach 1* and *Approach 2* based on the stated requirements. It is evident that, although both approaches meet the requirements we have set, there is a considerable difference in performance in terms of effort in favor of Approach 2.

Requirements	*Approach 1: Case-Specific Detectors*	*Approach 2: Reusable Detectors*
Component Model Reconstruction (R1&R2)	100% correctness (if detectors are not heuristics)	100% correctness (if detectors are not heuristics)
Polyglot Support (R3)	fully supported	fully supported
Continuous Comprehension (R4)	fully supported	fully supported
Traceability (R5)	automatically ensured	automatically ensured
Estimated Percentage of Overall System Development Effort (R6)	0.7%	0.1%
Comparison to Manual Reconstruction (R7) as Percentage of Manual Reconstruction	56.98%	6.76%

Table 5.2: Comparison between the two approaches on requirement fulfillment

5.7 Extending the Approach to Cases from Different Domains

To further assess whether our approach is also applicable in different case study settings, we applied it to two cases in the domain of modeling inter-service communication in API-centric communication models. We did this with the purpose of automatically detecting *asynchronous cycles in communication at the API-level*. These *unintended domain-based cyclic dependencies* [Wol16] manifest mainly on the conceptual level and less on the implementation level and are therefore considered relatively difficult to track exclusively through static code analysis

System Release	16.04.2018	**23.10.2019**	06.07.2020	26.08.2020	22.02.2021
System Lines of Code	3650	**5311**	5115	4997	5746
Number of System Elements	12 components 17 connectors	18 components 29 connectors	18 components 29 connectors	18 components 29 connectors	18 components 29 connectors
System Changes	•No Ratings service yet •Prometheus, Paypal Payment, Instana not yet added	•Ratings service has been added •Prometheus, Paypal Payment, Instana have been added	•Dispatch service has been modified	•Shipping service has changed framework from Java Spark to Spring Boot	•An Event Listener has been introduced in Ratings service.
Time Needed (in minutes) to adjust execution script (Approach 1 / Approach 2)	8 min / 5 min‡	**NA**	5 min / 3 min†	5 min / 3 min†	5 min / 3 min†

‡: compared to the case study release version, †: compared to the previous release

Table 5.3: Continuous Comprehension Support (release examined in case study in **bold**)

methods. However, collecting runtime information to track these dependencies can often be very time-consuming. Hence an approach that focuses on source code analysis would be preferred. To this end, we implemented essentially two different types of detectors: The first type of detectors was responsible for recognizing relevant architectural elements, like API Interfaces, API Operations, and specific calls and invocations that establish interservice communication. We, therefore, refer to them as *Hot Spot Detectors*. The second type, *Invocation Detectors*, are responsible for tracking call chains between the various hot spots, thus creating the final graph structure representing our communication model. As the approach should be offered as a reusable component to be used e.g. in a continuous delivery pipeline of a project, based on the data from our case study reported in Section 5.4.1, we selected Approach 2 for the two additional cases. We developed reusable detectors that were not bound to any project-specific implementation, apart from some project-specific code in order to reduce the implementation effort (see below).

The first API model case study was conducted on two versions of the open-source *Lakeside Mutual*[6] project. This project realizes the architecture of a fictional insurance company and consists of several Java Spring-based backend microservices. To detect the relevant hot spots and invocations, we had to implement ten detectors in total. While these were specific to Java-based communication technologies, such as *SpringRestControllers*, *FeignClients* or *JmsTemplates*, their usage would a) not be restricted to any concrete project and b) could easily be adapted for other Java-based microservice implementations. The overall implementation for analyzing the Lakeside Mutual project took 431 lines of code (LoC) for Java-specific detectors and 373 lines of generic code to orchestrate the detection process and generate the model, resulting in a total implementation size of 804 LoC. Using our detectors on the Fall 2020 revision of the project[7], we

[6] https://github.com/Microservice-API-Patterns/LakesideMutual

[7] https://github.com/Microservice-API-Patterns/LakesideMutual/tree/spring-term-2020

were able to identify 40 different API operations (35 synchronous and five asynchronous ones) and 16 interservice connectors between these operations. Based on the generated model, we were able to track two domain-based cycles in the system. Our detectors were also able to analyze the latest[8] version of the Lakeside Mutual project without requiring any changes to our existing code base. The analysis revealed a slight decrease in synchronous API operations (to 29), and we could verify that all cyclic dependencies we had detected in the previous version had been resolved. Although the architecture has undergone significant changes between the two revisions, this case study demonstrates that our detectors were still able to identify the relevant structural elements in both versions without adjustments, which underlines the reusability aspect of our approach.

In the second API model case study, we examined the communication structure of the *eShopOnContainers*[9] system, an open-source microservice reference implementation for the .NET technology stack. Compared to Lakeside Mutual project, this one uses a pure event-based asynchronous communication model for interservice communication. In addition, some service implementations pursue a domain-driven design approach, resulting in a more implicit invocation call chain within the services themselves. While writing generic reusable detectors to track these invocations would be possible, generically covering all of these cases would require a considerable amount of implementation effort. Therefore our detector implementation used some heuristics that were specifically tailored to the underlying project. Because of that, our implementation amounted to 162 lines of code for project-specific detectors and 181 lines of generic detectors that could also be reused for other C#-based microservice applications. Table 5.4 summarizes the implementation efforts of our case studies. Please note that due to the close syntactic relationship between Java and C#, the amount of language-specific detector code could be reduced by implementing some of the generic detectors in a language-agnostic way.

	System Size (LoC)[1]	Generic reusable code (LoC)	Language-specific reusable detector code (LoC)	Project-specific detector code (LoC)	Total implementation (LoC)
Lakeside Mutual	35.2K / 35.4K[2]	373	431	–	**804**
eShopOn Containers	81.4K	373	181	162	**716**

[1] all files except *json* configuration files, calculated with *cloc*: `http://cloc.sourceforge.net/`
[2] *Spring-Term 2020* branch / master branch *April 2021*

Table 5.4: Size Comparison between Detector Implementations for two Example Case Studies

The two API model cases show that the detector-based approach is well suited for problem-specific scenarios, too. The effort for implementation and configuration is significantly lower than using language-specific parsers, especially if these language parsers have to be kept up-to-date. While the generic detector approach might require some upfront implementation, this

[8] `https://github.com/Microservice-API-Patterns/LakesideMutual/commit/bdc6d30135149563c057dd30f21b7df68608c500`
[9] `https://github.com/dotnet-architecture/eShopOnContainers`

work amortizes considerably soon if more than one project revision needs to be analyzed, as seen in our first case. The ability to combine both approaches in the second case in order to reduce implementation effort shows that the approach is flexible, while the two projects show that detectors can be used to retrieve the system architecture automatically, eliminating the need for manual maintenance of the architecture model.

In terms of the design requirements, our approach satisfies in both cases R1 and R2 (adapted to the given context, e.g. it does not need to cover the entire system architecture). R3 is not applicable due to the mostly homogeneous code basis of the relevant parts of the two projects examined, but the detectors could easily be (re)written so as to cover both Java and C#-based systems. R4 is not relevant, as there is no manually created architectural model. R5 is implicitly supported by the approach, but traceability is not used any further in the cases. R6 and R7 are of limited relevance due to the different context, i.e. the focus on a subset of the system, and the automatic detection of relevant elements only; but the overall coding effort required in LoC (and thus, implicitly, time) is a fraction of the project size, even when leaving aside the gains from detector reuse relative to the changes in the project implementation over time.

5.8 Threats to Validity

It is important to consider the threats to validity during the design of the study to increase its validity. [WRH+12] distinguish four types: conclusion, internal, external, and construct validity. Internal validity address establishing a causal relationship between variables. It is not relevant for this study, as we do not aim to create a causal relationship between variables using statistical means.

Conclusion Validity: Threats to the conclusion are concerned with issues that affect the ability to draw the right conclusions between the treatments and the outcome of the study. As we do not apply statistical testing, related sources to conclusion validity reported by [WRH+12] do not apply to our study. As data reported about the resulting design and the case study ground truth is collected semi-formally (i.e., contains qualitative elements), there is the risk that the researchers' background, development experience, and understanding influence the interpretations. This risk is reduced as the different research team members carefully reviewed the steps taken by the other researchers. The risk is further reduced by the researchers' deep background in both the microservices domain and architecture abstraction methods. Finally, we included industry experts as authors to further reduce the bias. To ensure the reliability of our measures, we used objective measurements such as precise time recording and lines of code. For both a subjective element remains: other developers might have needed a different amount of time or structured their lines of code differently. Also, a few minor aspects of our measurements are estimated such as the common code base of Approach 1 and Approach 2. As we only aim for the measurements to provide a very rough estimate (i.e., in orders of magnitude), we believe possible differences to be negligible. The choice to perform a so-called *mechanism experiment* [Wie14], i.e. implement the architecture model abstraction prototypically for the case ourselves, might have negatively influenced the reliability of the treatment implementation for the reported measurements and the comparison between the two approaches. Regarding RQ2.1 and RQ2, both of which investigate whether and how a feasible design is possible, the reliability of the treatment implementation is

given, as two feasible designs have been found. We do not claim that the found designs are the only possible designs or optimal.

Construct Validity: Construct validity is concerned with obtaining the right measures and instruments for the phenomena being studied. Our study combines design science and case study research to minimize possible mono-method bias. There is a risk of a possible interaction between treatments, as the treatments in the two approaches are applied one after another. Again, this does not impact the research for RQ2.1 and RQ2.2. A learning effect may have impacted the comparison between the two approaches. However, we argue that our goal is only to understand the differences in orders of magnitude, and do not claim generalizability of the precise measures. Continuous comprehension support, i.e., the ability of our detectors to detect all changes in the implementation (addition, modification, removal) of connectors in every release version of the system, has been tested in Section 5.6.2. The only problem we anticipate would be in the case where major changes occur between releases, but this goes beyond continuous comprehension, and effectively requires a *de novo* architectural reconstruction. Additionally, the evolution aspect has been tested and validated in three systems already for the predecessor work [HZ14] of the approach presented here.

External Validity: Threats to external validity are conditions that limit our ability to generalize the results. Our research has been performed by researchers and not in an industrial setting. This might limit the generalizability to industrial practices. As our main line of research aims to find a feasible design in context, the threat appears negligible. For the approach comparisons and precise measurements, the threat is realistic for the reasons given above. In addition, in order to come up with a sound design, we applied substantial refactoring effort throughout the study, which may not reflect current common practice in industry. That is, the reported effort might be slightly higher than what a "quick and dirty" implementation in industry would actually require. On the other hand, no effort was needed for meetings, design sessions, deployment, and thus also not reported. This might substantially increase the relevant measurements when creating industry-grade solutions. As all this should not considerably impact the *relative* difference between Approach 1 and Approach 2, we believe the threat to be negligible for RQ2.3. Finally, the basic soundness of the approach has been tested on open-source case studies realized by industry practitioners. This way we have ensured that our approach is applicable in a realistic setting. The fact that the case study projects are merely demonstrators and not a production-ready systems might result in differences compared to actual industrial implementations. From our point of view, the risk is that real-life industry systems are usually larger and less well-structured than the system we have studied. From our experiences with industry systems in the microservice domain (we included industry experts in the author team to confirm this point), many current industry systems are realized in a similar fashion, and we are confident that a) the chosen system is a representative cross-cut of current practices in the microservice domain and b) that it applies a sufficient number of different technologies so as to be a representative of industrial polyglot systems. The risk that our methods might require additional engineering when applied to very large-scale systems remains, but this is a question of additional adaptation and coverage effort, and does not invalidate the basis of our approach. On the contrary, once a basic set of technologies has been covered and a sufficiently large library of detectors exists, and with a more automated method for detecting architecture changes so that the manual model maintenance is further

reduced, we are confident that the savings in effort due to the reusability of the detectors will end up being much more pronounced in a continuously-evolved, large-scale system than in our case study.

5.9 Conclusions and Future Work

In this chapter we have reported on a design-science study combined with case-study research for studying methods for comprehending the component architecture of highly polyglot systems, as exemplified by state-of-the-art microservices systems. With regard to **RQ2.1**, we conclude that our study revealed that the approach taken by [HZ14] (discussed in Section 5.3) probably allows to design a highly accurate architecture abstraction, but that it involves considerable effort in a highly polyglot setting. By contrast, both of the semi-automatic, detector-based approaches developed in our study work well in the highly polyglot setting and fulfill all our requirements. We further observed that the reusable detectors from Approach 2 tend to enable cleaner solutions: due to the detector specificity, similar detections sometimes led to different approaches in Approach 1, whereas in Approach 2 – as reuse was already a goal of the design – the solution was provided by more generic, common code. Thus, Approach 2 makes it more unlikely that recurring code issues or possible defects stay undetected. Moreover, in a system with 11 different technologies, we managed to retrieve its component architecture with a reasonable effort (i.e., within a reasonably short period of time), with high hopes for high reusability and small deltas during future evolution of the same system. With regard to **RQ2.2**, all three approaches studied can support continuous comprehension well. The approach by Haitzer and Zdun is limited, as many parser technologies and grammars need continuous maintenance and testing effort; the two detector approaches developed as part of the present study work better in this regard. Approach 2 is superior to the one from Approach 1, as new or changed detection requirements need to be realized only once, in reusable detectors, and are then applied automatically for all projects using those same detectors. Approach 1, in contrast, would require searching for all related custom detectors and changing them individually. As a result, the reusable detector approach also seems to be slightly better suited for RQ2.2, as it requires a lot of discipline and refactoring to reach the same quality of evolution stability in Approach 1 as is built into Approach 2. For **RQ2.3** we have compared our two approaches in terms of time (effort), using time recording, and lines of code measurements. The results indicate that creating new reusable detectors for a single project requires substantially higher time (effort) for Approach 2 than for Approach 1. Creating the case-specific code requires significantly less time (effort) for Approach 2 than for Approach 1. Our data indicates there is a break-even point where the reusable detector method pays off. A very rough estimation, based on the averages in our case study data, indicates that the break even-point for effort is reached at about 6 uses of a reusable detector. For the lines of code measurements, it is reached after 8 uses. This does not consider the extra effort needed in Approach 1 compared to Approach 2 during software evolution; to cover those, further studies of an evolving software system would be needed. Based on all data and observations, we thus would recommend the reusable detector approach, unless a project is certain to use all detectors a very few times at maximum. We have also tested the adaptability of the same approach to different tasks, which are well suited to be used in combination with the architecture abstraction detectors, potentially greatly reducing the

already small manual overhead required by our method.

As future research we plan to further investigate the methods reported in this chapter in an industry context, and for other kinds of models than component models. We further plan to exploit the traceability links provided through our method in various research approaches. It would also be interesting to investigate whether a more precise prediction of required time and efforts is possible.

6 Assessing Architecture Conformance to Coupling-Related Patterns and Practices in Microservices

Microservice architecture is a widely adopted approach for building applications that offer high scalability, independent development, and adaptability to change, supporting the use of various technologies. One of the essential principles of this architecture is the high degree of independence among individual microservices, which is commonly achieved through loose coupling. Despite the abundance of established patterns and best practices, most microservice-based systems, in whole or in part, do not adhere to them, making a manual assessment of conformance impractical for large-scale systems. In this chapter we provide the groundwork for an automated approach to assess conformance to coupling-related patterns and practices specific to microservice architectures.

6.1 Introduction

Microservice architectures [New15, Zim17, LF04] describe an application as a collection of autonomous and loosely coupled services, typically modeled around a domain. Key microservice tenets are development in independent teams, cloud-native technologies and architectures, polyglot technology stacks including polyglot persistence, lightweight containers, loosely coupled service dependencies, high releasability, and continuous delivery [Zim17]. Many architectural patterns that reflect recommended "best practices" in a microservices context have already been published in the literature [Ric17, ZSZ+19, Sko19]. The fact that microservice-based systems are complex and polyglot means that an automatic or semi-automatic assessment of their conformance to these patterns is difficult: real-world systems feature combinations of these patterns, and different degrees of violations of the same; and different technologies in different parts of the system implement the patterns in different ways, making the automatic parsing of code and identification of the patterns a haphazard process.

This work focuses on describing a method for assessing architecture conformance to coupling-related patterns and practices in microservice architectures. Coupling between microservices is caused by existence of dependencies, e.g. whenever one service calls another service to fulfill a request or share data. Loose coupling is an established topic in service-oriented architectures [Zim17] but the application to the specific context of microservice architectures has not, to our knowledge, been examined so far.

Strong coupling is conflicting with some of the key microservice tenets mentioned above. In particular, releasability, which is a highly desirable characteristic in modern systems due to the emergence of DevOps practices, relies on the rapid and independent release of individual

microservices, and is compromised by strong dependencies between them. For the same reason, development in independent teams becomes more difficult, and independent deployment of individual microservices in lightweight containers is also impeded. This work covers three broad coupling aspects: *Coupling through Databases*, resulting from reliance on commonly accessed data via shared databases; *Coupling through Synchronous Invocations*, resulting from synchronous communication between individual services; and *Coupling through Shared Services*, which arises through the dependence on common shared services (for details see Section 6.3).

In reality, of course, no microservice system can support *all* microservice tenets well at the same time. Rather the *architectural decisions* for or against the use of specific patterns and practices must reflect a trade-off between ensuring the desired tenets and other important quality attributes [HWB17, Zim17]. From these considerations, this work aims to study the following research questions:

- **RQ3.1** How can we automatically assess conformance to loose coupling-related patterns and practices in the context of microservice architecture decision options?
- **RQ3.2** How well do measures for assessing coupling-related decision options and their associated tenets perform?
- **RQ3.3** What is a set of minimal elements needed in a microservice architecture model to compute such measures?

In pursuing of these questions, we surveyed the relevant literature (Section 6.2) and gathered knowledge sources about established architecture practices and patterns, their relations and tenets in form of a *qualitative study on microservice architectures*. This enabled us to create a meta-model for the description of microservice architectures, which was verified and refined through iterative application in modelling a number of real-world systems, as outlined in Section 6.4. We manually assessed all models and model variants on whether each decision option is supported, thereby deriving an objective *ground truth* (Section 6.5). As the basis for an automatic assessment, we defined a number of generic, technology-independent metrics to measure architecture conformance to the decision options, i.e. at least one metric per major decision option (Section 6.6). These metrics (and combinations thereof) were applied on the models and model variants to derive a numeric assessment, and then compared to the ground truth assessment via an ordinal regression analysis (Section 6.7). Section 6.8 discusses the results of our approach, as well as its limitations and potential threats to validity. Finally, in Section 6.9 we draw our conclusions and discuss options for future work.

6.2 Related Work

Many studies focus on best practices for microservice architectures. Richardson [Ric17] has published a collection of microservice patterns related to major design and architectural practices. Patterns related to microservice APIs have been introduced by Zimmermann et al. [ZSZ+19], while Skowronski [Sko19] collected best practices for event-driven microservice architectures. Microservice fundamentals and best practices are also discussed by Fowler and Lewis [LF04], and are summarized in a mapping study by Pahl and Jamshidi [PJ16]. Taibi and Lenarduzzi [TL18]

study microservice "bad smells", i.e. practices that should be avoided (which would correspond to violations in our work).

Many software metrics-related studies for evaluating the system architecture and individual architectural components exist, but most of them are not specific to the microservices domain. Allen et al. [AGG06, AGG07] study component metrics for measuring a number of quality attributes, e.g. size, coupling, cohesion, dependencies of components, and the complexity of the architecture. Additional studies for assessing quality attributes related to coupling and cohesion have been proposed and validated in the literature [CK94, BD02, HCN98, BBM96]. Furthermore, a small number of studies [PW09, ZNL17, BWZ17] propose metrics specifically for assessing microservice-based software architectures. Although these works study various aspects of architecture, design metrics, and architecture-relevant tenets such as coupling and independent deployment, their approach is usually generic. None of the works covers all the related software aspects for measuring coupling in a microservice context: the use of databases, system asynchronicity, and shared components. This is the overarching perspective of our work, and the chief contribution of this study.

6.3 Decisions

In this section, we briefly introduce the three coupling-related decisions along with their decision options (i.e. the relevant patterns and practices). We also discuss the impact on relevant microservice tenets, which we later on use as an argumentation for our manual ground truth assessment in Section 6.5.

Inter-Service Coupling through Databases. One important decision in microservice-based systems is data persistence, which needs to take into account qualities such as reliability and scalability, but also adhere to microservice-specific best practices, which recommend that each microservice should be loosely coupled and thus able to be developed, deployed, and scaled independently [LF04]. At one extreme of the scale, one option is *No Persistent Data Storage*, which is applicable only for services whose functions are performed on transient data. Otherwise, the most recommended option is the *Database per Service* pattern [Ric17]: each service has its own database and manages its own data independently. Another option, which negatively affects *loose coupling*, is to use a *Shared Database* [Ric17]: a service writes its data in a common database and other services can read these data when required. There are two different ways to implement this pattern: in *Data Shared via Shared Database* multiple services share the same table, resulting in a strongly coupled system, whereas in *Databased Shared but no Data Sharing* each service writes to and reads from its own tables, which has a lesser impact on coupling.

Inter-Service Coupling through Synchronous Invocations. Service integration is another core decision when building a microservice-based system. A theoretically optimal system of independent microservices would feature no communication between them. Of course, services need to communicate in reality, and so the question of integrating them so as to not result in tight inter-service coupling becomes paramount. The recommended practice is that communication between the microservices should be, as much as possible, asynchronous. This

can be achieved through several patterns which are widely implemented in typical technology stacks: the *Publish/Subscribe* [HW03a] pattern, in which services can subscribe to a channel to which other services can publish; the use of a *Messaging* [HW03a] middleware, which decouples communication by using a queue to store messages sent by the producer until they are received by the consumer; the *Data Polling* [Ric17] pattern, in which services periodically poll other services for data changes; and the *Event Sourcing* [Ric17] pattern, that ensures that all changes to application state are stored as a sequence of events; *Asynchronous Direct Invocation* technique, in which services communicate asynchronously via direct invocations. Applying these patterns ensures loose coupling (to different degrees), and increases the system reliability.

Inter-Service Coupling through Shared Services. Many of the microservice patterns focus on system structure, i.e. avoiding services sharing other services altogether, or at least not in a strongly coupled way. An optimal system in terms of architecture quality should not have any shared service. In reality, this is often not feasible, and in larger systems service sharing leads to chains of transitive dependencies between services. This is problematic when a service is unaware of its transitive dependencies, and of course for the shared service itself, where the needs of its dependents must always be taken into account during its evolution. We define three cases: a *Directly Shared Service* is a microservice which is directly linked to and required by more than one other service; a *Transitively Shared Service* is a microservice which is linked to other services via at least one intermediary service; and a *Cyclic Dependency* [GM14] which is formed when there is a direct or transitive path that leads back to its origin, i.e. that allows a service to be ultimately dependent on itself after a number of intermediary services. Cyclic dependencies often emerge inadvertently through increasing complexity over the system's lifecycle, and require extensive refactoring to resolve. All three cases are inimical to the principle of *loose coupling* as well as to system qualities such as *performance*, *modifiability*, *reusability*, and *scalability*.

6.4 Research and Modeling Methods

In this section, we summarize the research and modeling methods applied in our study. The code and models used in and produced as part of this study have been made available online for reproducibility [1].

6.4.1 Research Method

Figure 6.1 shows the research steps from initial data collection to final data analysis. For the data collection phase we conducted a multi-vocal literature study using *web resources*, *public repositories*, and *scientific papers* as sources [GFM17]. We then analyzed the data collected using qualitative methods based on *Grounded Theory* [CS90] coding methods, such as open and axial coding, and extracted the three core architectural decisions described in the previous section along with their corresponding decision drivers and impacts. As data for our further research we used generated models taken from the *Model Generation* process, described below. We defined a rating scheme for systematic assessment based on support or violation of core practices and tenets.

[1] https://bit.ly/2WmFP3N

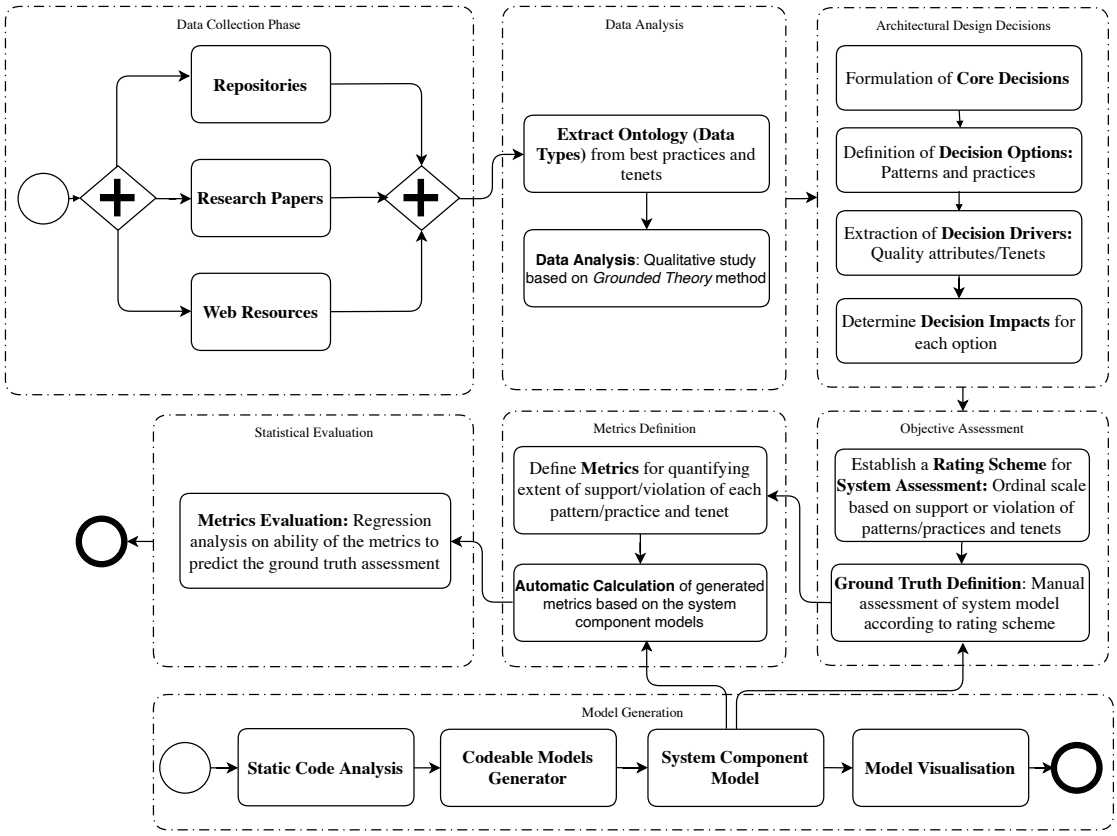

Figure 6.1: Overview diagram of the research method followed in this study

From these we derived a ground truth for our study (the ground truth and its calculation rules are described in Section 6.5) as well as a set of metrics for automatically calculating conformance to each individual pattern or practice per decision. We then used the ground truth data to assess how well the hypothesized metrics can possibly predict the ground truth data by performing an ordinal regression analysis. Ordinal regression is a widely used method for modeling an ordinal response's dependence on a set of independent predictors, which is applicable in a variety of domains. For the ordinal regression analysis we used the *lrm* function from the *rms* package in R [FEH15].

6.4.2 Model Generation

We began by performing an iterative study of a variety of microservice-related knowledge sources, and we gradually refined a meta-model which contains all the required elements to help us reconstruct existing microservice-based systems. In order to investigate the ontology, and to evaluate the meta-model's efficiency, we gathered a number of microservice-based systems, summarized in Table 3.1. Each is either a system published by practitioners (on GitHub and/or practitioner blogs) or a system variant adapted from a published example according to discussions in the relevant literature in order to explore the possible decision space. Apart from the specific

variations described in Table 3.1 all other system aspects remained the same as in the base models.

The systems were taken from 9 independent sources in total. They were developed by practitioners with microservice experience, and they are representative of the best practices summarized in Section 6.3. We performed a fully manual static code analysis for those models where the source code was available (7 of our 9 sources; two were modeled from documentation published by the practitioners).

To create our models, we used the Python modeling library CodeableModels described in Chapter 3.7.

The result is a set of precisely modeled component models of the examined software systems (modeled using the techniques described below). This resulted in a total of 27 models summarized in Table 3.1. We assume that our evaluation systems are, or reflect, real-world practical examples of microservice architectures. As many of them are open source systems with the purpose of demonstrating practices or technologies, they are at most of medium size and modest complexity, though.

6.4.3 Methods for Modeling Microservice Component Architectures

From an abstract point of view, a microservice-based system is composed of components and connectors, with a set of component types for each component and a set of connector types for each connector. For modeling microservice architectures we followed the method reported in our previous work [ZNL17].

6.5 Ground Truth Calculations

In this section, we present and describe the calculation of the ground truth assessment for each of the decisions from Section 6.3. The results of those assessments are reported in Table 6.1. The assessment begins with a manual evaluation by the authors on whether each of the relevant patterns (decision options) is either Supported, Partially Supported, or Not Supported (**S**, **P**, **N** in Table 6.1). Based on this and informed by the description of the impacts of the various decision options in Section 6.3, we combined the outcome of all decision options to derive an ordinal assessment on how well the decision as a whole is supported in each model, using the ordinal scale: [**++**: very well supported, **+** : well supported, **o** : neutral, **-**: badly supported, **--**: very badly supported]. This was done according to best practices documented in literature. For instance, following the ordinal scale the assessment for the model BM1 is **+** : well supported, since a) option *Database per Service* is not supported, b) some services have a shared database, but c) they do not share data via the shared database.

For the *Inter-Service Coupling through Databases* decision, we derive the following scoring scheme for our ground truth assessment:

- **++**: All services (which require data persistence) have individual databases *Database per Service*.
- **+**: Some services have *Shared Databases* and no *Data Shared via the Shared Databases*.
- **o**: All services have *Shared Databases* and no *Data Shared via the Shared Databases*.

	BM1	BM2	BM3	CO1	CO2	CO3	CI1	CI2	CI3	CI4	EC1	EC2	EC3	ES1	ES2	ES3	FM1	FM2	FM3	HM1	HM2	RM1	RM2	RM3	RS	TH1	TH2
Database-based Inter-Service Coupling																											
Database per Service	N	S	S	S	S	P	S	S	S	S	S	S	N	S	N	P	S	S	S	S	S	N	S	S	N	S	S
Shared Database	P	N	N	N	N	P	N	N	N	N	N	N	S	N	S	P	N	N	N	N	N	S	N	N	S	N	N
Data Shared via Shared DB	N	N	N	N	N	P	N	N	N	N	N	N	S	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Assessments	+	++	++	++	++	-	++	++	++	++	++	++	--	++	o	+	++	++	++	++	++	o	++	++	o	++	++
Inter-Service Coupling through Synchronous Invocations																											
	BM1	**BM2**	**BM3**	**CO1**	**CO2**	**CO3**	**CI1**	**CI2**	**CI3**	**CI4**	**EC1**	**EC2**	**EC3**	**ES1**	**ES2**	**ES3**	**FM1**	**FM2**	**FM3**	**HM1**	**HM2**	**RM1**	**RM2**	**RM3**	**RS**	**TH1**	**TH2**
Asynchronous Direct Interconnections	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	S	S	P	P	N	N	N	P	N	N
PubSub/Event Sourcing Interconnections	S	N	N	N	N	N	N	N	N	N	N	S	N	P	N	N	N	N	N	N	S	P	N	N	N	N	P
Asynch Inter-communication via API GW	N	N	S	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Shared Database Interconnections	N	N	N	N	N	P	N	N	N	N	N	N	S	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Messaging Interconnections	N	N	N	S	S	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	P	N	N	P	N	N
Assessments	++	--	-	--	++	-	--	--	--	--	--	++	o	+	--	--	--	+	+	-	++	+	--	--	o	--	+
Inter-Service Coupling through Shared Services																											
	BM1	**BM2**	**BM3**	**CO1**	**CO2**	**CO3**	**CI1**	**CI2**	**CI3**	**CI4**	**EC1**	**EC2**	**EC3**	**ES1**	**ES2**	**ES3**	**FM1**	**FM2**	**FM3**	**HM1**	**HM2**	**RM1**	**RM2**	**RM3**	**RS**	**TH1**	**TH2**
Direct Service Sharing	N	N	N	N	N	N	P	N	P	N	N	N	N	P	P	P	P	P	P	P	N	N	N	N	P	P	N
Transitively Shared Services	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	P	N	N	N	S	N	N	N	N
Cyclic Dependencies	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	S	N	N	N	N	S	N	N	N
Assessments	++	++	++	++	++	++	o	++	o	++	++	++	++	o	o	o	o	o	--	o	++	++	o	--	o	o	++

Table 6.1: Ground truth assessment results

- **-**: Some services have *Shared Databases* and *Data Shared via the Shared Databases*.
- **--**: All services have *Shared Databases* and *Data Shared via the Shared Databases*.

From the *Inter-Service Coupling through Synchronous Invocations* decision, we derive the following scoring scheme for our ground truth assessment:

- **++**: All services communicate asynchronously via *Message Brokers* or *Publish/Subscribe* or *Stream Processing*
- **+**: All services communicate asynchronously via *API Gateway* or *HTTP Polling* or *Direct Asynchronous calls*, or (some) via *Message Brokers* or *Publish/Subscribe* or *Stream Processing*.
- **o**: None or some services communicate asynchronously and all other services communicate asynchronously via *Data Sharing* (e.g. Shared DB).
- **-**: None or some services communicate asynchronously, none or some communicate asynchronously via *Data Sharing*, some services communicate synchronously.
- **--**: All services communicate synchronously.

Finally, from the *Inter-Service Coupling through Shared Services* decision, we derive the following scoring scheme for our ground truth assessment:

- **++**: None of the services is a *Directly Shared Service* or *Transitively Shared Service* and no *Cyclic Dependencies* exist.
- **+**: Some of the services are *Transitively Shared Services* , but none are *Directly Shared Services* and no *Cyclic Dependencies* exist.
- **o**: Some or none of the services are *Transitively Shared Services* and some are *Directly Shared Services*, but no *Cyclic Dependencies* exist.
- **-**: Some of the services are *Transitively Shared Services* and all other services are *Directly Shared Services*, but no *Cyclic Dependencies* exist.
- **--**: There are *Cyclic Dependencies* or all the services are *Transitively Shared Components* and all the services are *Directly Shared Components*.

6.6 Metrics

In this section, we describe the metrics we have hypothesized for each of the decisions described in Section 6.3. All metrics, unless otherwise noted, are a continuous value with range from 0 to 1, with 1 representing the optimal case where a set of patterns is fully supported, and 0 the worst-case scenario where it is completely absent.

6.6.1 Metrics for Inter-Service Coupling through Databases Decision

Database Type Utilization (DTU) metric. This metric returns the number of the connectors from *Services* to *Individual Databases* in relation to the total number of *Service–to–Database* connectors. This way, we can measure how many services are using individual databases.

$$DTU = \frac{\textit{Database per Service Links}}{\textit{Total Service-to-Database Links}}$$

Shared Database Interactions (SDBI) metric. Although a *Shared Database* is considered as an anti-pattern in microservices, there are many systems that make use of it either partially or completely. To measure its presence in a system, we count the number of interconnections via a *Shared Database* compared to the *total number of service interconnections*.

$$SDBI = \frac{\textit{Service Interconnections with Shared Database}}{\textit{Total Service Interconnections}}$$

6.6.2 Metrics for Inter-Service Coupling through Synchronous Invocations Decision

Service Interaction via Intermediary Component (SIC) metric. We defined this metric to measure the proportion of service interconnections via asynchronous relay architectures such as *Message Brokers*, *Publish/Subscribe*, or *Stream Processing*. These represent the best current practices, and are not exhaustive; should any new architectures emerge, these should be added to this list.

$$SIC = \frac{\textit{Service Interconnections via [Message Brokers | Pub/Sub | Stream]}}{\textit{Total Service Interconnections}}$$

Asynchronous Communication Utilization (ACU) metric. This metric measures the proportion of the sum of asynchronous service interconnections (via *API Gateway* / *HTTP Polling* / *Direct calls* / *Shared Database*) to the total number of service interconnections.

$$ACU = \frac{\textit{Asynchronous Service Interconnections via [API | Polling | Direct Calls | Shared DB]}}{\textit{Total Service Interconnections}}$$

6.6.3 Metrics for Inter-Service Coupling through Shared Services Decision

Direct Service Sharing (DSS) metric. For measuring DSS we count all the directly shared services and set this number in relation to the total number of system services. To this add all the shared services connectors in relation to the total number of services interconnections. This gives

us the proportion of the directly shared elements in the system.

$$DSS = \frac{\frac{\textit{Shared Services}}{\textit{Total Services}} + \frac{\textit{Shared Services Connectors}}{\textit{Total Service Interconnections}}}{2}$$

Transitively Shared Services (TSS) metric. For measuring TSS we count all the transitively shared services and set this number in relation to the total number of system services. To this we add all the transitively shared service connectors in relation to the total number of service interconnections. This gives us the proportion of the transitively shared elements in the system.

$$TSS = \frac{\frac{\textit{Transitively Shared Services}}{\textit{Total Services}} + \frac{\textit{Transitively Shared Services Connectors}}{\textit{Total Service Interconnections}}}{2}$$

Cyclic Dependencies Detection (CDD) metric. Let $SG = (S, C)$ be the service graph, S the set of service nodes, and C the set of connector edges in a microservice model. Based on the generic definition of closed paths, we define a *closed service path* in SG as a sequence of services $s_1, s_2, \ldots, s_n$ (each service $\in S$) such that $(s_n, s_n + 1) \in C$ is a directed connector between services for $i = 1, 2, \ldots, n$ and $s_1 = s_n$. A *service cycle* is a closed service path in which no service node is repeated except the first and last, and which contains at least two distinct service nodes. Let $ServiceCycles()$ return the set of all service cycles in a service graph. CDD returns 1 (True) if there is at least one cyclic dependency in the model:

$$CDD = \begin{cases} 1: & \text{if } |ServiceCycles(SG)| = 0 \\ 0: & otherwise \end{cases}$$

6.6.4 Metrics Calculation Results

We note that for the *Inter-Service Coupling through Shared Services* decision as well as *SDBI* metric, our metrics scale is reversed in comparison to the other two decisions, because here we detect the *presence of an anti-pattern*: the optimal result of our metrics is 0, and 1 is the worst-case result.

The metrics results for each model per decision metric are presented in Table 6.2.

6.7 Ordinal Regression Analysis Results

The dependent outcome variables are the ground truth assessments for each decision, as described in Section 6.5 and summarized in Table 6.1. The metrics defined in Section 6.6 and summarized in Table 6.2 are used as the independent predictor variables. The ground truth assessments are ordinal variables, while all the independent variables are measured on a scale from 0.0 to 1.0. The objective of the analysis is to predict the likelihood of the dependent outcome variable for each of the decisions by using the relevant metrics for each decision.

Metrics	BM1	BM2	BM3	CO1	CO2	CO3	CI1	CI2	CI3	CI4	EC1	EC2	EC3	
Database-based Inter-Service Coupling														
DTU	0.33	1.00	1.00	1.00	1.00	0.60	1.00	1.00	1.00	1.00	1.00	1.00	0.00	
SDBI	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	
Inter-Service Coupling through Synchronous Invocations														
SIC	1.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	
ACU	0.00	0.00	1.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	
Inter-Service Coupling through Shared Services														
DSS	0.00	0.00	0.00	0.00	0.00	0.00	0.20	0.00	0.38	0.00	0.00	0.00	0.00	
TSS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
CDD	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	

Metrics	ES1	ES2	ES3	FM1	FM2	FM3	HM1	HM2	RM1	RM2	RM3	RS	TH1	TH2
Database-based Inter-Service Coupling														
DTU	1.00	0.00	0.33	1.00	1.00	1.00	1.00	1.00	0.00	1.00	1.00	0.66	1.00	1.00
SDBI	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Inter-Service Coupling through Synchronous Invocations														
SIC	0.60	0.00	0.00	0.00	0.00	0.00	0.00	0.80	1.00	0.00	0.00	0.11	0.00	0.60
ACU	0.00	0.00	0.00	0.00	1.00	0.08	0.50	0.20	0.00	0.00	0.00	0.11	0.00	0.00
Inter-Service Coupling through Shared Services														
DSS	0.27	0.34	0.34	0.62	0.47	0.55	0.52	0.00	0.00	0.00	0.00	0.36	0.33	0.00
TSS	0.00	0.00	0.00	0.00	0.00	0.18	0.00	0.00	0.00	0.18	0.16	0.00	0.00	0.00
CDD	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00

Table 6.2: Metrics Calculation Results

Each resulting regression model consists of a *baseline intercept* and the independent variables multiplied by *coefficients*. There are different intercepts for each of the value transitions of the dependent variable ($\geq$*Badly Supported*, $\geq$*Neutral*, $\geq$*Well Supported*, $\geq$*Very Well Supported*), while the coefficients reflect the impact of each independent variable on the outcome. For example, a positive coefficient, such as +5, indicates a corresponding five-fold increase in the dependent variable for each unit of increase in the independent variable; conversely, a coefficient of -30 would indicate a thirty-fold decrease.

The statistical significance of each regression model is assessed by the p-value; the smaller the p-value, the stronger the model. A p-value smaller than 0.05 is generally considered statistically significant. In Table 6.3, we report the p-values for the resulting models, which in all cases are very low, indicating that the sets of metrics we have defined are able to predict the ground truth assessment for each decision with a high level of accuracy.

Intercepts/Coefficients	*Value*	*Model p-value*
Database-based Inter-Service Coupling		
Intercept (≥Badly Supported)	*2.6572*	*1.706019e-06*
Intercept (≥Neutral)	*0.8789*	
Intercept (≥Well Supported)	*-1.3820*	
Intercept (≥Very Well Supported)	*-3.1260*	
Metric Coefficient (DTU)	*6.4406*	
Metric Coefficient (SDBI)	*–3.7048*	
Inter-Service Coupling through Synchronous Invocations		
Intercept (≥Badly Supported)	*-2.6973*	*6.705525e-11*
Intercept (≥Neutral)	*-4.4087*	
Intercept (≥Well Supported)	*-5.8513*	
Intercept (≥Very Well Supported)	*-15.3677*	
Metric Coefficient (SIC)	*17.3520*	
Metric Coefficient (ACU)	*6.5520*	
Inter-Service Coupling through Shared Services		
Intercept (≥Neutral)	*59.4089*	*1.625730e-10*
Intercept (≥Very Well Supported)	*9.7177*	
Metric Coefficient (DSS)	*-82.4474*	
Metric Coefficient (TSS)	*-122.2583*	
Metric Coefficient (CDD)	*-57.4650*	

Table 6.3: Regression Analysis Results

6.8 Discussion

In this section, we first discuss what we have learned in our study that helps to answer the research questions and then discuss potential threats to validity.

6.8.1 Discussion of Research Questions

To answer **RQ3.1** and **RQ3.2**, we proposed a set of generic, technology-independent metrics for each coupling-related decision, and to each decision option corresponds at least one metric. We objectively assessed for each model how well patterns and/or practices are supported for establishing the ground truth, and extrapolated this to how well the broader decision is supported. We formulated metrics to numerically assess a pattern's implementation in each model, and performed an ordinal regression analysis using these metrics as independent variables to predict the ground truth assessment. Our results show that every set of decision-related metrics can predict our objectively evaluated assessment with high accuracy. This suggests that automatic metrics-based assessment of a system's conformance to the tenets embodied in each design decision is possible with a high degree of confidence.

Here, we make the assumption that the source code of a system can be mapped to the models used in our work. To enable this, we used rather simplistic modeling means, which can rather easily be mapped from a specific source code to the system models. However, it should be noted that full automation of this mapping is an additional effort that needs to be considered and is the subject of ongoing work on our part.

Regarding **RQ3.3**, we consider that existing modeling practices can be easily mapped to our microservice meta-model and there is no need for major extensions. More specifically, for completing the modeling of our evaluation system set, we needed to introduce 25 component types and 38 connector types, ranging from general notions such as the *Service* component type, to very technology-specific classes such as the *RESTful HTTP* connector, which is a subclass of *Service Connector*. Our study shows that for each pattern and practice embodied in each decision, and the proposed metrics, only a small subset of the meta-model is required.

The decisions *Inter-Service Coupling through Databases* and *Inter-Service Coupling through Shared Services* require to model at least the *Service* and the *Database* component types and the technology-related connector types (e.g. *Database Connector*, *RESTful HTTP* and *Asynchronous Connector*) and the read/write process which explicitly modeled in the *Database Connector* type. The *Inter-Service Coupling through Synchronous Invocations* decision requires a number of additional components (e.g. *Event Sourcing, Stream Processing, Messaging, PubSub*) and the respective connectors (e.g. *Publisher, Subscriber, Message Consumer*, *Messages Producer*, *RESTful HTTP* and *Asynchronous Connector*) to be modeled.

6.8.2 Threats to Validity

We deliberately relied on third-party systems as the basis for our study to increase internal validity, thus avoiding bias in system composition and structure. It is possible that our search procedures introduced some kind of unconscious exclusion of certain sources; we mitigated this by assembling an author team with many years of experience in the field (including substantial industry experiences), and performing very general and broad searches. Given that our search was not exhaustive, and that most of the systems we found were made for demonstration purposes, i.e. relatively modestly sized, this means that some potential architecture elements were not included in our meta-model. In addition, this raises a possible threat to external validity of generalization to other, and more complex, systems. We nevertheless feel confident that the systems documented are a representative cross-cut of current practices in the field, as the points of variance between them were limited and well attested in the literature. Another potential threat is the fact that the variant systems were derived by the author team. However, this was done according to best practices documented in literature. We carefully made sure only to change specific aspects in a variant and keep all other aspects stable. That is, while the variants do not represent actual systems, they are reasonable evolutions of the original designs.

The modeling process is also considered as source of internal validity threat. The models of the systems were repeatedly and independently cross-checked by the author team that has considerable experience in similar methods, but the possibility of some interpretative bias remains: other researchers might have coded or modeled differently, leading to different models. As a mitigation, we also offer the whole models and the code as open access artifacts for review. Since we aimed only to find one model that is able to specify all observed phenomena, and this was achieved, we consider this threat not to be a major issue for our study. The ground truth assessment might also be subject to different interpretations by different practitioners. For this purpose, we deliberately chose only a three-step ordinal scale, and given that the ground truth evaluation for each decision is fairly straightforward and based on best practices, we do not consider our interpretation controversial. Likewise, the individual metrics used to evaluate the

presence of each pattern were deliberately kept as simple as possible, so as to avoid false positives and enable a technology-independent assessment. As stated previously, generalization to more complex systems might not be possible without modification. But we consider that the basic approach taken when defining the metrics is validated by the success of the regression models.

6.9 Conclusions and Future Work

Our approach considered that it is achievable to develop a method for automatically assessing coupling related tenets in microservice decisions based on a microservice system's component model. We have shown that this is possible for microservice decision models that contain patterns and practices as decision options. In this work, we first modeled the key aspects of the decision options using a minimal set of component model elements. These could be possibly automatically extracted from the source code. Then we derived at least one metric per decision option and used a small reference model set as a ground truth. We then used ordinal regression analysis for deriving a predictor model for the ordinal variable. The statistical analysis shows that each decision related metrics are quite close to the manual, pattern-based assessment.

There are many studies related on metrics for component model and other architectures so far, but specifically for microservice architectures and their coupling related tenets have not been studied. Based on our discussion in Section 6.2, assessing microservice architectures using general metrics it is not very helpful. Our approach is one of the first that studies a metrics-based assessment of coupling-related tenets in the microservices domain. We aim to a continuous assessment, i.e. we envision an impact on continuous delivery practices, in which the metrics are assessed with each delivery pipeline run, indicating improvement, stability, or deterioration in microservice architecture conformance. With small changes, our approach could also be applied, for instance, during early architecture assessment. As future work, we plan to study more decisions, tenets, and related metrics. We also plan to create a larger data set, thus better supporting tasks such as early architecture assessment in a project.

7 Metrics for Assessing Architecture Conformance to Microservice Architecture Patterns and Practices

This chapter builds upon the approach outlined in Chapter 6 by introducing three new architectural design decisions that address important aspects, namely *External API*, *Inter-Service Message Persistence* and *End-to-end Tracing*. By combining these two approaches, it becomes possible to assess whether most of the significant aspects in microservices are being supported or violated.

7.1 Introduction

Microservices architectures [New15, Zim17] structure an application as a collection of autonomous services, modeled around a domain. They share a set of important *tenets* such as development in independent teams, cloud-native technologies and architectures, polyglot technology stacks including polyglot persistence, lightweight containers, loosely coupled service dependencies, high releasability, end-to-end tracing and monitoring, and continuous delivery [Zim17, LF04, New15]. This work examines ways to ensure architecture conformance to these microservice tenets while applying established patterns and practices. That is, many architectural patterns that reflect recommended "best practices" in a microservices context have already been published in the literature [Ric17, ZSZ+19, Sko19]. Conformance to these patterns impacts how far a microservice system supports the desired microservices tenets.

Unfortunately, as real-world, industrial microservice-based systems are usually highly complex, often highly polyglot, and rapidly changed and released (see, e.g. [KH19, Che18]), an automatic or semi-automatic assessment of their pattern conformance is difficult: real-world systems feature various combinations of these patterns and different degrees of violations of the same. Different technologies in various parts of the system implement the patterns in different ways, and these implementations are continuously changing at a high pace. Making matters even more challenging, a high level of automation is required for complex systems. While for small-scale systems of a few services, a manual assessment by an expert is probably as quick and as accurate as an automated one, that is not true for industrial-scale systems of several hundred or more services, which are being developed by different teams or companies, evolving at different paces. In that case, manual assessment is laborious and inaccurate, and a more automated method would vastly improve cost-effectiveness. Another major challenge is that no microservice system can support all microservice tenets well at once. Rather the *architectural decisions* for or against a set of related patterns and practices need to make a trade-off among the desired tenets and important other quality attributes [HWB17, Zim17]. Under these considerations, this work aims to study the following research questions:

- **RQ3.4** How can conformance to the tenets embodied in microservice architecture decision options (i.e. patterns and practices) be automatically assessed?
- **RQ3.5** How well do measures for assessing decision options and their associated tenets perform?
- **RQ3.6** What is a set of minimal elements needed in a microservice architecture model to compute such measures?

Our approach to address these challenges is to define a set of metrics for each microservice decision associated to the decision's options, i.e. at least one metric per major decision option. Based on a manual assessment of a small set of models and model variants that is representative for the possible decision options and option combinations of the studied decisions, we derive a ground truth. The ground truth is established by objectively assessing whether each decision option is supported. By combining the outcome of all options of a decision, we can then derive an ordinal assessment of how well the decision is supported in each model. We then use the ground truth data to assess how well the hypothesized metrics can possibly predict the ground truth data by performing an ordinal regression analysis. We propose an architectural component model based approach which uses only modeling elements that can be derived from the system's source code. For this reason, it is important to be able to work with a minimal set of modeling elements, else it might be difficult to continuously parse them from the source code.

To study the research questions we selected and modeled three major decisions, which represent important aspects in architecting microservices. To illustrate our approach we selected by purpose very different aspects of microservices architecture, in particular: the decision for an external API, message persistence, and end-to-end tracing. For each of these we hypothesized a number of generic, technology-independent metrics to measure conformance to the respective decisions. For the evaluation of these metrics, we modeled 24 architecture models taken from the practitioner literature and assessed each of them manually regarding its support of the patterns and practices contained in each decision. We then compared the results in depth and statistically over the whole evaluation model set. The results show that a subset of each decision related metrics are quite close to the manual, pattern-based assessment.

This chapter is structured as follows: Section 7.2 compares to related work. In Section 7.3 we explain the decisions considered in this chapter and the related patterns/practices. Next, we describe the research methods and the tools we have applied in our study in Section 7.4. In Section 7.5 we report how the ground truth data for each decision is calculated. Section 7.6 introduces our hypothesized metrics. Section 7.7 describes the metrics calculations results for our models and the results of the ordinal regression analysis. Section 7.8 discusses the RQs regarding the evaluation results and analyses the threats to validity. Finally, in Section 7.9 we draw cocnclusions and discuss future work.

7.2 Related Work

Much research has been conducted in collecting and systematizing microservice patterns. For instance, Richardson [Ric17] collected microservice patterns related to major design and architectural practices. Zimmermann et al. [ZSZ+19] introduce microservice API related patterns.

Skowronski [Sko19] collected best practices for event-driven microservice architectures. Microservice fundamentals and best practices are also discussed by Fowler and Lewis [LF04], and are summarized in a mapping study by Pahl and Jamshidi [PJ16]. Taibi and Lenarduzzi [TL18] study microservice bad smells, i.e. practices that should be avoided (which would correspond to metrics violations in our work).

Many of the works on service metrics today are focused on runtime properties (see e.g. [PRG18]). A number of studies has used metrics to assess microservice-based software architectures, e.g. [PW09, ZNL17, BWZ17], but each is focused on narrow sets of architecture-relevant tenets (e.g. loose coupling), and no general approach for an assessment across different microservice tenets exists. Pautasso and Wilde [PW09] propose a composite, facet-based metric for the assessment of loose coupling in service-oriented systems. Zdun et al. [ZNL17] study the independent deployment of microservices by defining metrics to assess architecture conformance to microservice patterns, focused on two aspects: independent deployment and shared dependencies of services. Bogner et al. [BWZ17] propose a maintainability quality model which combines eleven easily extracted code metrics into a broader quality assessment. Engel et al. [ELBH18] also propose a method of using real-time system communication traces to extract metrics on conformance to recommended microservice design principles such as loose coupling and small service size.

These studies focus on treating microservice architectures as a question of components and connectors, factoring in the technologies used, and producing assessments that combine different assessment parameters (i.e. metrics). Such metrics, if automatically collected, can be utilized as part of larger assessment models/frameworks during design and development time. Our work broadly follows the same approach, but extends it to different architecture tenets relevant to microservice-specific design decisions. Once metrics can be checked automatically, our approach can be classified as a metrics-based, microservice-specific approach for software architecture conformance checking. In general, approaches for architecture conformance checking are often based on automated extraction techniques [GAK99, VDHK+04]. Techniques that are based on a broad set of microservice-related metrics to cover multiple microservice tenets do not yet exist.

7.3 Background

External API Decision. One central decision in microservice-based systems is how the external API is offered to clients. This is tightly coupled to the loose coupling, releasability, independent development and deployment, and continuous delivery tenets, as it determines the coupling between client and internal system concerns. In some service-based systems, the *clients can call into system services directly*, meaning high coupling and thus difficulties in releasing, developing, and deploying the clients and system services independently of each other. A better decoupling level might be reached through an *API Gateway* [Ric17], a pattern that describes a common entry point for the system through which all requests are routed. It is a specialized variant of a *Reverse Proxy*, which covers only the routing aspects of an *API Gateway* but not further API abstractions such as authentication, rate limiting, and so on (see [ZSZ+19]). A variant of *API Gateway* for servicing different types of clients (e.g., mobile and desktop clients) is the *Backends for Frontends* pattern [Ric17], which offers a fine-grained API for each specific type of

client. A variant where clients can call into system services directly, but are still decoupled is *API Composition* [Ric17], i.e. a service which can invoke other microservices and provides an API for the connected services.

Inter-Service Message Persistence Decision. In many business-critical microservice systems, an important concern is that no messages get lost. This concern directly influences the communication between services, and, depending on which option is chosen, the coupling between services, their releasability, their independent development and deployment, as well as their continuous delivery are impacted. Many systems choose communication means that offer *no inter-service message persistence*. Some patterns better support the related aspects of the microservice tenets: The *Messaging* pattern [HW03a] describes service communication, in which persistent message queuing is used to store a producer's messages until the consumer receives them. Many *Stream Processing* [Sko19] components (e.g. Apache Kafka) offer a very similar message persistence level. These solutions offer optimal inter-service message persistence, in the sense that the technology is designed for providing support for it. Some other solutions applied in the microservice field can be used (or adapted) to support it: *Interaction through a Shared Database*, even though frowned upon with regard to other microservice tenet aspects, supports some level of message persistence as well, but not the automated support of *Messaging*. A more microservice-style technique that supports this level of database-based persistence is the combination of the *Outbox* and the *Transaction Log Tailing* patterns [Ric17] in which each service that sends messages has an outbox database table. As part of the database transaction, the service sends messages by inserting them into the outbox table. A message relay component reads the outbox table and publishes the messages to a message broker. Using the *Event Sourcing* pattern [Ric17] every change to the state of the system should be contained in an event object and stored sequentially in order to be accessible over time. The events are persisted in an event store. This way at least a temporary message persistence is achieved.

End-to-end Tracing Decision. Logging and monitoring are standard practices for creating observability of microservices. As microservice architectures are used for highly distributed and polyglot systems with complex interactions, many of them go one step further and realize end-to-end tracing. It supports tracing and monitoring tenets directly, as well as understandability concerns during independent development and deployment, mastering complexity of highly decoupled services, and thus indirectly releasability and continuous delivery. Like in the other decisions, one option is to offer *No Tracing Support*. In contrast, *Distributed Tracing* [Ric17] is a method used to profile and monitor applications through recording traces on the distributed components. It can either be supported on the microservices of a system, on the gateways of a system, or on both. If both support *Distributed Tracing*, this is optimal, as all relevant traces in ingress, egress, and inter-service communication can be recorded. If it is not supported, a lower level of tracing and monitoring can be reached by routing the service communication through a central component, such as a *Publish/Subscribe* or *Message Broker* component [HW03a]. This can also be achieved if all internal inter-service communication is routed through the *API Gateway*, or if *Event Sourcing* or *Event Logging* [Ric17, Sko19] are used, which store all events temporarily. None of the later techniques has the same level of support as *Distributed Tracing*,

but all of them can – with some programming or manual effort – be used to reconstruct traces.

7.4 Research and Modeling Methods

7.4.1 Model Selection Methods

This study focuses on architecture conformance to microservice patterns and practices. To be able to study this, we first performed an iterative study of a variety of microservice-related knowledge sources, and we refined a meta-model which contains all the required elements to help us reconstruct existing microservice-based systems. For problem investigation and as an evaluation model set for eventually creating a ground truth for our study, we have gathered a number of microservice-based systems, summarized in Table 3.1. Each of them is either taken directly from a system published by practitioners (on GitHub and/or practitioner blogs) or a system variant adapted according to discussions in the relevant literature. The systems were taken from 9 independent sources. They were developed by practitioners with microservice experience, and they provide a good representation of the microservices best practices summarized in Section 7.3. We performed a fully manual static code analysis for those models where the source code was available (i.e. 7 of our 9 sources; two were modeled based on documentation created by the practitioners). The result is a set of precisely modeled component models of the software systems (modeled using the techniques described below). Variations were modeled to cover the complete design space of our three decisions described in Section 7.3, according to the referenced practitioner sources. Apart from the variations described in Table 3.1 all other system aspects remained the same as in the base models. This resulted in a total of 24 models summarized in Table 3.1. We assume that our evaluation models are close to models used in practice and real-world practical needs for microservices. As many of them are open source systems with the purpose of demonstrating practices, they are at most of medium size, though.

7.4.2 Metrics Definition, Ground Truth Calculation, and Statistical Evaluation Methods

To measure conformance to the respective patterns and practices in the design decisions from Section 7.3, we defined a set of metrics for each microservice decision associated to the decision's options, i.e. at least one metric per major decision option. Based on the manual assessment of the models from Table 3.1, we derived a ground truth for our study (the ground truth and its calculation rules are described in Section 7.5). The ground truth is established by objectively assessing whether each decision option is supported, partially supported, or not supported. By combining the outcome of all options of a decision, we then derived an ordinal assessment on how well the decision is supported in each model, using the scale: [$++$: very well supported, $+$: well supported, o: neutral, $-$: badly supported, $--$: very badly supported]. Our scale does not assume equal distances (i.e. it is not a Likert scale), but it assumes the given order. We then used the ground truth data to assess how well the hypothesized metrics can possibly predict the ground truth data by performing an ordinal regression analysis.

Ordinal regression is a widely used method for modeling an ordinal response's dependence on a set of independent predictors. For the ordinal regression analysis we used the *lrm* function from

the *rms* package in R [FEH15].

7.4.3 Methods for Modeling Microservice Component Architectures

From an abstract point of view, a microservice-based system is composed of components and connectors with a set of component types and a set of connector types. Our work has the goal to automate metrics calculation and assessment based on the component model of a microservice system. That is, if the system is manually modeled or the model can be derived automatically from the source code, our approach is applicable. For modeling microservice architectures we followed the method reported in our previous work [ZNL17]. All the code and models used in and produced as part of this study have been made available online for reproducibility [1].

7.5 Ground Truth Calculations for the Study

In this section, we report for each of the decisions from Section 7.3 how the ground truth data is calculated based on manual assessment whether each of the relevant patterns is either Supported (**S** in Table 7.1), Partially Supported (**P** in Table 7.1), or Not-Supported (**N** in Table 7.1). The ordinal results of those assessments are then reported in the Assessments rows of Table 7.1.

External API																								
	BM1	**BM2**	**BM3**	**CO1**	**CO2**	**CO3**	**CI1**	**CI2**	**CI3**	**CI4**	**EC1**	**EC2**	**EC3**	**ES1**	**ES2**	**ES3**	**FM1**	**FM2**	**HM1**	**HM2**	**RM**	**RS**	**TH1**	**TH2**
Reverse Proxy	S	S	S	N	S	N	S	S	N	P	P	P	P	S	S	S	N	N	N	N	S	S	P	P
API Gateway	S	S	S	N	S	N	S	S	N	P	P	P	P	S	S	S	N	N	N	N	S	S	P	P
Backends for Frontends	N	N	N	N	N	N	N	N	N	N	N	N	N	S	S	S	N	N	N	N	N	N	N	N
API Composition	N	N	N	N	N	N	N	N	P	P	N	N	N	N	N	N	N	S	S	S	N	N	P	P
Assessments	++	++	++	--	++	--	++	++	-	o	o	o	o	++	++	++	--	+	+	+	++	++	o	o
Persistent Messaging for Inter-Service Communication																								
	BM1	**BM2**	**BM3**	**CO1**	**CO2**	**CO3**	**CI1**	**CI2**	**CI3**	**CI4**	**EC1**	**EC2**	**EC3**	**ES1**	**ES2**	**ES3**	**FM1**	**FM2**	**HM1**	**HM2**	**RM**	**RS**	**TH1**	**TH2**
Messaging or Persistent PubSub	N	N	N	N	S	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	S	P	N	N
Shared Database Interaction	N	N	N	N	N	S	N	N	N	N	N	N	S	N	N	N	N	N	N	N	N	N	N	N
Outbox and Trans. Log Tailing	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Event Sourcing	S	N	N	N	N	N	N	N	N	N	N	S	N	N	N	N	N	N	N	P	N	N	N	P
All Service Comm. Persistent	S	N	N	N	S	S	N	N	N	N	N	S	N	N	N	N	N	N	N	N	S	N	N	P
Assessments	+	--	--	--	++	+	--	--	--	--	--	+	-	--	--	--	--	--	--	-	++	o	--	o
End-to-End Tracing																								
	BM1	**BM2**	**BM3**	**CO1**	**CO2**	**CO3**	**CI1**	**CI2**	**CI3**	**CI4**	**EC1**	**EC2**	**EC3**	**ES1**	**ES2**	**ES3**	**FM1**	**FM2**	**HM1**	**HM2**	**RM**	**RS**	**TH1**	**TH2**
Distributed Tracing on Services	N	N	N	N	S	S	N	N	N	N	N	N	N	N	N	N	N	S	P	S	P	P	N	N
Distributed Tracing on Gatew.	N	N	N	N	S	N	N	N	N	N	N	N	N	N	N	N	N	N	S	S	N	N	N	N
Pub/Sub, Messaging	S	N	N	N	S	N	N	N	N	N	N	S	N	P	N	N	N	N	N	P	S	P	N	S
Inter-service comm. via Gatew.	N	S	S	N	N	N	P	S	N	P	S	N	N	N	P	P	N	N	N	N	N	N	N	N
Event Sourcing/Logging	S	N	N	N	N	N	N	N	N	N	N	S	N	N	N	N	N	N	N	P	S	N	N	S
Assessments	o	-	-	--	++	++	--	-	--	--	-	o	--	--	--	--	--	+	+	++	o	o	--	o

Table 7.1: Ground Truth Data

Following the argumentation, which decision option explained in Section 7.3 has which impact on the *External API Decision* related tenets, we can derive the following scoring scheme for our ground truth assessment of this decision:

[1] https://doi.org/10.5281/zenodo.3999477

- **++**: All client traffic is routed through an *API Gateway* or *Backends for Frontends*.
- **+**: All client-connected services provide *API Composition* or only *Reverse Proxy* capabilities.
- **o**: Some client traffic is routed through *API Gateway* or *Backends for Frontends*.
- **-**: Some client-connected services provide *API Composition* or only *Reverse Proxy* capabilities.
- **--**: All client traffic is directly connected to backend services and no *API Composition* happens.

From the argumentation for the *Inter-service Message Persistence Decision*, we can derive the following scoring scheme for our ground truth assessment:

- **++**: *Message Brokers* or a persistent *Publish/Subscribe* or *Stream Processing* component are used for all inter-service communication.
- **+**: All interservice communication is persisted by some combination of partial *Message Brokers*, persistent *Publish/Subscribe*, or persistent *Stream Processing* or partial or full coverage with *Shared Database*, *Event Sourcing*, *Outbox/Transaction Log Tailing*.
- **o**: A part of the interservice communication is persisted by partial coverage with *Message Brokers*, persistent *Publish/Subscribe*, or persistent *Stream Processing*.
- **-**: A part of the interservice communication is persisted by partial coverage with *Shared Database*, *Event Sourcing*, *Outbox/Transaction Log Tailing*.
- **--**: None of the above is supported.

Finally, from the argumentation for the*End-to-end Tracing Decision*, we can derive the following scoring scheme for our ground truth assessment:

- **++**: *Distributed Tracing* is fully supported on all services and gateways.
- **+**: *Distributed Tracing* is fully supported on either the services or the gateways.
- **o**: *Distributed Tracing* is partially supported or *Event Sourcing/Event Logging* are fully supported.
- **-**: *Publish/Subscribe*, *Message Broker*, or *Invocations Routed Via API Gateway* are fully supported for service interactions or those patterns are partially supported and at the same time *Event Sourcing/Event Logging* are supported.
- **--**: None of the above is supported.

7.6 Metrics

All metrics, unless otherwise noted, are a continuous value with range from 0 to 1, with 1 representing the optimal case where a set of patterns is fully supported, and 0 the worst-case scenario where it is completely absent. For instance, in EC1 client traffic is partially routed through API Gateway resulting CCF = 0.25. The metrics results for each model per decision metric are presented in Table 7.2.

7.6.1 Metrics for the External API Decisions

Client-side Communication via Facade utilization metric (CCF). This metric returns the number of the connectors from *Clients* to *Facade* components set in relation to the total number of unique *Client* connectors. This way, we can measure how many unique client links are using the External API used by one of the Facade components (i.e. offered through patterns such as *API Gateway*, *Reverse Proxy*, *Backends for Frontends*).

$$CCF = \frac{\textit{Number of Client to Facade Links}}{\textit{Number of Unique Client Links}}$$

In this metric (and in other metrics below), the number of unique client links is defined as follows:

$$\begin{aligned} &\textit{Number of Unique Client Links} = \\ &max\{\textit{Number of Facades Linked to Clients}, \\ &\textit{Number of Clients Linked to Facades}\} \\ &+ \textit{Number of Client to Non-Facade/Non-Client Links} \end{aligned}$$

As a result, the only decision option remaining is *API Composition*, for which we formulated the APIC metric.

API Composition utilization metric (APIC). In cases that a client is directly connected to services, it is possible that these services offer an *External API* shielding the interfaces of other services that are connected to them. That is, a client can have access to a system service via other services. To detect such cases, we count the routes from the client to system services via other services and set this number in relation to the total number of system services. That gives us the proportion of services that are accessible by clients via other services. We then divide this number with the unique client links to estimate the proportion of clients connected services which are possibly composing an *External API* using *API Composition*.

$$APIC = \frac{\dfrac{\textit{Number of Client to Services via other Services Routes}}{\textit{Total Number of Services}}}{\textit{Number of Unique Client Links}}$$

7.6.2 Metrics for Persistent Messaging for Inter-Service Communication Decision

Service Messaging Persistence utilization metric (SMP). One important aspect in services interconnections is the persistence of the exchanged messages. We defined this metric to measure the proportion of the services interconnections that are made persistent through supporting technology (i.e. *Messaging* or *Stream Processing*).

$$SMP = \frac{\textit{Service Interconnections with Messaging or Stream Processing}}{\textit{Number of Service Interconnections}}$$

Shared DataBase utilization metric (SDB). Although a *Shared Database* is considered as an anti-pattern in microservices, there are many systems that use it either partially or completely. The pattern might be beneficial for persistent messaging, but definitely is not the optimal option. To measure its presence in a system, we count the number of interconnections via a *Shared Database* compared to the total number of interconnections. We note that for this metric, our metrics scale is reversed in comparison to the other metrics, because here we detect the presence of an anti-pattern: the optimal result of our metrics is 0, and 1 is the worst-case result.

$$SDB = \frac{\textit{Service Interconnections with SharedDB}}{\textit{Number of Service Interconnections}}$$

Outbox/Event Sourcing utilization metric (OES). *Outbox* and *Event Souring* can ensure temporary message persistence. Our metric measures the proportion of the interconnections with *Outbox/Event Sourcing* to the total number of interconnections.

$$OES = \frac{\textit{Service Interconnection with Outbox or Event Sourcing}}{\textit{Number of Service Interconnections}}$$

7.6.3 Metrics for End-to-End Tracing Decision

$$SFT = \frac{\textit{Services and Facades Support Distributed Tracing}}{\textit{Number of Services and Facades}}$$

Service Interaction via Central Component utilization metric (SICC) and **Service Interaction with Event Sourcing utilization metric (SIES)**. *Distributed Tracing* can be supported by routing the inter-service communication via a central component (e.g. *Publish/Subscribe, Message Broker* and *API Gateway*). Since *Event Sourcing* also enables tracing by tracking the messages, we distinguish between systems that support *Event Sourcing* (SIES), and systems that do not (SICC).

$$SICC = \frac{\textit{Service Interaction via Central Component w/o Event Sourcing}}{\textit{Number of Service Interconnections}}$$

$$SIES = \frac{\textit{Service Interaction via Central Component with Event Sourcing}}{\textit{Number of Service Interconnections}}$$

7.7 Ordinal Regression Analysis Results

The metrics calculations for each model per each decision metric are presented in Table 7.2. The dependent outcome variables are the ground truth assessments for each decision, as described in Section 7.5 and summarized in Table 7.1. The metrics defined in Section 7.6 are used as the independent predictor variables. The ground truth assessments are ordinal variables, while all the independent variables are measured on a scale from 0.0 to 1.0. The aim of the analysis is to predict the likelihood of the dependent outcome variable for each of the decisions by using the relevant metrics.

Each resulting regression model consists of a *baseline intercept* and the independent variables multiplied by *coefficients*. There are different intercepts for each of the value transitions of the dependent variable ($\geq$*Badly Supported*, $\geq$*Neutral*, $\geq$*Well Supported*, $\geq$*Very Well Supported*), while the coefficients reflect the impact of each independent variable on the outcome. For example, a positive coefficient, such as +5, indicates a corresponding five-fold increase in the dependent variable for each unit of increase in the independent variable; conversely, a coefficient of -30 would indicate a thirty-fold decrease.

In Table 7.3, we report the p-values for the resulting models, which in all cases are very low, indicating that the sets of metrics we have defined are able to predict the ground truth assessment for each decision with a high level of accuracy.

7.8 Discussion

7.8.1 Discussion of Research Questions

For answering **RQ3.4** and **RQ3.5**, we suggested a set of generic, technology-independent metrics for each microservice decision, and we associated at least one metric to each major decision option. The ground truth is established by objectively assessing how well a pattern and/or practice is supported in each model, and extrapolating this to how well the broader decision is supported. We formulated metrics to assess a pattern's implementation in each model, and performed an ordinal regression analysis using these metrics as independent variables to predict the ground truth assessment. Our results show that every set of decision-related metrics can predict with high accuracy our objectively evaluated assessment. This suggests that automatic metrics-based assessment of a system's conformance to the tenets embodied in each design decision is possible with a high degree of confidence.

Regarding **RQ3.6**, we can assess that our microservice meta-model has no need for major extensions and is easy to map to existing modeling practices. More specifically, in order to fully model our evaluation model set, we needed to introduce 25 component types and 38 connector types, ranging from general notions such as the *Service* component type, to very technology-specific classes such as the *RESTful HTTP* connector, which is a subclass of *Service Connector*. Our study shows that for each pattern and practice embodied in each decision and the proposed metrics, only a small subset of the meta-model is required. The decision *External API* requires to model at least the *Service*, *Client*, and the *Facade* component types and the technology-related connector types (e.g. *RESTful HTTP, Synchronous Connector, HTTP, HTTPS*). The *Persistent Messaging for Inter-Service Communication* and *End-to-End Tracing* decisions need a number of additional components (e.g. *Event Sourcing, Stream Processing, Messaging, PubSub*) and the respective connectors (e.g. *Publisher, Subscriber, Message Consumer* and *Messages Producer*) to be modeled.

7.8.2 Threats to Validity

We deliberately relied on third-party systems as the basis for our study to increase internal validity, thus avoiding bias in system composition and structure. It is possible that our search procedures introduced some kind of unconscious exclusion of certain sources; we mitigated this by assembling an author team with many years of experience in the field, and performing very general and broad searches. Given that our search was not exhaustive, and that most of the systems we found were made for demonstration purposes, i.e. relatively modestly sized, this means that some potential architecture elements were not included in our meta-model. In addition, this raises a possible threat to external validity of generalization to other, and more complex, systems. We nevertheless feel confident that the systems documented are a representative cross-cut of current practices in the field, as the points of variance between them were limited and well attested in the literature. Another potential threat is the fact that the variant systems were derived by the author team. However, this was done according to best practices documented in literature. We made sure only to change specific aspects in a variant and keep all other aspects stable.

Another potential source of internal validity threat is the modeling process itself. The author team has considerable experience in similar methods, and the models of the systems were repeatedly and independently cross-checked, but the possibility of some interpretative bias remains: other researchers might have coded or modeled differently, leading to different models. As our goal was only to find one model that is able to specify all observed phenomena, and this was achieved, we consider this threat not to be a major issue for our study. The ground truth assessment might also be subject to different interpretations by different practitioners. For this purpose, we deliberately chose only a three-step ordinal scale, and given that the ground truth evaluation for each decision is fairly straightforward and based on best practices, we do not consider our interpretation controversial. Likewise, the individual metrics used to evaluate the presence of each pattern were deliberately kept as simple as possible, so as to avoid false positives and enable a technology-independent assessment. As stated previously, generalization to more complex systems might not be possible without modification. But we consider that the basic approach taken when defining the metrics is validated by the success of the regression models.

7.9 Conclusions and Future Work

In this chapter we have hypothesized that it is possible to develop a method to automatically assess microservices tenets in microservice decisions based on a microservice system's component model. We have shown that this is possible for microservice decision models comprising patterns and practices as decision options. Our approach first modeled the key aspects of the decision options using a minimal set of component model elements (which could be automatically extracted from the source code). Then we derived at least one metric per decision option and used a small reference model set as a ground truth. We then used ordinal regression analysis for deriving a predictor model for the ordinal variable. Our statistical analysis shows a high level of accuracy.

While so far many studies on metrics for component model and other architectures exist, the specifics of microservice architectures and their particular tenets have not been studied. As discussed in Section 7.2, only using general metrics does not help much in assessing microservice architectures. Our approach is one of the first that studies a metrics-based assessment of multiple, very different microservice tenets. Our main goal is a continuous assessment, i.e. we envision an impact on continuous delivery practices, in which the metrics are assessed with each delivery pipeline run, indicating improvements, stability, or deteriorations in microservice architecture conformance. With small changes, our approach could also be applied, during early architecture assessment.

As future work, we plan to study more decisions, tenets, and related metrics. We also plan to create a larger data set, thus better supporting tasks such as early architecture assessment in a project.

Table 7.2: Metrics Calculation Results

Metrics	BM1	BM2	BM3	CO1	CO2	CO3	CI1	CI2	CI3	CI4	EC1	EC2
External API												
CCF	1.00	1.00	1.00	0.00	1.00	0.00	1.00	1.00	0.00	0.50	0.25	0.25
APIC	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.30	0.10	0.00	0.00
Persistent Messaging for Inter-Service Communication												
SMP	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SDB	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
OES	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00
End-to-End Tracing												
SFT	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
SICC	0.00	1.00	1.00	0.00	1.00	1.00	0.14	1.00	0.00	0.60	1.00	0.00
SIES	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00

Metrics	EC3	ES1	ES2	ES3	FM1	FM2	HM1	HM2	RM	RS	TH1	TH2
External API												
CCF	0.25	1.00	1.00	1.00	0.00	0.00	0.00	0.00	1.00	1.00	0.25	0.25
APIC	0.00	0.00	0.00	0.00	0.25	0.50	0.70	0.70	0.00	0.00	0.12	0.04
Persistent Messaging for Inter-Service Communication												
SMP	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.66	0.11	0.00	0.00
SDB	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
OES	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.08	0.00	0.00	0.00	0.66
End-to-End Tracing												
SFT	0.00	0.00	0.00	0.00	0.00	1.00	0.90	0.90	0.14	0.62	0.00	0.00
SICC	0.00	0.60	0.45	0.45	0.00	0.00	0.00	0.00	1.00	0.11	0.00	0.00
SIES	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.80	0.00	0.00	0.00	0.66

Table 7.3: Regression Analysis Results

Intercepts/Coefficients	***Value***	***Model p-value***
External API		
Intercept (≥Badly Supported)	*-3.5690*	*4.423828e-11*
Intercept (≥Neutral)	*-4.5042*	
Intercept (≥Well Supported)	*-10.2692*	
Intercept (≥Very Well Supported)	*-15.7271*	
Metric Coefficient (CCF)	*20.3552*	
Metric Coefficient (APIC)	*18.1419*	
Persistent Messaging for Inter-Service Communication		
Intercept (≥Badly Supported)	*-5.6344*	*2.002198e-09*
Intercept (≥Neutral)	*-9.5937*	
Intercept (≥Well Supported)	*-11.2074*	
Intercept (≥Very Well Supported)	*-21.0398*	
Metric Coefficient (SMP)	*94.5503*	
Metric Coefficient (SDB)	*10.4199*	
Metric Coefficient (OES)	*13.3840*	
End-to-End Tracing		
Intercept (≥Badly Supported)	*-35.4940*	*4.440892e-15*
Intercept (≥Neutral)	*-53.7947*	
Intercept (≥Well Supported)	*-103.6085*	
Intercept (≥Very Well Supported)	*-135.5906*	
Metric Coefficient (SFT)	*44.6971*	
Metric Coefficient (SICC)	*94.1809*	
Metric Coefficient (SIES)	*125.5634*	

8 Evaluating Architecture Conformance to Coupling-Related Infrastructure-as-Code Best Practices

The method used in Chapters 6 and 7 was replicated in this chapter to examine architectural design decisions related to coupling in IaC-based deployments. The findings comprise 8 metrics that evaluate the support for coupling in deployment strategy and infrastructure stack grouping.

8.1 Introduction

Today, many microservice-based systems are being rapidly released, resulting in frequent changes not only in the system implementation but also in its infrastructure and deployment [HF10, Nyg07]. Furthermore, the number of infrastructure nodes that a system requires is increasing significantly [Nyg07] and the managing and structuring of these elements can have a significant impact on the development and deployment processes. *Infrastructure as Code* enables automating the provisioning and management of the infrastructure nodes through reusable scripts, rather than through manual processes [Mor15]. IaC can ensure that a provisioned environment remains the same every time it is deployed in the same configuration, and configuration files contain infrastructure specifications making the process of editing and distributing configurations easier [Mor15, ABDN+17]. IaC can also contribute to improving consistency and ensuring loose coupling by separating the deployment artifacts according to the services' and teams' responsibilities. The deployment infrastructure can be structured using infrastructure stacks. An infrastructure stack is a collection of infrastructure elements/resources that are defined, provisioned, and updated as a unit [Mor15]. A wrong structure can result in severe issues if coupling-related aspects are not considered. For instance, defining all the system deployment artifacts as only one unit in one infrastructure stack can significantly impact the dependencies of system parts and teams as well as the independent deployability of system services. Most of the established practices in the industry are mainly reported in the so-called "grey literature," consisting of practitioner blogs, system documentation, etc. The architectural knowledge is scattered across many knowledge sources that are usually based on personal experiences, inconsistent, and incomplete. This creates considerable uncertainty and risk in architecting microservice deployments.

We investigate such IaC-based best practices in microservice deployments. In this context, we formulate a number of coupling-related *Architectural Design Decisions (ADDs)* with corresponding decision options. In particular, the ADDs focus on *System Coupling through Deployment Strategy* and *System Coupling through Infrastructure Stack Grouping*. For each of these, we define a number of generic, technology-independent metrics to measure the conformance of a given deployment model to the (chosen) ADD options. Based on this architectural knowledge,

our goal is to provide an automatic assessment of architecture conformance to these practices in IaC deployment models. We also aim for a continuous assessment, i.e., we envision an impact on continuous delivery practices, in which the metrics are assessed with each delivery pipeline run, indicating improvement, stability, or deterioration in microservice deployments. In order to validate the applicability of our approach and the performance of the metrics, we conducted three case studies on open source microservice-based systems that also include the IaC-related scripts. The results show that our set of metrics is able to measure the support of patterns and practices.

This work aims to answer the following research questions:

- **RQ3.7** How can we measure conformance to coupling-related IaC best practices in the context of IaC architecture decision options?
- **RQ3.8** What is a set of minimal elements needed in an IaC-based deployment and microservice architecture model to compute such measures?

This chapter is structured as follows: Section 8.2 discusses related work. Next, we describe the research methods and the tools we have applied in our study in Section 8.3. In Section 8.4 we explain the ADDs and the related patterns and practices. Section 8.5 introduces our metrics in a formal model. Then, three case studies are explained in Section 8.6. Section 8.7 discusses the RQs regarding the evaluation results and analyses the threats to validity of our study. Finally, in Section 8.8 we draw conclusions and discuss future work.

8.2 Related Work

Several existing works target collecting IaC bad and best practices. For instance, Sharma et al. [SFS16] present a catalog of design and implementation language-specific smells for Puppet. A broad catalog of language-agnostic and language-specific best and bad practices related to implementation issues, design issues, and violations of essential IaC principles is presented by Kumara et al. [KGR+21]. Schwarz et al. [SSL18] offer a catalog of smells for Chef. Morris [Mor15] presents a collection of guidance on managing IaC. In his book, there is a detailed description of technologies related to IaC-based practices and a broad catalog of patterns and practices. Our work also follows IaC-specific recommendations given by Morris [Mor15], as well as those more microservice-oriented given by Richardson [Ric17]. We indeed build on their guidelines and catalogs of bad/best practices to support architecting the deployment of microservices, while also enabling us to assess and improve the quality of obtained IaC deployment models.

In this perspective, it is worth relating our proposal with existing tools and metrics for assessing and improving the quality of IaC deployment models. Dalla Palma et al. [DDPT20, DDT20] suggest a catalog of 46 quality metrics focusing on Ansible scripts to identify IaC-related properties and show how to use them in analyzing IaC scripts. A tool-based approach for detecting smells in TOSCA models is proposed by Kumara et al. [KVM+20]. Sotiropoulos et al. [SMS20] provide a tool to identify dependency-related issues by analyzing Puppet manifests and their system call trace. Van der Bent et al. [vdBHVG18] define metrics reflecting IaC best practices to assess Puppet code quality. All such approaches focus on the use of different

metrics to assess and improve the quality of IaC deployment models, showing the potential and effectiveness of metrics in doing so. We hence follow a similar, metrics-based approach but targeting a different aspect than those of the above mentioned approaches, namely *system coupling*. To the best of our knowledge, ours is the first solution considering and tackling such aspects.

Other approaches worth mentioning are those by Fischer et al. [FBKL17] and Krieger et al. [KBKL18], who both allow automatically checking the conformance of declarative deployment models during design time. They both model conformance rules in the form of a pair of deployment model fragments. One of the fragments represents a detector subgraph that determines whether the rule applies to a particular deployment model or not. The comparison of the model fragments with a given deployment model is done by subgraph isomorphism. Unlike our study, this approach is generic and does not introduce specific conformance rules, such as checking coupling-related ADDs in IaC models.

Finally, it is worth mentioning that architecture conformance can also be checked with other techniques such as dependency-structure matrices, source code query languages, and reflexion models as shown by Passos et al. [PTV+10]. So far, methods based on various interrelated IaC-based metrics to check pattern/best practice conformance like ours do not yet exist. Also, none are able to produce assessments that combine different assessment parameters (i.e., metrics). Such metrics, if automatically computed, can be used as a part of larger assessment models during development and deployment time.

8.3 Research and Modeling Methods

Figure 8.1 shows the research steps followed in this study. We first studied knowledge sources related to IaC-specific best practices from practitioner books and blogs, and the scientific literature (such as [KGR+21, Mor15, Ric17, SSL18, SFS16]) as well as open-source repositories (such as the case studies discussed in Section 8.6). We then analyzed the data collected using qualitative methods based on Grounded Theory [CS90] coding methods, such as open and axial coding, and extracted the two core IaC decisions described in Section 8.4 along with their corresponding decision drivers and impacts. We followed the widely used pattern catalogs by Morris [Mor15] and Richardson [Ric17] closely to obtain the necessary information since both are well documented, detailed, and address many relevant concerns in the IaC domain. Among the many design decisions, covered in these catalogs, we selected those that are directly connected to IaC practices, operate at a high abstraction level (i.e., they are "architectural" design decisions), and are related to architectural coupling issues. We then defined a set of metrics for automatically computing conformance to each coupling-related pattern or practice per decision described in Section 8.4. We studied and modeled three case studies following the *Model Generation* process. Finally, we evaluated our set of metrics using the case studies. Furthermore, in our work [NZP+21], we have introduced a set of detectors to automatically reconstruct an architecture component model from the source code. Combining the automatic reconstruction with the automatic computation of metrics, the evaluation process can be fully automated.

The systems we use as case studies were developed by practitioners with relevant experience and are supported by the companies Microsoft, Instana, and Weaveworks as microservice ref-

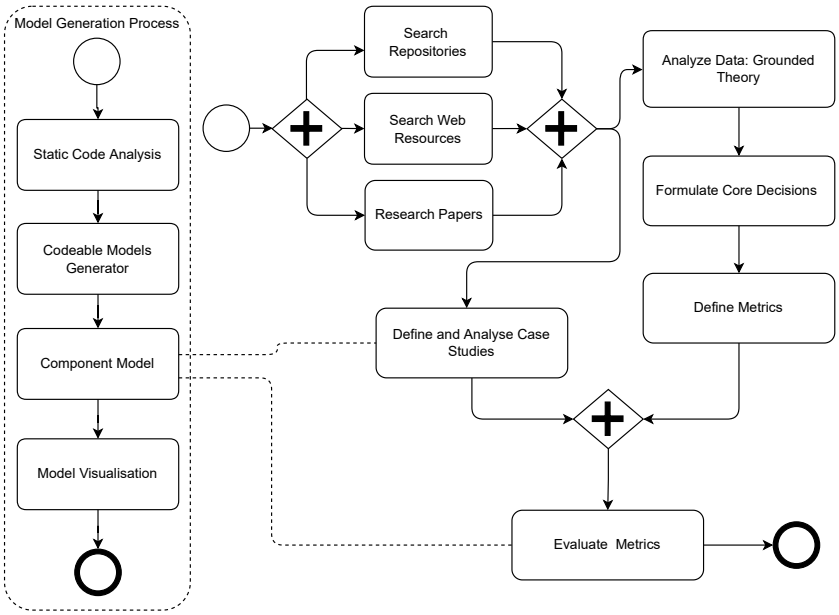

Figure 8.1: Overview of the research method followed in this study

erence applications, which justifies the assumption that they provide a good representation of recent microservice and IaC practices. We performed a fully manual static code analysis for the IaC models that are in the repositories together with the application source code. To create our models, we used the Python modeling library CodeableModels described in Chapter 3.7. The result is a set of decomposition models of the software systems along with their deployments (see Section 8.5.1). The code and models used in and produced as part of this study have been made available online for reproducibility [1].

Figure **??** shows an excerpt of the resulting model of *Case Study 1* in Section 8.6. The model contains elements from both application (e.g., *Service, Database*) and infrastructure (e.g., *Container, Infrastructure Stack, Storage Resources*). Furthermore, we have specified all the deployment-related relationships between these elements. In particular, a *Service* and a *Web Server* are *deployed on* a *Container*. A *Database* is *deployed on Storage Resources* and also on a separate *Container*. An *Infrastructure Stack defines deployment of* a *Container* as well as a *Web Server*. All the containers *run on* a *Cloud Server* (e.g., ELK, AWS, etc.).

8.4 Decisions on Coupling-related, IaC-Specific Practices

In this section, we briefly introduce the two coupling-related ADDs along with their decision options. In one decision, we investigate the deployment strategy between services and execution environments, and in the second, we focus on the structure of all deployment artifacts.

[1] https://doi.org/10.5281/zenodo.6696130

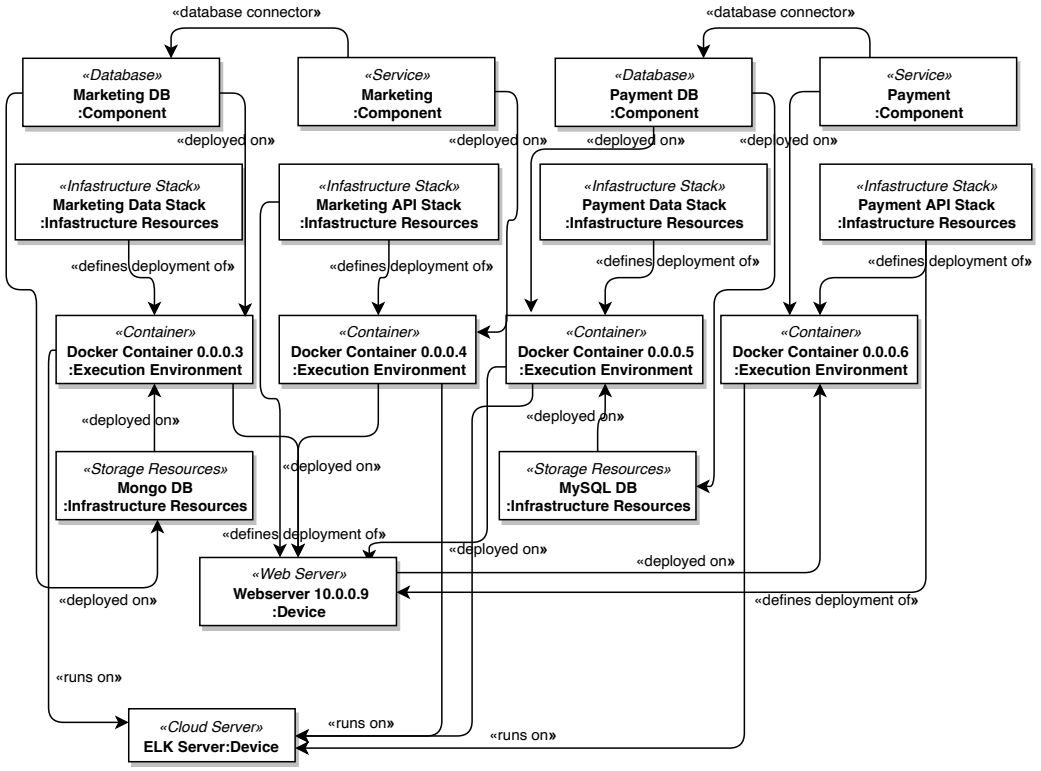

Figure 8.2: Excerpt of the reconstructed model CS1 from Table 8.1

System Coupling through Deployment Strategy Decision. An essential aspect of deploying a microservice-based system is to keep the services independently deployable and scalable and ensure loose coupling in the deployment strategy. Services should be isolated from one another, and the corresponding development teams should be able to build and deploy a service as quickly as possible. Furthermore, resource consumption per service is another factor that should be considered, since some services might have constraints on CPU or memory consumption [Ric17]. Availability and behavior monitoring are additional factors for each independent service that should be ensured in a deployment. One option, which hurts the *loose coupling* of the deployment, is *Multiple Services per Execution Environment* [2] In this pattern, services are all deployed in the same execution environment making it problematic to change, build, and deploy the services independently. A second option for service deployment is *Single Service per Execution Environment* [Ric17]. This option ensures loose coupling in deployment since each service is independently deployable and scalable, and resource consumption can be constrained for each service and monitoring services separately. Development team dependencies are also

[2]The term *Execution Environment* is used here to denote the environment in which a service runs such as a VM, a Container, or a Host. Please note that execution environments can be nested. For instance, a VM can be part of a Production Environment which in turn runs on a Public Cloud Environment. Execution environments run on Devices (e.g., Cloud Server).

reduced. Although *Single Service per Execution Environment* reduces coupling significantly, an incorrect structure of the system artifacts can introduce additional coupling in deployment even if all services are deployed on separate execution environments. The following decision describes in detail the structuring practices.

System Coupling through Infrastructure Stack Grouping Decision. Managing the infrastructure resources can impact significant architectural qualities of a microservice-based system, such as loose coupling between services and independent development in different teams. Grouping of different resources into infrastructure stacks should reflect the development teams' responsibilities to ensure independent deployability and scalability. An infrastructure stack may include different resources such as *Compute Resources* (e.g., VMs, physical servers, containers, etc.) and *Storage Resources* (e.g., block storage (virtual disk volumes), structured data storage, object storage, etc.) [Mor15]. An important decision in infrastructure design is to set the size and structure of a stack. There are several patterns and practices on how to group the infrastructure resources into one or multiple stacks. A pattern that is useful when a system is small and simple is the *Monolith Stack* [Mor15]. This pattern facilitates the process of adding new elements to a system as well as stack management. However, there are some risks to using this pattern. The process of changing a larger monolith stack is riskier than changing a smaller one, resulting in more frequent changes. Also, services cannot be deployed and changed independently and different development teams may be dependent on each other [Mor15]. A similar pattern is the *Application Group Stack* [Mor15]. This kind of stack includes all the services of multiple related applications. This pattern is appropriate when there is a single development team for the infrastructure and deployment of these services and has similar consequences as the *Monolith Stack* pattern.

A structuring that can work better with microservice-based systems is the *Service Stack* [Mor15]. In this pattern, each service has its own stack which makes it easier to manage. Stacks boundaries are aligned to the system and team boundaries. Thus, teams and services are more autonomous and can work independently. Furthermore, services and their infrastructure are loosely coupled since independent deployability and scalability for each service are supported. The pattern *Micro Stack* [Mor15] goes one step further by breaking the *Service Stack* into even smaller pieces and creates stacks for each infrastructure element in a service (e.g., router, server, database, etc.). This is beneficial when different parts of a service's infrastructure need to change at different rates. For instance, servers have different life cycles than data and it might work better to have them in separate stacks. However, having many small stacks to manage can add complexity and make it difficult to handle the integration between multiple stacks [Mor15].

8.5 Metrics Definition

In this section, we describe metrics for checking conformance to each of the decision options described in Section 8.4.

8.5.1 Model Elements Definition

We use and extend a formal model for metrics definition based on our prior work [ZNL17]. We extend it here to model the integration of component and deployment nodes. A microservice decomposition and deployment architecture model M is a tuple $(N_M, C_M, NT_M, CT_M, c\_source, c\_target, nm\_connectors, n\_type, c\_type)$ where:

- N_M is a finite set of component and infrastructure **nodes** in Model M.
- $C_M \subseteq N_M \times N_M$ is an ordered finite set of **connector edges**.
- NT_M is a set of **component types**.
- CT_M is a set of **connector types**.
- $c\_source : C_M \rightarrow N_M$ is a function returning the component that is the **source** of a link between two nodes.
- $c\_target : C_M \rightarrow N_M$ is a function returning the component that is the **target** of a link between two nodes.
- $nm\_connectors : \mathbb{P}(N_M) \rightarrow \mathbb{P}(C_M)$ is a function returning the set of connectors for a set of nodes: $nm\_connectors(nm) = \{c \in C_M : (\exists n \in nm : (c\_source(c) = n \wedge c\_target(c) \in C_M) \vee (c\_target(c) = n \wedge c\_source(c) \in C_M))\}$.
- $n\_type : N_M \rightarrow \mathbb{P}(NT_M)$ is a function that maps each node to its set of **direct and transitive node types**. (for a formal definition of node types see [ZNL17]).
- $c\_type : C_M \rightarrow \mathbb{P}(CT_M)$ is a function that maps each connector to its set of **direct and transitive connector types**. (for a formal definition of connector types see [ZNL17]).

All deployment nodes are of type *Deployment_Node*, which has the subtypes *Execution_Environment* and *Device*. These have further subtypes, such as *VM* and *Container* for *Execution_Environment*, and *Server*, *IoT Device*, *Cloud*, etc. for *Device*. Environments can also be used to distinguish logical environments on the same kind of infrastructure, such as a *Test_Environment* and a *Production_Environment*. All types can be combined, e.g. a combination of *Production_Environment* and *VM* is possible.

The microservice decomposition is modeled as nodes of type *Component* with component types such as *Service* and connector types such as *RESTful HTTP*.

The connector type *deployed_on* is used to denote a deployment relation of a *Component* (as a connector source) on an *Execution_Environment* (as a connector target). It is also used to denote the transitive deployment relation of *Execution_Environments* on other ones, such as a *Container* is deployed on a *VM* or a *Test_Environment*. The connector type *runs_on* is used to model the relations between execution environments and the devices they run on.

The type *Stack* is used to define deployments of *Devices* using the *defines_deployment_of* relation. Stacks include environments with their deployed components using the *includes_deployment_node* relation.

8.5.2 Metrics for System Coupling through Deployment Strategy Decision

The *System Coupling through Deployment Strategy* related metrics, introduced here, each have a continuous value with range from 0 to 1, with 0 representing the optimal case where the coupling is minimized by applying the recommended IaC best practices.

Shared Execution Environment Connectors Metric (SEEC). This metric $SEEC : \mathbb{P}(C_M) \rightarrow [0, 1]$ returns the number of the *shared* direct connectors from deployed service components to execution environments (e.g., containers or VMs) in relation to the total number of such service to environment connectors. For instance, the connectors of two services that are deployed on the same container are considered as shared. This gives us the proportion of the shared execution environment connectors in the system. In this context, let the function $service\_env\_connectors : \mathbb{P}(C_M) \rightarrow \mathbb{P}(C_M)$ return the set of all connectors between deployed services and their execution environments: $service\_env\_connectors(cm) = \{c \in cm : Service \in n\_type(c\_source(c)) \wedge Execution\_Environment \in n\_type(c\_target(c)) \wedge deployed\_on \in c\_type(c)\}$. Further, let the function $shared\_service\_env\_connectors : \mathbb{P}(C_M) \rightarrow \mathbb{P}(C_M)$ return the set of connectors from multiple components to the same execution environment: $shared\_service\_env\_connectors(cm) = \{c1 \in service\_env\_connectors(cm) : \exists\, c2 \in C_{EE} : c\_source(c1) \neq c\_source(c2) \wedge c\_target(c1) = c\_target(c2)\}$. Then SEEC can be defined as:

$$SEEC(cm) = \frac{|shared\_service\_env\_connectors(cm)|}{|service\_env\_connectors(cm)|}$$

Shared Execution Environment Metric (SEE). The metric $SEE : \mathbb{P}(N_M) \rightarrow [0, 1]$ measures the shared execution environments that have service components deployed on them (e.g., a container/VM that two or more services are deployed on) in relation to all executions environments with deployed services:

$$SEE(nm) = \frac{|\{n \in nm : (\exists\, c \in nm\_connectors(nm) : c \in shared\_service\_env\_connectors(cm) \wedge c\_target(c) = n)\}|}{|\{n \in nm : (\exists\, c \in nm\_connectors(nm) : c \in service\_env\_connectors(cm) \wedge c\_target(c) = n)\}|}$$

8.5.3 Metrics for System Coupling through Infrastructure Stack Grouping Decision

The metrics for *System Coupling through Infrastructure Stack Grouping* decision are return boolean values as they detect the presence of a decision option. Please note that the boolean metrics are defined for arbitrary node sets, i.e. they can be applied to any subset of a model to determine sub-models in which a particular practice is applied.

For the metrics below, let the function $services : \mathbb{P}(N_M) \rightarrow \mathbb{P}(N_M)$ return the set of services in a node set: $services(nm) = \{n \in nm : Service \in n\_type(n)\}$. Further, let the function $stack\_deployed\_envs : N_M \times \mathbb{P}(N_M) \rightarrow \mathbb{P}(N_M)$ return environments included in a Stack s with $stack\_included\_envs(s, nm) = \{e \in nm : (\exists\, c \in nm\_connectors(cm) : Stack \in n\_type(s) \wedge c\_source(c) = s \wedge c\_target(c) = e \wedge includes\_deployment\_node \in$

$c\_type(c))\}$. Let the function $stack\_deployed\_components : N_M \times \mathbb{P}(N_M) \rightarrow \mathbb{P}(N_M)$ return the components deployed via an environment by a Stack s with $stack\_deployed\_components(s, nm) = \{n \in nm : (\exists\, c \in nm\_connectors(cm) : Component \in n\_type(c\_source(c)) \wedge c\_target(c) \in stack\_included\_envs(s) \wedge deployed\_on \in c\_type(c))\}$. With this, $stacks\_deploying\_services : \mathbb{P}(N_M) \rightarrow \mathbb{P}(N_M)$ can be defined, which returns all stacks that deploy at least one service: $stacks\_deploying\_services(nm) = \{s \in nm : Stack \in n\_type(s) \wedge (n \in services(nm) : n \in stack\_deployed\_components(s))\}$.
Finally, we can define $stacks\_deploying\_non\_service\_components(nm) : \mathbb{P}(N_M) \rightarrow \mathbb{P}(N_M)$ as $stacks\_deploying\_non\_service\_components(nm) = \{s \in nm : Stack \in n\_type(s) \wedge (n \in nm : n \in stack\_deployed\_components(s) \wedge n \notin services(nm))\}$.

Monolithic Stack Detection Metric (MSD). The metric $MSD : \mathbb{P}(N_M) \rightarrow Boolean$ returns $True$ if only one stack is used in a node set that deploys more than one service via stacks (e.g., a number of/all system services are deployed by the only defined stack in the infrastructure) and $False$ otherwise.

$$MSD(nm) = \begin{cases} True: & \text{if } |stacks\_deploying\_services(nm)| = 1 \\ False: & otherwise \end{cases}$$

Application Group Stack Detection Metric (AGSD). The metric $AGSD : \mathbb{P}(C_M) \rightarrow Boolean$ returns $True$ if multiple stacks are used in a node set to deploy services and more services are deployed via stacks than there are stacks (e.g., system services are deployed by one stack and other elements such as routes are deployed by different stack(s)). That is, multiple services are clustered in groups on at least one of the stacks.

$$AGSD(nm) = \begin{cases} True: & \text{if } |stacks\_deploying\_services(nm)| > 1 \wedge \\ & \quad |stacks\_deploying\_services(nm)| < |services(nm)| \\ False: & otherwise \end{cases}$$

Service-Stack Detection Metric (SES). The metric $SES : \mathbb{P}(N_M) \rightarrow Boolean$ returns $True$ is the number of services deployed by stacks equals the number of stacks (e.g., each system service is deployed by its own stack).

$$SES(nm) = \begin{cases} True: & \text{if } |stacks\_deploying\_services(nm)| = |services(nm)| \\ False: & otherwise \end{cases}$$

Micro-Stack Detection Metric (MST). The metric $MST : \mathbb{P}(C_M) \rightarrow Boolean$ returns $True$ if SES is $True$ and also there is one or more stacks that deploy non-service components (e.g., databases, monitoring components, etc.). For instance, a service is deployed by one stack and its database is deployed by another stack as well as a router is deployed by its own stack, etc.:

$$MST(nm) = \begin{cases} True: & \text{if } SES(nm) = True \wedge \\ & \quad |stacks\_deploying\_non\_service\_components(nm)| > 0 \\ False: & otherwise \end{cases}$$

Services per Stack Metric (SPS). To measure how many services on average are deployed by a service-deploying stack, we define the metrics $SPS : \mathbb{P}(C_M) \rightarrow \mathbb{R}$ as:

$$SPS(nm) = \frac{\begin{array}{r}|\{n \in services(nm) : (\exists s \in nm : Stack \in n\_type(s) \wedge \\ n \in stack\_deployed\_components(s))\}|\end{array}}{|stacks\_deploying\_services(nm)|}$$

Components per Stack Metric (CPS). To measure how many components on average are deployed by a component-deploying stack, we define the metrics $CPS : \mathbb{P}(C_M) \rightarrow \mathbb{R}$ as:

$$CPS(nm) = \frac{\begin{array}{r}|\{n \in nm : (\exists s \in nm : Stack \in n\_type(s) \wedge \\ n \in stack\_deployed\_components(s))\}|\end{array}}{\begin{array}{r}|stacks\_deploying\_services(nm) \cup \\ stacks\_deploying\_non\_service\_components(nm)|\end{array}}$$

8.6 Case Studies

In this section, we describe the case studies used to evaluate our approach and test the performance of the metrics. We studied three open-source microservice-based systems. We also created variants that introduce typical violations of the ADDs described in the literature or refactorings to improve ADD realization to test how well our metrics help to spot these issues and improvements. The cases are summarized in Table 8.1 and metrics results are presented in Table 8.2.

Case Study 1: eShopOnContainers Application. The *eShopOnContainers* case study is a sample reference application realized by Microsoft, based on a microservices architecture and Docker containers, to be run on Azure and Azure cloud services. It contains multiple autonomous microservices, supports different communication styles (e.g., synchronous, asynchronous via a message broker). Furthermore, the application contains the files required for deployment on a Kubernetes cluster and provides the necessary IaC scripts to work with ELK for logging (Elasticsearch, Logstash, Kibana).

To investigate further, we performed a full manual reconstruction of an architecture component model and an IaC-based deployment model of the application as ground truth for the case study. Figure 8.2 shows the excerpt component model specifying the component types (e.g., *Services, Facades, and Databases*), and connector types (e.g., *database connectors*, etc.) as well as all the IaC-based deployment component types (e.g., *Web Server, Cloud Server, Container, Infrastructure Stack, Storage Resources*, etc.) and IaC-based deployment connector types (e.g., *defines deployment of, deployed on*, etc.) using types as introduced in Section 8.5 shown here as stereotypes.

The component model, consists of in total 235 elements such as component types, connector types, IaC-based deployment component types and IaC-based deployment connector types. More specifically, *19 Infrastructure Stacks*, *19 Execution Environments*, *6 Storage Resources*, *7 Services*, *6 Databases*, *19 Stack-to-Execution Environment connectors*, *7 Stack-to-Service connectors* and *6*

Case Study ID	*Model Size*	*Description / Source*
CS1	68 components 167 connectors	E-shop application using pub/sub communication for event-based interaction as well as files for deployment on a Kubernetes cluster. All services are deployed in their own infrastructure stack (from `https://github.com/dotnet-architecture/eShopOnContainers`).
CS1.V1	67 components 163 connectors	Variant of Case Study 1 in which half of the services are deployed on the same execution environment and some infrastructure stacks deploy more than one service.
CS1.V2	60 components 150 connectors	Variant of Case Study 1 in which some services are deployed on the same execution environment and half of the non-services components are deployed by a component-deploying stack.
CS2	38 components 95 connectors	An online shop that demonstrate and test microservice and cloud-native technologies and uses a single infrastructure stack to deploy all the elements (from `https://github.com/microservices-demo/microservices-demo`).
CS2.V1	40 components 101 connectors	Variant of Case Study 2 where multiple infrastructure stacks are used to deploy the system elements as well as some services are deployed on the same execution environment.
CS2.V2	36 components 88 connectors	Variant of Case Study 2 where two infrastructure stacks are used to deploy the system elements (one for the services and one for the rest elements) as well as some services are deployed on the same execution environment.
CS3	32 components 118 connectors	Robot shop application with various kinds of service interconnections, data stores, and Instana tracing on most services as well as an infrastructure stack that deploys the services and their related elements (from `https://github.com/instana/robot-shop`).
CS3.V1	56 components 147 connectors	Variant of Case Study 3 where some services are deployed in their own infrastructure stack as well as some services are deployed on the same execution environment.
CS3.V2	56 components 147 connectors	Variant of Case Study 3 where all services are deployed in their own infrastructure stack as well as all services are deployed on their own execution environment.

Table 8.1: Overview of modeled case studies and the variants (size, details, and sources)

Storage Resources-to-Database connectors. There are also other 146 elements in the application (e.g., *Web Server*, *Cloud Server*, *Stack-to-Web Server connectors*, etc.).

The values of metrics *SEEC* and *SEE* show that *Single Service per Execution Environment* pattern is fully supported. We treat both components and connectors as equally essential elements; thus, we use two metrics to assess coupling in our models. Given that *SEE* returns the shared execution environments, it is crucial to measure how strongly these environments are shared. The *SEEC* value indicates this specific aspect by returning the proportion of the shared connectors that these environments have.

The application uses multiple stacks to deploy the services and the other elements, and this is shown by the outcomes of the metrics for the *System Coupling through Infrastructure Stack Grouping* decision. Since multiple stacks have been detected the metrics *MSD* and *AGSD* return *False*. The *SES* metric returns *True* meaning that the *Service Stack* pattern is used. The *MST* returns *True* which means the *Micro-Stack* pattern for the node sub-set *Storage Resources* is also used. The *SPS* value shows that every service is deployed by a service-deploying stack. Furthermore, *CPS* also shows that components that belong to node sub-set *Storage Resources* are deployed by a component-deploying stack. Overall the metrics results in this case study show no coupling issue in deployment, and all best practices in our ADDs have been followed.

For further evaluation, we created two variants to test our metrics' performance in more problematic cases. Our analysis in *CS1.V1* shows that half of the execution environments are shared, and around two-thirds of the connectors between services and execution environments are also considered as shared, meaning these execution environments are strongly coupled with the system services. Using both values, we have a more complete picture of the coupling for all essential elements in this model. Furthermore, the *SPS* value indicates that the *Service Stack* pattern is partially supported, meaning some services are grouped in the same stacks. The analysis in *CS1.V2* shows that our metrics can measure all the additional violations that have been introduced.

Case Study 2: Sock Shop Application. The *Sock Shop* is a reference application for microservices by the company Weaveworks to illustrate microservices architectures and the company's technologies. The application demonstrates microservice and cloud-native technologies. The system uses *Kubernetes* for container-orchestration and services are deployed on *Docker* containers. *Terraform* infrastructure scripts are provided to deploy the system on *Amazon Web Services (AWS)*. We believe it to be a good representative example of the current industry practices in microservice-based architectures and IaC-based deployments.

The reconstructed model of this application contains in total 133 elements. In particular, *1 Infrastructure Stack*, *13 Execution Environments*, *3 Storage Resources*, *7 Services*, *4 Databases*, *13 Stack-to-Execution Environment connectors*, *7 Stack-to-Service connectors* and *4 Storage Resources-to-Database connectors*. There are also another 74 elements in the application such as *Web Server*, *Cloud Server*, *Stack-to-Web Server connectors* and *Execution Environment-to-Cloud Server connectors*.

We have tested our metrics to assess the conformance to best patterns and practices in IaC-based deployment. The outcome of the metrics related to *System Coupling through Deployment Strategy* decision shows that also this application fully supports the *Single Service per Execution*

Environment pattern. That is, all services are deployed in separate execution environments. Regarding the *System Coupling through Infrastructure Stack Grouping* decision, we detected the *Monolith Stack* pattern, which means one stack defines the deployment of all system elements, resulting in a highly coupled deployment. The metrics *AGSD*, *SES*, and *MST* are all *False*, and *SPS* and *CPS* return 0, since a monolith stack has been detected. These values can guide architects to improve the application by restructuring the infrastructure to achieve the desired design.

In our variants, we introduced gradual but not perfect improvements. The metrics results for *CS2.V1* show an improvement compared to the initial version. That is, *Monolith Stack* pattern is not used since three infrastructure stacks have been detected, and some services are deployed by service-deploying stacks. In *CS2.V2* there is a slight improvement since *Application Group Stacks* has been detected. However, *SEEC* and *SEE* metrics indicate that there is a strong coupling between services and execution environments. In both variants, the metrics have well detected the improvements made.

Case Study 3: Robot-Shop Application. *Robot-Shop* is a reference application by the company Instana provides to demonstrate polyglot microservice architectures and Instana monitoring. It includes the necessary IaC scripts for deployment. All system services are deployed on *Docker* containers and use *Kubernetes* for container-orchestration. Moreover, *Helm* is also supported for automating the creation, packaging, configuration, and deployment to *Kubernetes* clusters. End-to-end monitoring is provided, and some services support *Prometheus* metrics.

The reconstructed model of this application contains in total 150 elements. In particular, *2 Infrastructure Stacks*, *18 Execution Environments*, *2 Storage Resources*, *10 Services*, *3 Databases*, *13 Stack-to-Execution Environment connectors*, *10 Stack-to-Service connectors* and *3 Storage Resources-to-Database connectors*. There are also 89 additional elements in the application.

The metrics results for the *System Coupling through Deployment Strategy* decision are both optimal, showing that in this application all services are deployed in separate execution environments. For the *System Coupling through Infrastructure Stack Structuring* decision, the *AGSD* metric return *True* which means that the *Application Group Stack* pattern is used, resulting in highly coupled services' deployment. Thus, the metrics *SES* and *MST* are *False* and *SPS* and *CPS* return 0. According to these values, architects can be supported to address the detected violations (e.g., as done in *CS3.V2*).

In the variants, we introduced one gradual improvement first and then a variant that addresses all issues. Our analysis in *CS3.V1* shows significant improvement in infrastructure stack grouping. Most of the services are deployed on their own stack, and components that belong to node sub-set *Storage Resources* are completely deployed by a separate stack. However, coupling between services and execution environments has also been detected. Variant *CS3.V2* is even more improved since, in this case, all services are deployed by service-deploying stacks, and no coupling has been detected. In both variants, the metrics have faithfully identified the changes made.

Table 8.2: Metrics Calculation Results

Metrics	CS1	CS1.V1	CS1.V2	CS2	CS2.V1	CS2.V2	CS3	CS3.V1	CS3.V2
System Coupling through Deployment Strategy									
SEEC	0.00	0.71	0.42	0.00	0.25	0.62	0.00	0.37	0.00
SEE	0.00	0.50	0.20	0.00	0.14	0.40	0.00	0.16	0.00
System Coupling through Infrastructure Stack Grouping									
MSD	False	False	False	True	False	False	False	False	False
AGSD	False	False	False	False	False	True	True	False	False
SES	True	False	False	False	False	False	False	False	True
MST	True	True	False	False	False	False	False	False	True
SPS	1.00	0.20	0.57	0.00	0.12	0.00	0.00	0.62	1.00
CPS	1.00	1.00	0.50	0.00	0.00	0.00	0.00	1.00	1.00

8.7 Discussion

Discussion of Research Questions. To answer **RQ3.7**, we proposed a set of generic, technology-independent metrics for each IaC decision, and each decision option corresponds to at least one metric. We defined a set of generic, technology-independent metrics to assess each pattern's implementation in each model automatically and conducted three case studies to test the performance of these metrics. For assessing pattern conformance, we use both numerical and boolean values. In particular, *SEEC* and *SEE* measures return a range from 0 to 1, with 0 representing the optimal case where a set of patterns is fully supported. Having this proportion, we can assess not only the existence of coupling but also how severe the problem is. However, this is not the case for the *MSD*, *AGSD*, *SES*, and *MST* metrics that return *True* or *False*. Using these metrics, we intend to detect the presence of the corresponding patterns. For *Service Stack* and *Micro Stack* decision options, we introduce two additional metrics with numerical values that can be applied on node subsets to assess the level of the pattern support when patterns are not fully supported. However, applying them on node subsets has the limitation that many runs need to be made, leading to ambiguous results. Our case studies' analysis shows that every set of decision-related metrics can detect and assess the presence and the proportion of pattern utilization.

Regarding **RQ3.8**, we can assess that our deployment meta-model has no need for significant extensions and is easy to map to existing modeling practices. More specifically, to fully model the case studies and the additional variants, we needed to introduce 13 device type nodes and 11 execution environment nodes types such as *Cloud Server* and *Virtual Machine* respectively, and 9 deployment relation types and 7 deployment node relations. Furthermore, we also introduced a deployment node meta-model to cover all the additional nodes of our decisions, such as *Storage Resources*. The decisions in *System Coupling through Deployment Strategy* require modeling several elements such as the *Web Server*, *Container*, and *Cloud Server* nodes types and technology-related connector types (e.g. *deployed on*) as well as deployment-related connector types (e.g. *Runs on, Deployed in Container*). For the *Coupling through Infrastructure Stack*

Grouping decision, we have introduced attributes in the system nodes (e.g., in *Infrastructure Stack, Storage Resources*) and connector types (e.g., *defines deployment of, includes*).

Threats to Validity. We mainly relied on third-party systems as the basis for our study to increase internal validity and thus avoid bias in system composition and structure. It is possible that our search procedures resulted in some unconscious exclusion of specific sources; we mitigated this by assembling a team of authors with many years of experience in the field and conducting a very general and broad search. Because our search was not exhaustive and the systems we found were created for demonstration purposes, i.e., were relatively modest in size, some potential architectural elements were not included in our metamodel. Furthermore, this poses a potential threat to the external validity of generalization to other, more complex systems. However, we considered widely accepted benchmarks of microservice-based application as reference applications, in a way to reduce this possibility. Another potential risk is that the system variants were developed by the author team itself. However, this was done following best practices documented in the literature. We were careful to change only certain aspects in a variant and keep all other aspects stable. Another possible source of internal validity impairment is the modeling process. The author team has considerable experience with similar methods, and the systems' models have been repeatedly and independently cross-checked, but the possibility of some interpretive bias remains. Other researchers may have coded or modeled differently, resulting in different models. Because our goal was only to find a model that could describe all observed phenomena, and this was achieved, we do not consider this risk to be particularly problematic for our study. The metrics used to assess the presence of each pattern were deliberately kept as simple as possible to avoid false positives and allow for a technology-independent assessment.

8.8 Conclusions and Future Work

We have investigated the extent to which it is possible to develop a method to automatically evaluate coupling-related practices of ADDs in an IaC deployment model. Our approach models the critical aspects of the decision options with a minimal set of model elements, which means it is possible to extract them automatically from the IaC scripts. We then defined a set of metrics to cover all decision options described in Section 8.4 and used the case studies to test the performance of the generated metrics. Before, for the coupling aspects of IaC deployment models, no general, technology-independent metrics have been studied in depth. Our approach treats deployment architectures as a set of nodes and links, considering the technologies used, which were not supported in prior studies. The goal of our approach is a continuous evaluation, taking into account the impact of continuous delivery practices, in which metrics are evaluated continuously, indicating improvements and loose coupling of deployment architecture compliance.

In future work, we plan to study more decisions and related metrics, test further in larger systems, and integrate our approach in a systematic guidance tool.

9 Assessing Security Conformance in Infrastructure-as-Code Deployments

Building on the approach outlined in Chapter 8, this chapter delves into security practices in IaC-based deployments, with a specific focus on observability, access control, and traffic control. We present three architectural design decisions related to security: *Security Observability*, *Security Access Control*, and *Security Traffic Control*. We also report a total of 13 metrics that evaluate compliance or noncompliance with the relevant security practices.

9.1 Introduction

Cloud computing has significantly increased the number of infrastructure nodes that a system requires [Nyg07]. Moreover, nowadays systems are being released to production more rapidly, often many times a day, resulting in more and more frequent changes in the infrastructure [HF10, Nyg07]. *Infrastructure as Code (IaC)* is the management and provisioning of the infrastructure using reusable scripts instead of manual processes [Mor15]. IaC can ensure that a provisioned environment remains the same every time it is deployed in the same configuration, and configuration files contain infrastructure specifications making the process of editing and distributing configurations easier [Mor15, ABDN+17]. IaC can also contribute to improving consistency, security, avoiding errors, and reducing manual configuration effort. Without an IaC practice in place, it becomes increasingly difficult to manage the scale of current systems' components and infrastructure.

IaC technologies (e.g., Ansible, Terraform) enable to provision, deploy, and configure arbitrary application architectures. Thus, developers and operators, respectively, can model any desired deployment. This freedom quickly results in severe problems if security-related aspects are not taken into account. For example, vulnerabilities in IaC deployment models (e.g. weak access and traffic control) could allow attackers to access procedures and run code to hack the application.

The focus of this work is to investigate security-related practices in deployment architectures that can be configured and managed via IaC scripts. In this context, we formulate a number of *Architectural Design Decisions (ADDs)* with corresponding decisions options that have been documented as best practices in the gray literature, i.e. informal guidelines for practitioners, blog posts, public repositories, and so on. Based on this architectural knowledge, we aim to provide an automatic assessment of architecture conformance to these practices in the IaC deployment models.

So far, not many architectural patterns or formal guidelines that reflect security best practices in the context of IaC have been documented. Currently, the literature seems to rather focus on specific code-level practices. In combination with the fact that industry-scale systems support

multiple such architectural practices at once and different implementations of them, this makes it difficult to assess whether an IaC deployment model that implicitly describes also the application's architecture is conforming to recommended best practices or not. In modern cloud-based architectures, such as microservice architectures [New15] and other frequently released systems [Nyg07], an automatic assessment method would vastly improve cost-effectiveness and produce more accurate results. For instance, this could be applied in the context of continuous delivery practices employed in these systems requiring the automated setup of infrastructures in usually every run of the delivery pipeline [HF10]; for example, consider production environments and identical test environments. Therefore, this work aims at answering the following research questions:

- **RQ3.9** How can conformance to IaC architecture security-related decision options (i.e. patterns or practices) be automatically assessed?
- **RQ3.10** What kinds of measures can be applied to asses this conformance and how well do they perform?
- **RQ3.11** What is the minimal set of modeling elements required in an IaC deployment model to compute these measures?

To address these research questions, we introduced a set of metrics for different security-related architectural decisions to cover all their possible decision options. We derived a ground truth from a manual assessment of a set of models that evaluates their conformance to all considered decision options and their combinations. In particular, the decisions focus on *Security Observability*, *Security Access Control*, and *Security Traffic Control*. To create the ground truth, we first objectively assessed whether each decision option is supported in a system and to which extent. We derived an ordinal rating scheme to distinguish different levels of security support, and If security flaws were indeed found, the rating scheme indicates how far the conformance to best practices is nonetheless supported. Both, the rating scheme and the ground truth assessment, have been reviewed and discussed with two industrial experts (both having experience in security of microservice systems) until a consensus was reached. We then used the ground truth data to assess how well the defined metrics can predict the ground truth assessment by performing an ordinal regression analysis.

In this chapter, we propose a deployment model-based approach, which uses only modeling elements that can be derived from the typical scripts used by IaC technologies. A deployment model implicitly also describes the architecture of the application to be deployed, thus, enabling assessing the conformance to architectural decisions. For this reason, it is important to be able to work with a minimal set of elements, else it might be difficult to parse them from the IaC scripts.

This chapter is structured as follows: Section 9.2 compares related work. Next, we describe the research methods and the tools we have applied in our study in Section 9.3. In Section 9.4 we explain the decisions and the related patterns and practices. In Section 9.5 we report how the ground truth data for each decision is calculated. Section 9.6 introduces our hypothesized metrics. Section 9.7 describes the metrics calculations results for our models and the results of the ordinal regression analysis. Section 9.8 discusses the RQs regarding the evaluation results and analyses the threats to validity. Finally, in Section 9.9 we draw conclusions and discuss future work.

9.2 Related Work

9.2.1 Related Works on Best Practices and Patterns

As IaC practices are becoming more and more popular and widely adopted by the industry, there is also more scientific research into collecting and systematizing IaC-related patterns and practices. Kumara et al. [KGR+21] present a broad catalog of best and bad practices, both language-agnostic and language-specific, that reflect implementation issues, design issues, and violations of essential IaC principles. Morris [Mor15] presents a collection of guidance on how to manage Infrastructure as Code. In his book, there is a detailed description of technologies related to IaC-based practices and a broad catalog of patterns and practices embodied in several categories. Language-specific practices have been proposed by Sharma et al. [SFS16] who present a catalog of design and implementation smells for Puppet. Schwarz et al. [SSL18] present a catalog of smells for Chef. Our work also follows IaC-specific recommendations described in AWS [AWS21], OWASP [OWA21a, OWA21c, OWA21b] and the Cloud Security Alliance [Clo18, Clo21]. In contrast to our work, many of these works are less focused on architectural decisions in the deployment architecture of the systems to be deployed. Furthermore, these works do not support conformance checking as suggested in our work.

9.2.2 Related works on Frameworks and Metrics

Tool-based and Network Metrics Approaches There are several studies that propose tools and metrics for assessing and improving the quality of IaC deployment models. Dalla Palma et al. [DDPT20, DDT20] propose a catalog of 46 quality metrics focusing on Ansible scripts to identify IaC-related properties and show how to use them in analyzing IaC scripts. Wurster et al. [WBH+20] present TOSCA Lightning, an integrated toolchain for specifying multi-service applications with TOSCA Light and transforming them into different production-ready deployment technologies. They also present a case study on the toolchain's effectiveness based on a third-party application and Kubernetes. A tool-based approach for detecting smells in TOSCA models is proposed by Kumara et al. [KVM+20]. Sotiropoulos et al. [SMS20] develop a tool-based approach that identifies dependencies-related issues by analyzing Puppet manifests and their system call trace. Van der Bent et al. [vdBHVG18] define metrics that reflect also the best practices to assess Puppet code quality. Pendleton et al. [PGLCX16] present a comprehensive survey on security metrics. It focuses on how a system security state can evolve as an outcome of cyber-attack defense interactions. They then propose a security metrics framework for measuring system-level security.

Although some of these works focus on quality assurance of IaC systems, none of them address and focus specifically on security critical concerns and measures in IaC deployment models, which is the case in our work.

Software Architecture Conformance Checking In [FBKL17, KBKL18], an approach for automatically checking the compliance of declarative deployment models during design time is introduced. The approach allows modeling compliance rules in the form of a pair of deployment model fragments. One of the fragments represents a detector subgraph that determines

whether the rule applies to a given deployment model or not. If a compliance rule is found to be applicable, the second fragment determines a desired structure that the deployment model must contain. Comparing the model fragments to a given deployment model happens using subgraph isomorphism. In contrast to our study, this approach is generic and does not introduce specific compliance rules, e.g. for the security domain, and assumes the rule modeler is capable of translating best practices into compliance rules of the expected format. Moreover, it only provides a Boolean outcome indicating whether a compliance rule is violated or not, rather than indicating to what degree it is violated.

Our approach can be considered as a metrics-based, IaC-specific approach for deployment architecture conformance checking. Many approaches related to architecture conformance checking are usually based on automated extraction techniques [GAK99, VDHK+04]. Conformance to architecture patterns [GAK99, HZ14] or other kinds of architectural rules [VDHK+04] can often be checked by such approaches. As proposed in our work, here techniques that are based on various interrelated metrics of IaC-related metrics to cover security related features and practices do not yet exist. Furthermore, none of these approaches treat deployment architectures as a set of nodes and connectors i.e. a deployment architecture. Also, none are able to produce assessments that combine different assessment parameters (i.e. metrics). Such metrics, if automatically computed, can be utilized as part of larger assessment models/frameworks during design and development time.

9.3 Research and Modeling Methods

9.3.1 Overview

Figure 9.1 shows the research steps followed in this study. We first studied knowledge sources related to IaC-specific security best practices from industry organizations, practitioners blogs and scientific literature [Clo21, Clo18, OWA21b, OWA21c, OWA21a, Goo21, AWS21, KGR+21, Mor15, SSL18, SFS16]. We then analyzed the data collected using qualitative methods based on Grounded Theory [CS90] coding methods, such as open and axial coding, and extracted the three core IaC security-related decisions described in section 9.4 along with their corresponding decision drivers and impacts. Based on our dataset, which contains generated models, we derived a ground truth as well as a set of metrics to measure the proportion of the support of each practice. Next, we defined a rating scheme on support or violation of best practices and refined the ground truth assessment. The generated models, the rating scheme, and ground truth assessment were independently analyzed by two industrial security experts, each with around 5 years of experience in developing and architecting cloud-based systems. We discussed each assessment with the experts until a consensus was reached and used the ground truth data to assess how well the hypothesized metrics can possibly predict the ground truth data by performing an ordinal regression analysis..

9.3.2 Model Selection Methods

This study focuses on conformance to security-related features and practices in IaC deployment models. To be able to study this, we first performed an iterative study of a variety of IaC

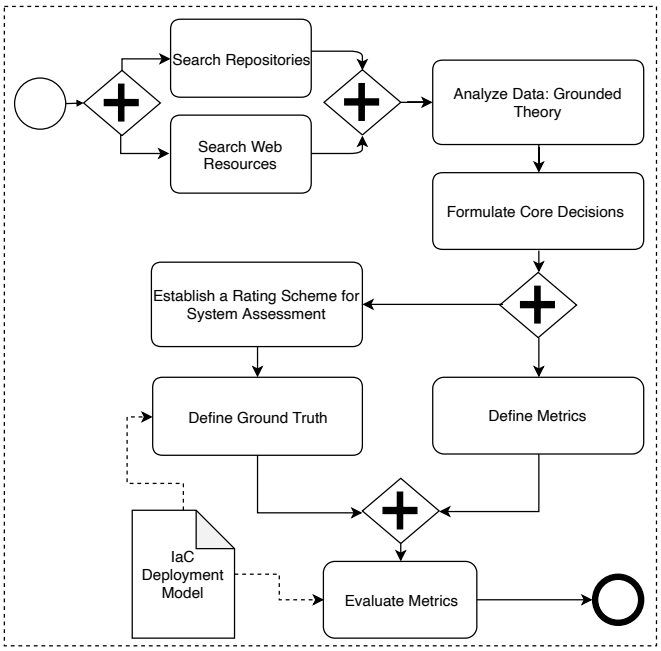

Figure 9.1: Overview of the research method followed in this study

knowledge sources. Next, we refined a meta-model, which contains all the required elements to help us reconstruct the actual architecture that gets deployed by the IaC model. For problem investigation and as an evaluation model set for eventually creating a ground truth for our study, we gathered a number of IaC systems, summarized in Table 9.1. Each of them is taken directly from a system published by practitioners (on GitHub and/or practitioner blogs) from 9 independent sources. We mostly used search engines to find all systems used in this study. Searching in such engines might lead to selection bias and to different types of data [PD18]. For avoiding such a selection bias we initially started our research with keywords taken from AWS [AWS21], OWASP [OWA21a, OWA21c, OWA21b] and the Cloud Security Alliance [Clo18, Clo21] such as *Infrastructure Architecture Security*, *Infrastructure Hardening*, *Monitoring and Logging* and *Security Groups and Traffic Control* etc. We used major search engines (e.g. Google, Bing, DuckDuckGo) and topic portals (e.g. InfoQ, DZone) to find relevant practitioner texts.

The systems we found were developed by practitioners with relevant experience, which justifies the assumption that they provide a good representation of the IaC security-related best practices summarized in Section 9.4. We performed a fully manual static code analysis for the IaC models that are in the repos together with the application source code. For model creation, we used the Python modeling library CodeableModels described in Chapter 3.7. Variations were modeled to cover the complete design space, including also the bad practices that can cause a violation, of our three ADDs and described in Section 9.4, according to the referenced practitioner sources. Apart from the variations described in Table 9.1, all other system aspects remained the same as in the base models. This resulted in a total of 21 models summarized in Table 9.1. We assume that

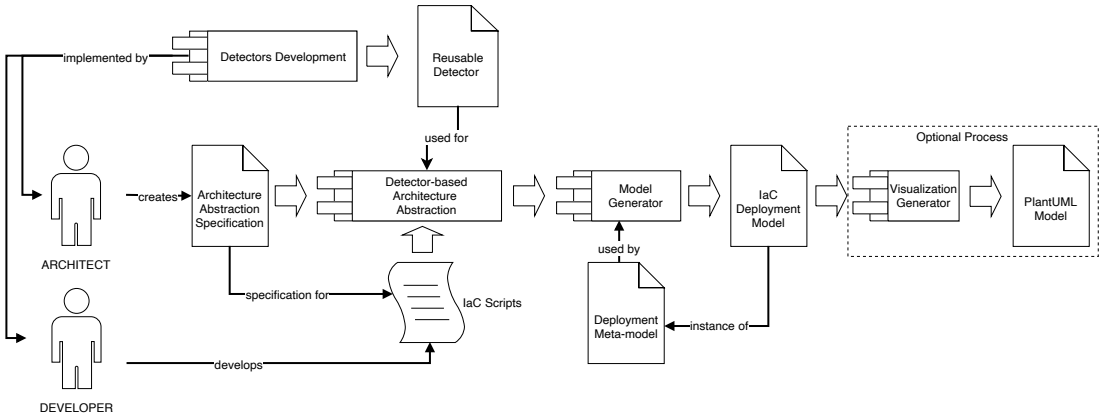

Figure 9.2: Tool Flow Architecture of the Proposed System

our evaluation models are close to models used in practice and real-world practical needs. As many of them are open source systems with the purpose of demonstrating practices, they are at most of medium size, though.

9.3.3 Metrics Definition, Ground Truth Calculation, and Statistical Evaluation Methods

We defined a set of metrics for each ADD to measure conformance to the respective practices in the design decisions from Section 9.4, i.e. at least one metric per major feature/practice. Based on the manual assessment of the models from Table 9.1, we derived a ground truth for our study (the ground truth and its calculation rules are described in Section 9.4). The ground truth is established by assessing whether each decision option is supported, partially supported, or not supported. We combined the outcome of all decision's options and then derived an ordinal assessment on how well the decision is supported in each model, using the scale:

- $++$: very well supported;
- $+$: well supported, but aspects of the solution could be improved;
- $\sim$: serious flaws in the security design, but substantial support can already be found in the system;
- $-$: serious flaws in the security design, but initial support can already be found in the system;
- $--$: no support for the security tactic can be found in the system.

We discussed this assessment scheme with the two industrial security experts until a consensus was reached. The authors assessed all the system models and the variants for conformance to each of the ADDs, and the assessments are again reviewed by the two industrial security experts.

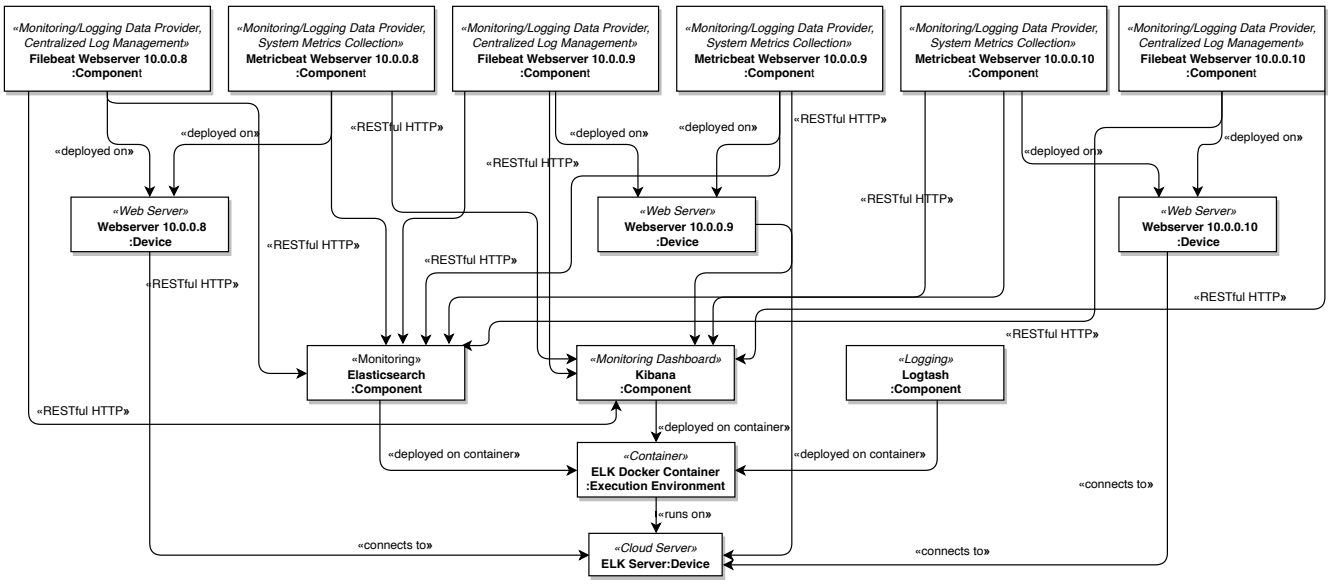

Figure 9.3: Overview of a reconstructed model (Model S4 in Table 9.1).

Our scale does not assume equal distances, but it assumes the given order. We then used the ground truth data to assess how well the defined metrics can predict the ground truth data by performing an ordinal regression analysis.

Ordinal regression is a widely used method for modeling an ordinal response's dependence on a set of independent predictors. For the ordinal regression analysis we used the *lrm* function from the *rms* package in R [FEH15].

9.3.4 Methods for IaC Architectural Reconstruction

From an abstract point of view, an IaC system is composed of nodes and connectors with a set of nodes types and a set of connector types. Our work has the goal to automate metrics calculation and assessment based on the node model of an IaC system. That is, if the system is manually modeled or the model can be derived automatically from the IaC scripts, our approach is applicable. For modeling IaC deployment models we followed the method reported in our previous work [ZNL17]. All the code and models used in and produced as part of this study have been made available online for reproducibility[1].

In the context of this approach, and based on our work [NZP+21], we have introduced a number of detectors in order to automatically reconstruct the IaC architecture from the source code. Combining the automatic reconstruction with the automatic metrics calculation, the overall assessment process can be improved. So far we have done this only for one system and for one technology (S4 in Table 9.1) fully and are developing extensions for the other systems at the moment. A detector is a piece of code that looks for specific characteristics in the IaC deployment model and produces architectural information. The idea is that by having many

[1] https://doi.org/10.5281/zenodo.6559385

Model ID	*Model Size*	*Description / Source*
S1	12 nodes 13 connectors	Consul-based application with specified security groups (from https://github.com/apiacademy/microservices-deployment).
S2	12 nodes 13 connectors	Variant of S1 which uses API keys authentication practice.
S3	8 nodes 9 connectors	Variant of S1 which uses Plaintext authentication practice.
S4	14 nodes 25 connectors	Elasticsearch, Logstash, and Kibana (ELK)-based system using metrics collection and log management tools (from https://github.com/babtunee/azure-cloud-security-architecture).
S5	14 nodes 25 connectors	Variant of S4 using Ingress Traffic Control and partial support of SSL-Based Authentication.
S6	14 nodes 25 connectors	Variant of S4 using Ingress Traffic Control and full support of Token-Based Authentication.
S7	10 nodes 17 connectors	ELK-based system using metrics collection and application monitoring related tools (from https://github.com/deviantony/docker-elk).
S8	10 nodes 17 connectors	Variant of S7 with additional servers and Plaintext Authentication.
S9	17 nodes 48 connectors	Variant of S8 using additional tools for centralized log management, service availability and performance analytics as well as Ingress and Egress traffic control, Token-Based Authentication and Single Sign-On authentication.
S10	8 nodes 11 connectors	ELK-based system using metrics collection related tools (from https://github.com/ManuelMourato25/elastic-stack-architecture-example).
S11	10 nodes 14 connectors	Variant of S10 using Application monitoring, Ingress traffic control and Security groups
S12	16 nodes 30 connectors	ELK-based system using metrics collection and log management tools as well as security groups (from https://github.com/frahmeto/Elk-Stack-Project-1).
S13	19 nodes 57 connectors	Variant of S12 using additional Application monitoring, service availability, performance analytics and API keys authentication.
S14	10 nodes 15 connectors	ELK-TLS-based system using metrics collection, log management, performance analytics and endpoint security tools (from https://github.com/swimlane/elk-tls-docker).
S15	14 nodes 35 connectors	Variant of S14 using additional servers and Application monitoring, performance analytics and SSL-Based Authentication, ingress traffic control and Single Sign-On authentication
S16	8 nodes 11 connectors	Java ELK-based system with service availability and log management tools (from https://github.com/twogg-git/java-elk).
S17	6 nodes 5 connectors	Variant of S16 using only Plaintext authentication and no additional tool to support other features.
S18	14 nodes 21 connectors	Variant of S16 which uses application monitoring, Ingress and Egress traffic control and security groups.
S19	9 nodes 8 connectors	Openshift-based application with application monitoring and metrics collection tools (from https://github.com/redhat-helloworld-msa/k).
S20	9 nodes 8 connectors	Variant of S19 with SSL-based and Single Sign-On authentication.
S21	8 nodes 7 connectors	Terraform application deployed in AWS with metric collection tool and security groups (from https://github.com/ryanmcdermott/terraform-microservices-example).

Table 9.1: Selected IaC Deployments Models: Size, Details, and Sources

of these, all lightweight and possibly looking at different languages (e.g. Ansible, Terraform), then deployment architectures can be extracted in an automatic way. Figure 9.3 shows the

resulting model after applying our detectors. Moreover, from the grounded theory analysis, we defined deployment meta-models, in which nodes are extended by the stereotype nodes types and connectors by the stereotype connector types, to reconstruct model instances for the deployments.

9.3.5 The Tool Flow of the Approach

Figure 9.2 illustrates the general tool flow architecture on how the building blocks interact. Using the detectors it's possible to generate models from our modeling tool Codeable Models. As Figure 9.2 shows, the user can be either a developer or an architect. The architect specifies the architecture abstraction specification for an IaC system, while the developer implements it in the IaC scripts potentially of different technologies. However, both can be involved in the development of detectors. The user can use the architecture specification, IaC scripts and the detector toolchain to generate the IaC deployment model. The architecture specification and the IaC scripts are the inputs of the detector component. The detector performs the detections, and, if successful, uses a code generator to generate the Codeable Models code. The generator uses the deployment meta-model to generate the IaC deployment model. It also contains a visualization generator for generating PlantUML diagrams such as the one in Figure 9.3.

9.4 IaC Security-Related ADDs

In this section, we briefly introduce the three security-related decisions along with their decision options (i.e. the relevant patterns and practices). We also discuss the impact on relevant security aspects, which we later on use as an argumentation for our manual ground truth assessment in Section 9.4.

Security Observability An important aspect in deployment architectures is to be able to identify and respond to what is happening within a system, what resources need to be observed, and inspect what is causing a possible issue. Using observability practices to collect, aggregate, and analyze log data and metrics is a key for establishing and maintaining more secure, flexible, and comprehensive systems [Clo18]. Moreover, collecting and analysing information improves the detection of suspicious system behavior or unauthorised system changes on the network and can facilitate the definition of different behavior types, in which an alert should be triggered. A crucial decision in infrastructure observability is *Server Monitoring* which is an essential process of observing the activity on servers, either physical or virtual. A single server can support hundreds or even thousands of requests simultaneously. As such, ensuring that all of the servers are operating according to expectations is a critical part of managing an infrastructure. Another equally critical decision is *Application Monitoring* which is a process of collecting log data to support aspects such as track availability, bugs, resource use, and changes to performance in an application. Moreover, features such as *Metrics Collection, Services Availability, Centralized Log Management* and *Monitoring and Performance Analytics Support* would additionally improve a system's security.

Security Access Control A critical security factor in cloud-based systems is how stable, verifiable, and secure the interactions between a user and a cloud-application are. For this, a secure authentication practice would address most of the possible issues. Authentication is the process of determining a user's identity. Moreover, authorization practices provide access control for systems by checking if a user's credentials match the credentials of an authorized user or in a data authentication server. Also, it assures secure systems, secure processes and enterprise information security [OWA21a]. There are a number of ways authentication can be achieved. At the level of deployment architectures, one strongly recommended practice is the *SSL Protocol-Based Authentication* [RKH14] in which a cryptographic protocol (SSL/TLS) encrypts the data that is exchanged between a web server and a user [OWA21c] and provides means for authentication of the connection. An alternative practice is *Token-Based Authentication* [Okt21], a protocol which allows users to verify their identity, and in return receive a unique access token for a specified time period. A similar practice but without granting tokens for a limited time period is *API Keys* based authentication [Goo21], which utilizes a unique identifier to authenticate a user or a calling program to an API. However, they are typically used to authenticate a project with the API rather than a human user. A less secure practice and not recommended for security critical interactions is *Plaintext Authentication* (or Shared Secret Based Authentication), where the user name and password are submitted to the server in plaintext, being easily visible in any intermediate router on the Internet. An authentication practice that can be implemented additionally is the *Single Sign-On (SSO)* [aut21]. This is a method that can allow users to log into one application and gain access to multiple applications. The goal of SSO is to make it unnecessary for users to have numerous kinds of credentials also benefits them because it allows them to log-off from all system components that use SSO with a single request. In this way, SSO can enable users to improve passwords by getting rid of the need to remember and use them for every single application, offering the best combination of simplicity and security for users.

Security Traffic Control Controlling incoming and outgoing traffic in a system can significantly improve the overall security. Two common practices in this field are *Ingress* and *Egress Traffic Control*. Optimally, a system should fully support both practices. *Egress Traffic Control* [The21] refers to traffic that exits a network boundary, while *Ingress Traffic Control* [Kub21] refers to traffic that enters the boundary of a network. The ability to control what is entering a system is of significant importance for security assurance, since it can prevent possible attacks from outside of the network, where many possible attacks originate. Furthermore, it is important to reduce the vulnerability as much as possible and prevent the attackers from using a cluster for further attacks on external services or systems outside of the cluster. This requires securing control of egress traffic. Both practices can be specified by security rules implementing security groups [AWS21] that act as a virtual firewall for a system.

9.5 Ground Truth Calculations for the Study

In this section, we report for each of the ADDs from Section 9.4 how the ground truth data is calculated based on manual assessment whether each of the relevant practices is either Supported (**S**), Partially Supported (**P**), or Not-Supported (**N**) in Table 9.2. Following the information

taken from the description of the impacts of the various decision options in Section 9.4, we combined the outcome of all decision options to derive an ordinal assessment on how well the decision as a whole is supported in each model, using the ordinal scale in Section 9.3.3. This was done according to best practices documented in literature and experts assessment. For instance, following the ordinal scale the assessment for the model S4 is ∼: serious flaws in the security design, but substantial support can already be found in the system, since the practices *ServicesAvailability Bugsand Performance Management* and *ApplicationMonitoring* are not supported. The ordinal results of assessments are reported in the Assessments rows of Table 9.2.

Security Observability																					
	S1	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**	**S10**	**S11**	**S12**	**S13**	**S14**	**S15**	**S16**	**S17**	**S18**	**S19**	**S20**	**S21**
Server Monitoring	S	S	P	S	S	S	S	S	S	S	N	S	S	S	P	S	N	S	N	S	S
Application Monitoring	N	N	N	N	S	N	S	P	S	N	N	N	S	N	P	N	N	S	S	S	N
System Metrics Collection	N	N	N	S	P	N	S	P	S	S	N	S	S	S	S	N	N	N	S	N	S
Centralized Log Management	N	N	N	S	S	N	N	N	S	N	N	S	S	S	S	S	N	S	N	N	N
Services Availability	N	N	N	N	N	N	N	N	S	N	N	N	S	N	N	S	N	S	N	N	N
Bugs and Performance Management	N	S	P	N	N	N	N	N	S	N	N	N	S	S	S	N	N	N	N	N	N
Assessments	-	∼	∼	∼	+	∼	+	o	++	o	--	∼	++	∼	-	∼	–	+	-	+	∼
Security Access Control																					
	S1	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**	**S10**	**S11**	**S12**	**S13**	**S14**	**S15**	**S16**	**S17**	**S18**	**S19**	**S20**	**S21**
SSL Protocol-Based Authentication	N	N	N	N	P	N	N	S	N	N	N	N	N	S	S	N	N	N	N	S	N
Token-Based Authentication	S	N	N	N	N	S	N	N	S	N	N	N	N	N	N	N	N	N	N	N	N
Plaintext Authentication	N	N	S	S	N	N	S	N	N	N	N	N	N	N	N	N	S	N	N	N	S
API Keys	N	S	N	N	N	N	N	N	N	N	N	N	S	N	N	N	N	N	S	N	N
Single Sign-On (SSO)	N	N	N	N	N	N	N	N	S	N	N	N	N	N	S	N	N	N	N	S	N
Assessments	+	∼	-	-	∼	+	-	+	++	--	--	--	∼	+	++	--	-	--	∼	++	-
Security Traffic Control																					
	S1	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**	**S10**	**S11**	**S12**	**S13**	**S14**	**S15**	**S16**	**S17**	**S18**	**S19**	**S20**	**S21**
Ingress Traffic Control	S	P	N	N	S	S	N	S	S	N	N	S	P	N	P	N	S	S	N	P	S
Egress Traffic Control	S	N	N	N	P	N	N	N	P	N	N	S	P	N	N	N	N	S	N	N	S
Assessments	++	-	--	--	+	∼	--	∼	+	--	--	++	-	--	-	--	∼	++	--	-	++

Table 9.2: Ground Truth Data

Following the argumentation, for the *Security Observability* related practices, we can derive the following scoring scheme for our ground truth assessment of this decision:

- ++: All server nodes support *Server Monitoring*, *Application Monitoring*, *Centralized Log Management*, *System Metrics Collection*, *Services Availability* and *Monitoring and Performance Analytics*.
- +: All server nodes support *Server Monitoring* and *Application Monitoring* and one or more of the practices *Centralized Log Management*, *System Metrics Collection*, *Services Availability* and *Monitoring and Performance Analytics* is supported.

- $\sim$: The majority of the server nodes support *Server Monitoring* and *Application Monitoring* and one or more of the practices *Centralized Log Management*, *System Metrics Collection*, *Services Availability* and *Monitoring and Performance Analytics* is supported.
- $-$: Some of the server nodes support *Server Monitoring* and/or *Application Monitoring*.
- $--$: None of the server nodes support monitoring.

From the argumentation for the *Security Access Control* decision, we can derive the following scoring scheme for our ground truth assessment:

- $++$: All server nodes support *SSL-Based Authentication* or *Token-Based Authentication* and *Single Sign-On authentication*.
- $+$: All server nodes support *Token-Based Authentication* or *SSL-Based Authentication*.
- $\sim$: All server nodes are authenticated and some of those only support *API Keys-Based Authentication*.
- $-$: All server nodes are authenticated and some of those only support *Plaintext-Based Authentication*.
- $--$: None of the server nodes support authentication.

Finally, from the argumentation for the *Security Traffic Control* decision, we can derive the following scoring scheme for our ground truth assessment:

- $++$: All server nodes support *Ingress Traffic Control* and *Egress Traffic Control*.
- $+$: All server nodes support *Ingress Traffic Control* and *Egress Traffic Control* to the majority of system nodes.
- $\sim$: All server nodes support *Ingress Traffic Control* and *Egress Traffic Control* not to the majority of system nodes.
- $-$: The majority of server nodes support *Ingress Traffic Control*.
- $--$: No traffic control is supported.

9.6 Metrics

In this section, we describe the metrics we have hypothesized for each of the decisions described in Section 9.4. They are deliberately rather simple, as to represent each decision point in our design decisions. The metrics, are a continuous value with range from 0 to 1, with 1 representing the optimal case where a set of patterns is fully supported, and 0 the worst-case scenario where it is completely absent, except *Plaintext Authentication utilization metric* in which the scale is reversed in comparison to the others, because here we detect the presence of an anti-pattern: the optimal result of our metrics is 0, and 1 is the worst-case result.. Using the model computed in the ordinal regression analysis below, we then provide more complex metrics per decision in section 9.4.

9.6.1 Metrics for the Security Observability Decision

Server Monitoring metric (SEM). This metric returns the proportion of *Server Nodes* that support server monitoring.

$$SEM = \frac{\textit{Server Monitoring Support}}{\textit{Number of Server Nodes}}$$

Application Monitoring Support metric (AMS). This metric measures the proportion of servers that support *Application Monitoring*.

$$AMS = \frac{\textit{Number of Application Monitoring Links}}{\textit{Number of Server Nodes}}$$

System Metrics Collection Support metric (SMC). This metric measures the proportion of servers that support *System Metrics Collection* tools.

$$SMC = \frac{\textit{Number of System Metrics Collection Links}}{\textit{Number of Server Nodes}}$$

Centralized Log Management Support metric (CLM). This metric measures the proportion of servers that support *Centralized Log Management* tools.

$$CLM = \frac{\textit{Number of Centralized Log Management Links}}{\textit{Number of Server Nodes}}$$

Service Availability Support metric (SAS). This metric measures the proportion of servers that support *Service Availability* tools.

$$SAS = \frac{\textit{Number of Service Availability Links}}{\textit{Number of Server Nodes}}$$

Monitoring and Performance Analytics Support metric (PAS). This metric measures the proportion of servers that support *Monitoring and Performance Analytics* tools.

$$PAS = \frac{\textit{Number of Monitoring and Performance Analytics Links}}{\textit{Number of Server Nodes}}$$

9.6.2 Metrics for Security Access Control Decision

SSL Protocol-based Authentication utilization metric (SSLA). We defined this metric to measure the proportion of servers that support *SSL Protocol-based Authentication*.

$$SSLA = \frac{\textit{SSL Protocol-based Authentication Support}}{\textit{Number of Server Nodes}}$$

Token-Based Authentication utilization metric (TBA). This metric measures the proportion of servers that support *Token-Based Authentication*.

$$TBA = \frac{\textit{Token-Based Authentication Support}}{\textit{Number of Server Nodes}}$$

API Keys utilization metric (API). This metric measures the proportion of servers that support *API Keys*.

$$API = \frac{\textit{API Keys Support}}{\textit{Number of Server Nodes}}$$

Plaintext Authentication utilization metric (PLA). This metric measures the proportion of servers that support *Plaintext Authentication*.

$$PLA = \frac{\textit{Plaintext Authentication Support}}{\textit{Number of Server Nodes}}$$

Single Sign-On Authentication utilization metric (SSO). This metric measures the proportion of servers that support *Single Sign-On Authentication*.

$$SSO = \frac{\textit{Single Sign-On Authentication Support}}{\textit{Number of Server Nodes}}$$

9.6.3 Metrics for Security Traffic Control Decision

Ingress Traffic Control utilization metric (ING). We defined this metric to measure the proportion of servers that support *Ingress Traffic Control*.

$$ING = \frac{\textit{Ingress Traffic Control Support}}{\textit{Number of Server Nodes}}$$

Egress Traffic Control utilization metric (EGR). We defined this metric to measure the proportion of servers that support *Egress Traffic Control*.

$$EGR = \frac{\textit{Egress Traffic Control Support}}{\textit{Number of Server Nodes}}$$

9.7 Evaluation of our Approach

In this section, we present and discuss the results of the metrics calculations for our models as well as the results of the ordinal regression analysis. The metrics calculations for each model per each decision metric are presented in Table 9.3. The dependent outcome variables are the ground truth assessments for each decision, as described in Section 9.5 and summarized in Table 9.2. The metrics defined in Section 9.6 and summarized in Table 9.3 are used as the independent predictor variables. The ground truth assessments are ordinal variables, while all the independent variables are measured on a scale from 0.0 to 1.0. The objective of the analysis is to predict the likelihood of the dependent outcome variable for each of the decisions by using the relevant metrics for each decision.

Table 9.3: Metrics Calculation Results

Metrics	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20	S21
Security Observability																					
SEM	1.00	1.00	0.75	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	1.00	1.00	1.00	0.40	1.00	0.00	1.00	0.00	1.00	1.00
APM	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.66	1.00	0.00	0.00	0.00	1.00	0.00	0.40	0.00	0.00	1.00	1.00	1.00	0.00
SMC	0.00	0.00	0.00	1.00	0.66	1.00	1.00	0.33	1.00	1.00	0.00	1.00	1.00	1.00	1.00	0.00	0.00	0.00	0.00	1.00	1.00
CLM	0.00	0.00	0.00	1.00	1.00	0.66	0.00	0.00	1.00	0.00	0.00	1.00	1.00	1.00	1.00	1.00	0.00	1.00	0.00	0.00	0.00
SAS	0.00	0.00	0.00	0.00	0.00	0.33	0.00	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00	1.00	0.00	0.00	0.00
PAS	0.75	1.00	0.75	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
Security Access Control																					
SSLA	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	1.00	0.00
TBA	1.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
API	0.00	1.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
PLA	0.00	0.00	1.00	1.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00
SSO	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	1.00	0.00
Security Traffic Control																					
ING	1.00	0.75	0.00	0.00	1.00	1.00	0.00	1.00	1.00	0.00	0.00	1.00	0.33	0.00	0.80	0.00	1.00	1.00	0.00	0.50	1.00
EGR	1.00	0.00	0.00	0.00	0.66	0.00	0.00	0.00	0.66	0.00	0.00	1.00	0.33	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00

Each resulting regression model consists of a *baseline intercept* and the independent variables multiplied by *coefficients*. There are different intercepts for each of the value transitions of the dependent variable ($\geq$ [-]: *serious flaws in the security design, but initial support can already be found in the system*, $\geq$[$\sim$]: *serious flaws in the security design, but substantial support can already be found in the system*, $\geq$ [+]: *well supported, but aspects of the solution could be improved*, $\geq$ [++]: *very well supported*), while the coefficients reflect the impact of each independent variable on the outcome.

The statistical significance of each regression model is assessed by the p-value; the smaller the p-value, the stronger the model is. A p-value smaller than 0.05 is generally considered statistically significant [MC]. In Table 9.4, we report the p-values for the resulting models, which in all cases are very low, indicating that the sets of metrics we have defined are able to predict the ground truth assessment for each decision with a high level of accuracy.

Often, the C-index, which is also called concordance index and is equivalent to the area under the Receiver Operating Characteristic (ROC) curve, is reported in the statistical literature as a measure of the predictive power of ordinal regression models [APW+11]. The C-index is a metric to evaluate the predictions made by an algorithm. Values over 0.7 indicate a good model, whereas values over 0.8 indicate a strong model. A value of 1 means that the model perfectly predicts those group members who will experience a certain outcome and those who will not. As Table 9.4 shows, the values of C-indexes are generally larger than 0.8 indicating a good enough model for predicting the outcomes of individuals. Harrell [FEH15] suggests bootstrapping as a method for obtaining nearly unbiased estimates of a model's future performance based on re-sampling. We used lrm's *validate* function to perform bootstrapping and calculate the bias-corrected C-index.

Table 9.4: Regression Analysis Results

Intercepts/Coefficients	*Value*
Security Observability	
Intercept: ≥ [-]	*-1.0154*
Intercept: ≥ [∼]	*-4.0071*
Intercept: ≥ [+]	*-7.9753*
Intercept: ≥ [++]	*-10.956*
Metric Coefficient (SEM)	*4.4224*
Metric Coefficient (APM)	*2.7670*
Metric Coefficient (SMC)	*1.7050*
Metric Coefficient (CLM)	*0.2630*
Metric Coefficient (SAS)	*2.1536*
Metric Coefficient (PAS)	*0.3730*
Model p-value	*1.153036e-06*
Model C-index (original)	*0.929*
Model C-index (bias-corrected)	*0.8934091*
Security Access Control	
Intercept: ≥ [-]	*0.1688*
Intercept: ≥ [∼]	*-2.0267*
Intercept: ≥ [+]	*-3.6306*
Intercept: ≥ [++]	*-6.6639*
Metric Coefficient (SSLA)	*4.3872*
Metric Coefficient (TBA)	*4.4533*
Metric Coefficient (API)	*1.2695*
Metric Coefficient (PLA)	*2.5834*
Metric Coefficient (SSO)	*3.8701*
Model p-value	*3.871694e-07*
Model C-index (original)	*0.951*
Model C-index (bias-corrected)	*0.9456178*
Security Traffic Control	
Intercept: ≥ [-]	*-29.792*
Intercept: ≥ [∼]	*-69.340*
Intercept: ≥ [+]	*-90.643*
Intercept: ≥ [++]	*-120.88*
Metric Coefficient (ING)	*77.7199*
Metric Coefficient (EGR)	*52.539*
Model p-value	*2.031708e-14*
Model C-index (original)	*0.904*
Model C-index (bias-corrected)	*0.8827108*

9.8 Discussion

9.8.1 Discussion of Research Questions

To answer **RQ3.9** and **RQ3.10**, we proposed a set of generic, technology-independent metrics for each IaC decision, and to each decision option corresponds at least one metric. We established the ground truth to objectively assessed how well patterns and/or practices are supported in each model, and extrapolated this to how well the broader decision is supported. We defined a set of generic, technology independent metrics to automatically and numerically assess each pattern's implementation in each model, and performed an ordinal regression analysis using these metrics as independent variables to predict the ground truth assessment. Our results show that every set of decision-related metrics can predict with high accuracy our objectively evaluated assessment.

This suggests that an automatic metrics-based assessment of a system's conformance to ADD's options is possible with a high degree of confidence.

Here, we make the assumption that the infrastructure code can be mapped to the models used in our work. For enabling this, we used rather simplistic modeling means, which can easily be mapped from a specific code to the models

Regarding **RQ3.11**, we can assess that our deployment meta-model has no need for major extensions and is easy to map to existing modeling practices. More specifically, in order to fully model our evaluation model set, we needed to introduce 13 device type nodes and 11 execution environment nodes types such as *Cloud Server* and *Virtual Machine* respectively, and 6 deployment relation types and 4 deployment node relations. Furthermore, we also introduced a deployment node meta-model to cover all the additional nodes of our decisions, such as *Centralized Log Management*, which is a subclass of node type. The decisions in *Security Observability* require modeling several elements such as the *Web Server*, *Container*, and *Cloud Server* nodes types and technology-related connector types (e.g. *RESTful HTTP*) as well as deployment-related connector types (e.g. *Runs on, Deployed in Container*). For the *Security Access Control* and *Security Traffic Control* decisions, we have introduced additional attributes in the system nodes (e.g. in *Web Server*) to specify whether an authentication practice is supported or not, and which specific practice is used (e.g. *SSL Protocol-Based Authentication*). Based on these considerations, we can assess that our decisions and the corresponding model elements can easily be extracted automatically, e.g. with an approach as described in Section 9.3.5.

9.8.2 Threats to Validity

We mostly relied on third-party systems as the basis for our study in order to increase the internal validity and thus avoid distorting the system composition and structure. It is possible that our search procedures have resulted in some kind of unconscious exclusion of certain sources; we have mitigated this by putting together a team of authors with years of experience in this field and doing a very general and broad search. As our search was not exhaustive and most of the systems we found were created for demonstration purposes and they were relatively small in size, meaning that some potential architectural elements were not included in our metamodel. Moreover, this contains a potential threat to the external validity of generalization to other, more complex systems. However, we are confident that the documented systems represent a representative cross-section of current practices in this area. Another potential risk is the fact that the system variants were developed by the team of authors themselves. However, this was done in accordance with the best practices documented in the literature. We made sure to only change certain aspects in one variant and to keep all other aspects stable.

Another possible source of compromising internal validity is the modeling process. The team of authors has considerable experience with similar methodologies, and the models of the systems have been repeatedly and independently cross-checked, but the possibility of some interpretive bias remains: other researchers might have coded or modeled differently, resulting in different models. Since our aim was only to find a model that could specify all observed phenomena, and this was achieved, we do not consider this risk to be particularly problematic for our study. The assessment of the basic truth could also be interpreted differently by different practitioners. The individual metrics that are used to evaluate the presence of the individual patterns were

deliberately kept as simple as possible in order to avoid false positive results and to enable an evaluation that is independent of technology.

9.9 Conclusions and Future Work

In this chapter we have studied how far it is possible to develop a method to automatically assess security practices in architectural design decisions in the scope of an IaC deployment model. We have shown that this is possible for IaC security-related decisions that contain patterns and practices as decision options. In our approach, we modeled the key aspects of the decision options using a minimal set of deployment model elements, which means that it is possible to automatically extract them from the IaC scripts. Then we defined a number of metrics to cover all the possible decision options and utilized the generated models as a ground truth. We then used statistical methods (ordinal regression analysis) to derive a prediction model. The results show that our metrics set can predict the ground truth assessment with a high level of accuracy.

So far, for security aspects of IaC deployment models, no generic, technology-independent metrics have been studied in depth. According to the discussion in Section 9.2, there are studies which are focused on quality assurance of IaC systems, but none specifically address security critical measures. Furthermore, our approach treats deployment architectures as set of nodes and connectors factoring out the technologies used, which is not the case for other studies. Our approach's goal is a continuous assessment, considering the impact of continuous delivery practices, in which the metrics are assessed in a continuous manner, indicating improvements, stability, and security of deployment architecture conformance.

As future work, we plan to study more decisions and related metrics, and to create a larger data set, thus better supporting tasks such as early architecture assessment in a project.

Part IV

Architecture Refactoring

10 Semi-Automatic Feedback for Microservice Architecture Conformance

With the architectural design decision outlined in Chapter 6, we devised an approach to automate architecting in the ongoing development of microservice-based systems. Our method provides the basis for automated architecture reconstruction, conformity assessment to patterns and practices, and detection and localization of potential violations. Subsequently, architects are presented with actionable options to enhance architecture conformity within a continuous feedback loop. The aim is to bolster architecting within the framework of continuous delivery practices, where architecture violations are continually scrutinized, and remedial options are persistently recommended.

10.1 Introduction

Microservices are one of many service-based architecture decomposition approaches (see e.g. [PJ16, PW09, Ric17, ZGK+07]). Microservices should not share their data with other services, can be highly polyglot, and communicate via message-based remote APIs in a loosely coupled fashion. This enables their independent deployment in lightweight containers or other virtualized environments, as well as the rapid evolution of individual microservices independently of one another. These features make microservices ideal for modern DevOps practices (see e.g. [Zim17, PZA+17]).

Many architectural patterns and other recommended "best practices" for microservices have been published in the literature [Ric17, ZSZ+19, Sko19]. So far little attention has been paid to providing usable means to enforce these practices during the evolution of microservice-based systems. This is problematic as it is hard to handle conformance manually, especially in large and/or complex microservice architectures. Best practices have dependencies among each other, meaning that improvement of one practice can lead to issues with another one. Many other system architecture and implementation requirements influence the architectures in ways that might lead to unwanted or accidental violations of microservice best practices, too. Finally, in the context of continuous delivery and DevOps practices, it is to be expected that the architecture will change rapidly and often without central coordination.

This study focuses on providing a set of actionable solutions to violations of loose coupling related microservice best practices. In particular, we investigate three major Architectural Design Decisions (ADD) related to microservice coupling: *Persistent Data Storage of Services* related to data sharing via shared data storage; *Service Interconnections* related to the effects of intermediate system components, such as API gateways, as well as of asynchronous integration; and *Dependencies through Shared Services* related to direct, transitive, and cyclic dependencies

between individual microservices. These decisions have been modeled based on an empirical study of existing best practices and patterns used by practitioners from our prior work [NZP+20a]

To address the outlined challenges, we propose a novel architecture refactoring approach that is specific for architectural design in the context of the microservice ADDs, and uses the empirically validated metrics proposed in our prior work [NZP+20a]. These metrics enable us to study, for each of the above named ADDs, precisely how much a microservice architecture model conforms to favored or less favored design options. For each possible design option in the ADDs, we propose to systematically specify each possible violation. Based on those specifications, we propose automated violation detection algorithms. From the combination of possible ADD options, the chosen option, possible violations, and the detected violations, in each design situation we can calculate all possible next decision options by applying possible solutions to the violations. This leads to a search tree of possible next architecture design models, which we each assess using our metrics. With this we can compare architecture conformance of the current design and possible refactorings, and suggest to an architect all possible improvements in the three ADDs. Please note that this approach is designed to be continuously applicable during each run of a continuous delivery pipeline. This work aims to study the following research questions:

- **RQ4.1** What are the possible architecture violations related to the above-mentioned coupling-related architectural design decisions and how can they be automatically detected?
- **RQ4.2** How can architects be guided in fixing those violations in a continuous feedback loop, while retaining enough flexibility for architect's to chose between possible options, e.g. because other architecture trade-offs need to be considered?

To evaluate our approach we utilized a set of 27 models based on microservice-based systems originally created or described by practitioners (see Table 3.1) as our main data set. We implemented the automated violation detection and refactoring algorithms to detect the possible violations and to generate all the possible fixes for addressing each violation. We then calculate our metrics on coupling in microservices to judge the improvements compared with the initial version. Our result is that in at most 4 refactoring steps, each of the violations found in the 27 models can be fully resolved leading to optimal metric values, usually with many suggested optimal models provided as options for architects to choose from.

This chapter is structured as follows: In Section 10.2 we explain the decisions. We also explain the related patterns and practices, as well as the corresponding metrics as the background of our work. Section 10.3 discusses and compares to related work. Next, we describe the research methods and the tools we have applied in our study in Section 10.4. We then describe the approach details in Section 10.5. In Section 10.6 we explain the evaluation process of our work. Section 10.7 discusses the RQs regarding the evaluation results. In Section 10.8, we then analyse the threats to validity. Finally, in Section 10.9 we draw conclusions and discuss future work.

10.2 Background

In this section, we briefly introduce the three coupling-related decisions, their decision options (i.e. the possible patterns and practices that can be chosen). This information comes from two of

our prior works: a) The decisions have been modeled based on an empirical study of existing best practices and patterns by practitioners [NZP+20a, NZP+20b]. This study also contained a detailed analysis of possible decision drivers (forces, quality attributes) of the decision options. b) We have defined and empirically validated metrics to assess, for a given system model, how well it conforms to the patterns and best practices modeled in our decision model. Based on the reported positive and negative decision outcomes on the mentioned decision drivers, we could further assess which of the options are more or less favored in the microservice practitioner literature. For evaluation we used 27 microservice component architecture models, summarized in Table 3.1 and described in Section 10.4.

10.2.1 Decisions

Decision: Persistent Data Storage of Services

This decision is about how persistent data storage is handled for services, if any is needed. The following decision options can be chosen: *No Persistent Data Storage* is applicable only for services whose functions are performed on transient data. The most recommended option is *Database per Service* pattern [Ric17], in which each service has its own database and manages its own data independently. Another option, is to use a *Shared Database* [Ric17]: two or more services read to and write from a common database. This option has two alternatives: *Data Shared via Shared Database* in which multiple services share the same table, resulting in a strongly coupled system. In contrast in the *Database Shared but no Data Sharing* option, each service writes to and reads from its own tables, which has a lesser impact on coupling.

In our previous work, we have empirically defined two metrics that can be used to assess conformance to each of the decision options:

- *Database Type Utilization* to measure the proportion of services that are using individual databases.
- *Shared Database Interactions* to measure the proportion of interconnections via a *Shared Database* among the *total number of service interconnections*.

Decision: Service Interconnections

Another important aspect in microservices is how the services communicate between each other. The decision is about how tightly service are coupled via their interconnections. *No Service Interconnection* is an optimal option but in reality this is not applicable. One other option is *Synchronous Service Interconnections* which is usually not the favored option in microservice systems. A number of asynchronous alternative options exist. One of these is *Asynchronous Direct Interconnections*, in which all the services communicate asynchronously via direct invocations. Another option is asynchronous communication through intermediary components. These can be *Pub/Sub Interconnections* [HW03a], in which services publish and subscribe to events between each other, maybe combined with *Event Sourcing*; *Messaging Interconnections* [HW03a], in which services produce and consume messages that are stored in a message broker; *Asynchronous Interconnections via API Gateway* [Ric17], where services route asynchronous invocations via

the API Gateway; *Shared Database Interconnection*, in which services interact via a shared database—every communication that is happening in this way is considered as asynchronous. Please note that the last option is beneficial over synchronous invocations in this decision, but leads to other problems, including shared-database interactions from the previous decision.

For this decision too we have empirically defined two metrics that can be used to assess conformance to each of the decision options:

- *Asynchronous Communication Utilization* to measure the proportion of asynchronous service interconnections in the system.
- *Service Interactions via Intermediary Components* to measure the proportion of service interconnections via asynchronous relay architectures, such as *Message Brokers*, *Publish/Subscribe*, or *Stream Processing*.

Decision: Dependencies through Shared Services

Optimally, in a microservice-based system, services should not share other services all together at least not in a strongly coupled fashion. There are many patterns that are related to system structures avoiding service sharing. Especially in large scale systems, service sharing can lead to additional issues such as a chain of transitive dependencies between services and severe maintenance issues. We have identified four decision options for this decision: First, the optimal case, the might be *No Shared Services*. There are three cases containing some service sharing: *Directly Shared Service* in which two or more services are directly connected to other service(s); *Transitively Shared Service* in which a service is linked to other services via at least one intermediary service creating a transitive chain; and *Cyclic Dependency* [GM14] in which services create a direct or transitive path that leads back to the initial service. Cyclic dependencies are considered as highly problematic since the services can no longer be changed, understood, or tested in isolation.

For this decision too we have empirically defined three metrics that can be used to assess conformance to each of the decision options:

- *Direct Service Sharing* to measure the proportion of directly shared elements in the system.
- *Transitively Shared Services* to measure the proportion of transitively shared elements in the system.
- *Cyclic Dependencies Detection* to detect the presence of *at least one* cyclic dependency in the system.

10.3 Related Work

Microservice best practices have been widely examined in various studies. A collection of microservice patterns and practices has been published by Richardson [Ric17] and another collection of practices related to event-driven microservice architectures has been published by Skowronski [Sko19]. Zimmermann et al. [ZSZ+19] introduced patterns related to microservices

APIs. Fowler and Lewis [LF04] have discussed microservice fundamentals and best practices. Pahl and Jamshidi [PJ16] have summarized many of those in a mapping study. Microservice "bad smells" have been studied by Taibi and Lenarduzzi [TL18], which correspond to violations in our work.

There is a number of studies that focus on techniques for detecting design or architecture smells, which are considered as violations in our case, but most of them are not specific to the microservices domain. A catalog of architectural bad smells using a format has been published in Garcia et al.'s [GPEM09a, GPEM09b] studies. These studies also propose possible techniques for identification of these architecture smells. Le et al. [LLSM18] examined the relations between smells and a project's issues. They further examined the detection of multiple types of architecture smells. A number of detection strategies that take advantage of metrics-based rules for detecting design flaws have been presented by Marinescu [Mar04]. Garcia et al. [GPM+11] present a machine learning-based technique for recovering an architectural view containing a system's components and connectors, which aims at detecting architecture drift or erosion. A prototypical tool for architecture recovery of microservice-based systems (MicroART) is presented by Granchelli et al. [GCD+17]. Alshuqayran et al. [AAE18b] suggest a microservice architecture recovery approach based on a meta-model and rules for mapping artifacts to it. A multivocal literature review, focused on identifying architectural smells for independent deployability, horizontal scalability, fault isolation and decentralisation of microservices, as well as suggesting refactorings to resolve them, is presented by Brogi et al. [NSZB19].

Although these works study various aspects of architecture violation detection, and some of them investigate aspects related to the microservice domain, none covers detecting and addressing coupling-related violations in a microservice context. In contrast, our work investigates in detail coupling-related aspects such as data persistence, communication types, and shared service dependencies. As a direct benefit of this, we expect that, in the context of loose coupling, our work produces more accurate detections of decision-specific violations and more targeted suggestions for fixes than those other works possibly could. As a downside, our work requires a model in which the component and connector roles in a microservice architecture have been modeled (as for instance done with stereotypes in the model introduced in Figure 10.3). That is, our work requires additional insight into a system's architecture, and some effort in encoding the corresponding models; however, this knowledge is at a relatively high level of abstraction and the resulting models are not impacted by changes in service implementation. We are currently working on a semi-automatic approach for architecture reconstruction and modelling that relies on reusable code abstractions and is thus suitable for complex systems with short delivery cycles.

10.4 Research and Modeling Methods

In this section, we summarize the main research and modeling methods applied in our study. For reproducibility, all the code and models produced in this study will be made available online, as an open access data set in a long-term archive[1].

[1] https://doi.org/10.5281/zenodo.4491583

10.4.1 Research Method

Figure 10.2 shows the research steps of this study. In Section 10.2 we have already explained in detail the architectural decisions and the model-based metrics on which this study is based. In Section 10.5 we present a) precise definitions and algorithms for the detection of possible violations for each decision option, and b) precise definitions and algorithms for the possible fixes for each violation.

To evaluate our approach, we have applied it on a dataset of 27 models, summarized in Table 3.1. This dataset comprises microservice-based systems from 9 independent sources, developed by practitioners and published in public repositories and practitioners' blogs. We assume that these systems are, or reflect, real-world practical examples of microservice architectures. However, as many are open-source systems for demonstrating practices or technologies, they are, at most, of medium size and modest complexity. For the specification of our Microservice Component Architectures meta-model and the calculation of all metrics, violation detection, and fixes, we used the Python modeling library CodeableModels described in Chapter 3.7.

Our approach is designed to detect all violations for every model in our data set, and perform all possible suggested architecture refactorings (fixes) to it. This we did recursively, i.e., on the resulting, refactored models for each violation fix, we again performed *all* violation detection algorithms and applied *all* possible refactorings, until either no more violations were detected, or the refactored model was identical to a previous version. In the latter case, this means that it is not possible to fix all violations, since resolving one violation will require introducing other violations. For each of the final models (the 'leaves' of the iteration tree), we assessed pattern conformance through our metrics on microservice coupling, to judge the improvement compared to the original model.

10.4.2 Using the Approach in a Continuous Delivery Pipeline

Figure 10.1 shows the position of our approach in a *delivery pipeline* in which every commit triggers an iterative loop of improvements. We place our approach after initial tests have been run, i.e. where usually code coverage and similar checks are run. In this stage we perform the *metrics calculation process* and if a violation is detected we determine the specific type of violation and provide a set of *fix options*. The *architect* or *developer* can select the optimal fix or to perform no more fixes. This triggers the *automatic architecture refactoring process* and a new version of the system component is generated. The *metrics calculation process* is performed again for the new version to evaluate the improvements. If there is no more violation after the fix process we continue in pipeline process. Of course, the approach can be equally applied to more complex systems with multiple delivery pipelines, for example, by mining docker files or other runtime logs in order to reconstruct the architecture.

10.5 Approach Details

In this section, we first give an overview of the violations and possible fixes we have identified, as well as the algorithms we have developed to detect violations or enact fixes. Also, we give

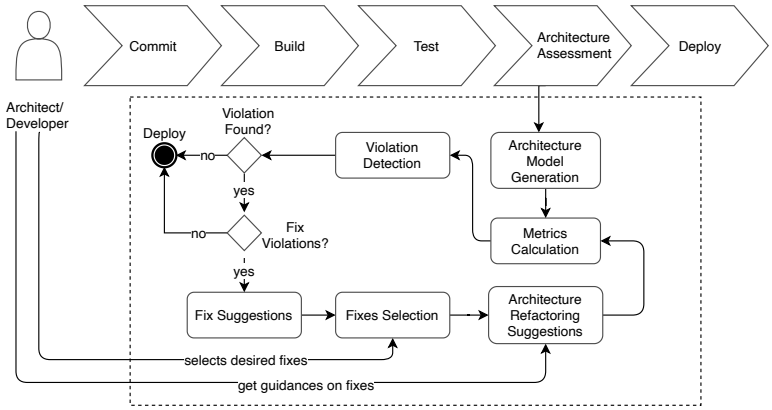

Figure 10.1: Placing of our approach in a delivery pipeline

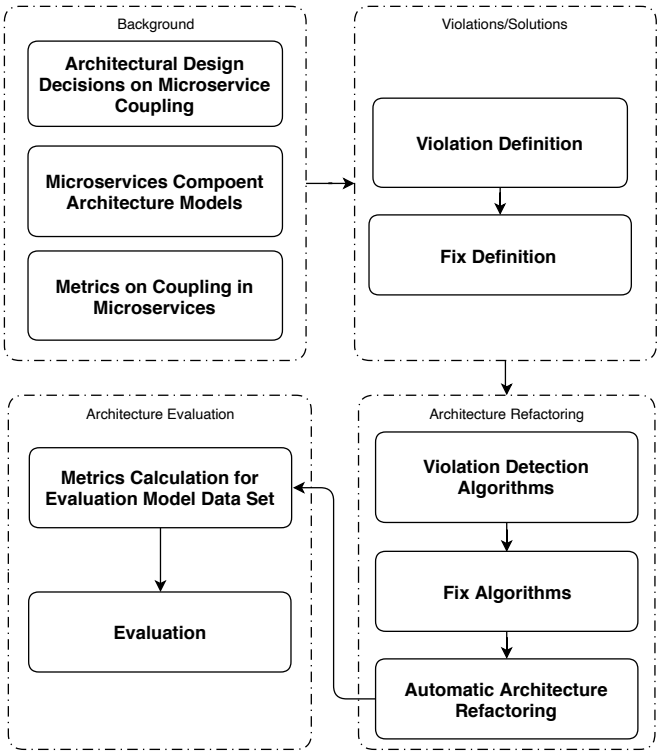

Figure 10.2: Overview diagram of the research method followed in this study

detailed examples from the *Dependencies through Shared Services* decision to illustrate the approach.

We base our violation and fix definitions on the notion of a microservice-based architecture model consisting of a directed components and connectors graph. This can be expressed formally as: A microservice architecture model M is a tuple (CP, CN, CPT, CNT, ST) where:

- CP is a finite set of **component nodes**. The operation $components(M)$ returns all components in M.
- $CN \subseteq CP \times CP$ is an ordered finite set of **connectors**. $connectors(M)$ returns all connectors in M.
- CPT is a set of **component types**. The operation $services(M)$ returns all components of type $service$ in M. The operation $service\_connectors(M)$ returns all connectors of components of type $service$ in M.
- CNT is a set of **connector types**.
- ST is a finite set of **stereotype nodes**. The operation $cp\_stereotypes(CP)$ returns all stereotypes of component CP. The operation $cn\_stereotypes(CN)$ returns all stereotypes of connector CN. Stereotypes can be applied to components to denote their type, such as *Service*, *API Gateway*, etc. Stereotypes can be applied to connectors to denote their type, such as *Read_Data*, *RESTful HTTP*, or *Asynchronous*. Some are specialized with tagged values (details omitted here for space reasons).
- $cp\_annotations : CP \rightarrow \{String\}$ is a function that maps an component to its set of annotations. Annotations are used in our approach (in some of the fixes) to document aspects that need further consideration or maybe manual refactoring.
- $cn\_annotations : CN \rightarrow \{String\}$ is a function that maps a connector to its set of annotations.

Please note that we define many additional model traversal operations not detailed here for space reasons.

10.5.1 Violation Detection

Table 10.1 summarizes the possible violations we have identified for each of the decisions. The table also describes in detail how the algorithms work that we use for detecting the violations in models based on our model definition above. In Algorithm 10.1 we exemplary detail the algorithm for detecting the Directly Shared Services Violation of Decision D3. It returns a list of violations, each described by a set of two services connectors in which two services s_i and s_j share a service s_m. Its sibling for the Transively Shared Services Violation is shown in Algorithm 10.2. This one returns a list of all service sets in which two services s_i and s_j share a service s_m via intermediaries.

Violation	*Violation Detection Algorithm Summary*
D1: Persistent Data Storage of Services	
D1.V1: Services have a shared database, but no data is shared via the shared database	The models are traversed for finding database accesses. Database accesses by more than one service are inspected for the data entities that are read and written. If none of those data entities are shared by two services, this violation is raised. All violating data accesses (including services, databases, and connectors involved) are returned by the detector operation.
D1.V2: Services have a shared database and data is shared via the shared database	The models are traversed for finding database accesses. Database accesses by more than one service are inspected for the data entities that are read and written. If *at least one* of those data entities is shared by *at least two services*, this violation is raised. All violating data accesses (including services, databases, and connectors involved) are returned by the detector operation.
D2: Service Interconnections	
D2.V1: System services communicate synchronously	All service connectors in the model are traversed. If any synchronous connector is encountered, the violation is raised and the list of all synchronous connectors is returned by the detector operation.
D3: Dependencies through Shared Services	
D3.V1: Directly shared services	All services in the model are traversed, and it is checked whether two services share another service via directly linking connectors. If this is the case, a violation is raised. Each pair of shared service connectors that is found is returned by the detector operation.
D3.V2: Transitively shared services	All services in the model are traversed, and it is checked whether two services share another service via transitively linking connectors. Transitive means here via any number of intermediary other services. If this is the case, a violation is raised. Each pair of shared service connectors that is found is returned by the detector operation.
D3.V3: Cyclic Dependency	On the graph of services and connectors in the model we run a cycle detection based on a *depth-first search algorithm*. If we detect a cycle, a violation is raised. All detected cycles are returned as a list of sets of connectors participating in the respective cycle.

Table 10.1: Identified Violations and Violation Detection Algorithms

Algorithm 1: Detect Directly Shared Services Violation

```
input: Model M
output: Set<Tuple>
begin
  violations ← ∅
  for s_m ∈ services(M):
  for s_i ∈ services(M):
  for s_j ∈ services(M):
    if ((s_i, s_m) ∈ service_connectors(M) ∧
    (s_i, s_m) ∈ service_connectors(M)):
      violations ← violations ∪ (s_i, s_m), (s_j, s_m)
  return violations
end
```

Algorithm 2: Detect Transitively Shared Services Violation

```
input: Model M
output: Set<Tuple>
begin
  violations ← ∅
  for s_m ∈ services(M):
  for s_i ∈ services(M):
  for s_j ∈ services(M):
    if (exists_transitive_link(M, s_i, s_m) ∧
    exists_transitive_link(M, s_i, s_m)):
      violations ← violations ∪ (s_i, s_m), (s_j, s_m)
  return violations
end
```

10.5.2 Fixes

Table 10.2 details all the fixes for each identified violation along with a summary of the fix algorithm. Please note that many algorithms can only be applied with default values or approaches fully automatically. Many of them require human review by the architect and sometimes a human decision to be applicable. For example, the architects can be presented with a choice of an intermediary component to use to replace cyclic links or which of a set of transitive connectors should be deleted. That is, our fix approach is intended to be used as guidance to architects in a feedback loop (as illustrated in Figure 10.1), not to replace them.

Please note that some obvious details of the algorithms that are repetitive have been omitted for space reasons. For example, if connectors are replaced, existing stereotypes and annotations on them that are not related to the type of provided replacement must be retained on the new connectors. To illustrate this consider a *RESTful HTTP, Synchronous* connector is replaced with a *RESTful HTTP, Asynchronous* connector. Obviously, the stereotype *Synchronous* changes to *Asynchronous* during the fix, but also it must be ensured that the *RESTful HTTP* annotation is retained.

The Algorithms 10.3–10.6 exemplary present the fixes for Decision D3 and its Violation V1 in detail. They represent the identically named fixes D3.V1.F2–D3.V1.F5 respectively. For explanations of each fix, please study Table 10.2.

Algorithm 3: Remove Connectors of Directly Shared Services

```
input: Model M, Set<Tuple> violation
output: –
begin
  for (s_i, s_m) ∈ violation:
    if (s_i, s_m) ∈ service_connectors(M):
      delete_connector(M, (s_i, s_m))
end
```

Algorithm 4: Remove Connectors of Directly Shared Services

```
input: Model M, Set<Tuple> violation, Component intermediary
output: –
begin
  for (s_i, s_m) ∈ violation:
    add_connector(s_i, intermediary,
      get_applicable_stereotypes(M, (s_i, s_m)))
    add_connector(intermediary, s_m,
      get_applicable_stereotypes(M, (s_i, s_m)))
  delete_connector(M, (s_i, s_m))
end
```

Algorithm 5: Integrated Shared Services into Calling Service

```
input: Model M, Set<Tuple> violation
output: –
begin
  integration_annotations ← ∅
  for (s_i, s_m) ∈ violation:
    integration_annotations ← integration_annotations ∪
      "integrated functionality from: " +
      get_service_name(M, s_i)
```

```
    for connector ∈ incoming_connections(M, s_i):
      change_target(model, connector, s_m)
    delete_service(M, s_i)
  add_annotations(M, s_m, integration_annotations)
end
```

Algorithm 6: Integrated Calling Service into Calling Services

```
input: Model M, Set<Tuple> violation
output: –
begin
  for (s_i, s_m) ∈ violation:
    for connector ∈ outgoing_connections(M, s_m):
      change_source(M, connector, s_i)
    add_annotations(M, s_i,
      {"integrated functionality from: " +
       get_service_name(M, s_m)})
  delete_service(M, s_m)
end
```

10.5.3 Violation Detection and Fixes Example

In Figure 10.3 the model TH1 from Table 3.1 is shown. As an illustrative example, we use it here to demonstrate the *Directly Shared Services Violation (D3.V1)* and possible fixes. In this model the *Payment* service is called directly by services *Passenger Management* and *Drive Management*. Here, the *Payment* service is considered as *shared service* causing the *Directly Shared Services Violation*. It would be triggered in our approach by providing a bad metric value, which would trigger the detailed detection, which would return the $\{$(*Passenger Management, Payment*), (*Passenger Management, Payment*)$\}$ set of tuples. If we run our fix algorithms the resulting model fix suggestions are for instance:

- *Applying Fix D3.V1.F2:* The architect can decide that the connectors or one of them is not really needed or can be replaced by some other manual refactoring. If this is the case, the connectors can get removed by this fix.
- *Applying Fix D3.V1.F3: Payment* service will be disconnected from *Passenger Management* and *Drive Management* services and connected to *API Gateway* (all interactions will be happening via the API Gateway). Alternatively, this fix could also introduce a new *intermediary component* (e.g., Pub/Sub) and all the involved services will be connected to it with *publish* and *subscribe* operations for the data exchange.
- *Applying Fix D3.V1.F4:* The *Passenger Management* and *Drive Management* services can be integrated into *Payment* service.
- *Applying Fix D3.V1.F5:* The *Payment* service can be integrated to *Passenger Management* and *Drive Management* services, if that's possible.

In all these fixes the identified *directly shared services violation* would get fixed.

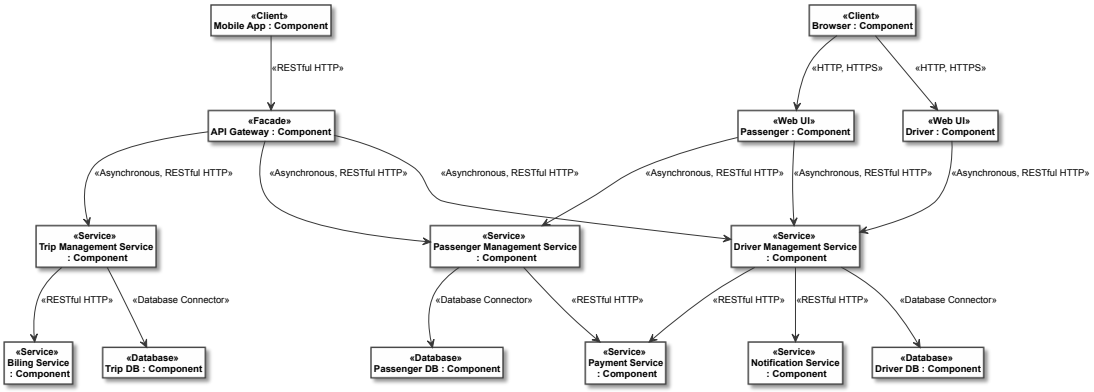

Figure 10.3: Example of an Architecture Component Model of model TH1 in Table 3.1: this architecture violates the Service Interaction (D2.V1) and Dependencies through Shared Services (D3.V1) decisions (cf. Table 10.1).

10.6 Evaluation

For evaluating our work, we have fully implemented our algorithms for detecting violations and performing fixes, as well as generating the set of metrics described in Section 10.2 to measure the improvements and the presence of remaining violations, in our model set. In case multiple violations are present in a model, then the algorithms can be employed iteratively, until all violations have been fully resolved.

As an example, let us illustrate this exhaustive iterative refactoring for the TH1 Model (see Figure 10.3). TH1 violates two of the decisions—"System services communicate synchronously" (D2.V1) fully and "Directly shared services" (D3.V1) partially, as indicated by the two respective measures (0.00 and 0.67 respectively) in Table 10.3. The incremental refactoring process is illustrated in Figure 10.4. At the first iteration step, there are two branches, depending on which violation is dealt with first. The first iteration step results in 7 possible model variants, one for each fix option from Table 10.2. Out of those, the model variant F shows no further violations (i.e., the fix for D2.V1 has also coincidentally fixed violation D3.V1 and thus optimally resolved all violations). In model variant G the violation D3.V1 was also coincidentally fixed, but this fix introduced a new "Services have a shared database and data is shared via the shared database" (D1.V2) violation. Thus, after the first iteration, there are 6 new model variants that still contain a violation.

The second iteration step results in further 18 models. In turn, 4 of the resulting models now exhibit violation D1.V2, requiring a third step to be resolved. At the end of the third step, we have 23 suggested model variants (A1–A2, A3_1–A3_2, B1–B2, B3_1–B3_2, C1–C2, C3_1–C3_2, D1–D2, D3_1–D3_2, E1–E4. F, G1–G2) which all optimally resolve the violations (i.e., scoring 1.00 in our assessment scale). The architect can choose the refactoring sequence, and from among those final optimal model variants, but can also choose to not apply certain fixes, e.g. due to other constraints that are outside of the scope of our study.

For evaluation purposes, we have performed this procedure for *all* 27 system models in

Table 3.1. The resulting number of intermediate models and violation instances per step, and the number of final suggested models with an optimal assessment of 1.00, are given in Table 10.3, along with the initial violations and architecture assessment values for each model. Please note that the metrics reported here are the ones associated to each of the violations; most metrics from Section 10.2 match one-to-one, only D2.V1 has two metrics associated (i.e., both from Section 10.2.1). Please also note that for D2.V1 to be fixed, it is enough that one of the two metrics is optimal (1.00); this is why the models BM3, CO2, and EC2 require no refactoring steps, even though they score less than 1.00 in the other of the two metrics. The metrics measure the degree of the violation, with *1.00* when *no violation exists* and *0.00* where the worst possible option is selected and not even partial conformance is measured. Obviously, the number of steps required to reach optimal models depends on a) the number of the violations present in the initial model and b) on the possible appearance of new violations during the refactoring process (as explained in detail in the TH1 example above). As can be seen in Table 10.3, *all* models are *fully resolved*—i.e., all assessment metrics are 1.00—after *at most* four steps.

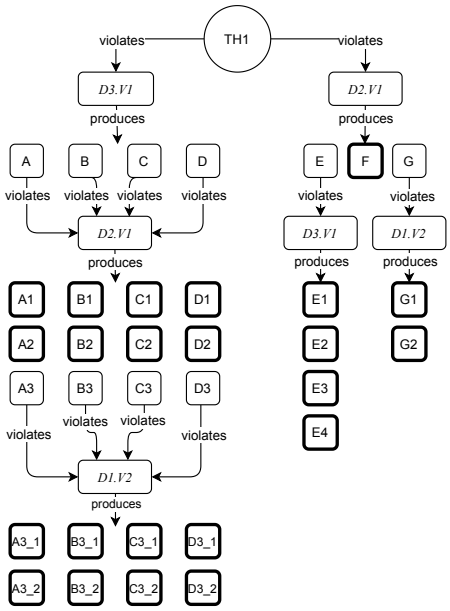

Figure 10.4: Example of an exhaustive iterative application of our approach in the TH1 model. Final (i.e., optimally resolved) resulting models are thickly outlined.

10.7 Discussion of Research Questions

To answer **RQ4.1** we have systematically specified a number of decision-based violations related to each possible decision option, summarized in Table 10.1. As we have empirically shown in our prior work [NZP+20a] that the metrics described in Section 10.2 can reliably distinguish favored or less favored design options, the role of the violation detectors is to find the precise spots in

the models where the violations occur. For each system model in our evaluation dataset it was possible to suggest fixes that bring the architecture to optimal values, meaning that the algorithms have found the right place(s) to apply the fixes.

Regarding **RQ4.2** we defined a number of algorithms addressing every possible violation, with multiple fix options (cf. Table 10.2). If all options are tried out, this results in a search tree of possible architecture models, which can in turn be assessed, using our metrics, to measure improvements to the initial architecture and detect any remaining violations. We have shown (cf. Table 10.3) that an iterative approach of using our algorithms successively, results, within a few steps, in a sufficient variety of possible architecture models that remove all detected violations and ensure pattern conformance of the system architecture. The multiple optimal model variants that result from our approach give architects substantial levels of freedom in their design decisions. As detection is fully automated and human expertise is limited to the fix process, the approach is well suited to be run in a continuous delivery environment, which was one of our research goals.

10.8 Threats to Validity

To increase the internal validity of our approach, throughout our study, as well as in the previous works on which it builds, we have relied on a large number of systems produced by third parties for testing and evaluation (cf. Table 3.1). Likewise, the solutions we propose are derived from best practices and patterns in the published (grey) literature: our work is limited to gathering, systematizing, and applying them to the given set of systems. Any omissions can be added to our model without invalidating the fundamental approach. One possible threat to the internal validity of our algorithms is that they depend on the particular modelling approach we have adopted. However, this approach is by design generic, based on typical component-and-connector models, and should be both capable of dealing with most microservice systems, as well as be easy to extend or adapt if required.

Our approach presently has some limitations: it operates at a relatively high level of abstraction, does not consider any decision parameters other than the metrics evaluating pattern conformance, and limits itself to specific, coupling-related patterns. We consider that these limitations can be addressed in future work, by applying the same fundamental, metrics-based approach to finer-granularity parameters , enriching our decision support model with additional data sources , and adding more patterns to our decision model. In terms of generalizability, we have not considered industrial-scale systems with hundreds of services, but are confident that the fundamental approach is sound. The challenge would be one of adapting our method and increase its ‘intelligence’ so as to target contiguous portions of large-scale systems at a time, so that the generated refactored models would be realistically feasible. The solutions proposed may also not be always optimal, as we aim to present generically applicable solutions. It remains possible that an architect will devise a custom, hybrid solution that optimizes a system in ways that an automated approach cannot. Again, however, this is a matter of extending the present approach with the metrics and the ‘intelligence’ required to model these hybrid solutions.

10.9 Conclusion and Future Work

In this chapter we have presented a set of violations for three ADDs and mapped them to existing, empirically validated metrics judging the outcomes of these decisions. We have defined automatic detectors for these violations, which provide the precise places in a model where the violations occur. Bases on this, we have defined a set of possible fixes for each violation, as part of an approach for ensuring pattern and best practice conformance in microservice-based architectures. We have evaluated our approach on a set of 27 models showing various degrees of pattern violations and architecture complexity, and have shown that our approach is capable of resolving these violations in at most 4 refactoring steps that can largely be automated. As both metric calculation and violation detection are fully automated, the approach can be applied as an additional "architecture assessment test" in a continuous delivery pipeline, as was our goal. Fixes are automated, too, but architects have to decide which of the suggested options should be applied (if any) and sometimes need to provide some input. Thus the approach is still flexible enough to let the architect make architectural design choices.

In our future work, we aim to broaden the set of ADDs and violations included in our approach, enrich it with runtime metrics and other architecture aspects such as deployment environments, and extend our model dataset to include larger and more complex systems. In addition, we hope to experimentally validate our approach by employing it in real-world delivery pipelines as part of a feedback loop.

Violation	*Fix*	*Fix and Fix Algorithm Summary*
D1: Persistent Data Storage of Services		
D1.V1	*D1.V1.F1: Do not fix the violation*	The architect should have the option to *not* fix the violation, e.g. because it is not critical.
	D1.V1.F2: Introduce new service-specific databases and migrate database accesses	Disconnect services from the shared database and introduce a new database per service. Migrate each service-specific database access to the respective service-specific database. The architect needs to check if this fix is possible and, if applied, whether the original database can be deleted after the migration.
	D1.V1.F3: Merge services with shared database into a single service using one database	Merge the services using the same shared database into a single service using that database. The architect needs to check whether such integration is possible and can provide annotations for implementers about the details of the envisaged service integration.
D1.V2	*D1.V2.F1: Do not fix the violation*	The architect should have the option to *not* fix the violation, e.g. because it is not critical.
	D1.V2.F2: Migrate to new database and add consistency mechanism for the shared data	Same as Fix D1.V1.F3. In addition, add consistency mechanism for the data shared between services. The architect needs to select the consistency mechanism applied (e.g., eventual consistency via an event store). Then connectors will be extended with respective stereotypes and exchange of data-related events.
	D1.V2.F3: Merge Services with Shared Database into a single service using one Database	Same as Fix D1.V1.F3.
D2: Service Interconnections		
D2.V1	*D2.V1.F1: Do not fix the violation*	The architect should have the option to *not* fix the violation, e.g. because it is not critical.
	D2.V1.F2: Replace synchronous direct interconnection with asynchronous direct interconnection	Disconnect all the synchronously connected services and connect them using asynchronous connectors with the same stereotypes as the synchronous ones. Delete the synchronous connectors.
	D2.V1.F3: Replace synchronous direct interconnection with interactions via an intermediary component, e.g., API Gateway, Pub/Sub, Message Broker	The architect has to select if an existing intermediary component can be used for the fix, or a new one has to be created. Replace synchronous direct interconnections with asynchronous interconnections via this component. Delete the synchronous connectors.
	D2.V1.F4: Introduce communication by writing to and reading from common databases	The architect has to select if an existing database can be used for the fix, or a new one has to be created. For each synchronous connectors, introduce communication by writing to and reading from this database. Delete the synchronous connectors. Please note, while this fix repairs this violation, it leads to a violation of D1.
D3: Dependencies through Shared Services		
D3.V1	*D3.V1.F1: Do not fix the violation*	The architect should have the option to *not* fix the violation, e.g. because it is not critical.
	D3.V1.F2: Remove connectors of directly shared services	Change the connections between services and remove connectors between the involved services. This fix is only applicable, if a solution makes sense that performs the same functionality without those connectors. This must be judged by a human architect.
	D3.V1.F3: Replace direct links via an intermediary component, e.g., API Gateway, Pub/Sub, Message Broker	The architect has to select if an existing intermediary component can be used for the fix, or a new one has to be created. Replace direct interconnections with asynchronous interconnections via this component. Delete the direct connectors.
	D3.V1.F4: Integrate shared services into calling service	Integrate the responsibility and functionality of the *shared services* (i.e., the services that are called by two or more services) into *calling services* (services that call a shared service), and delete the shared services and any connectors accessing them. Here we add annotations that functionality has been added to the calling service, so that implementers later on can realize this functionality.
	D3.V1.F5: Integrate calling service into shared service	Integrate the responsibility and functionality of the *calling services* into *shared services*, delete the calling service, and rewire their clients directly to the shared services. Here we add annotations that functionality has been added to the shared services, so that implementers later on can realize this functionality. Which functionality goes to which shared service can be further annotated by the architects.
D3.V2	*D3.V2.F1: Do not fix the violation*	The architect should have the option to *not* fix the violation, e.g. because it is not critical.
	D3.V2.F2: Remove connectors of transitively shared services	The architect needs to decide which of the connectors in each transitive link path are safe to delete. It must then be checked that this is enough to break up the transitive sharing. Then: Same as D3.V1.F2 for selected connectors.
	D3.V2.F3: Replace direct links via an intermediary component, e.g., API Gateway, Pub/Sub, Message Broker	As in D3.V2.F2 the architect needs to select connectors, and a check that sharing is broken needs to be performed. Then: Same as D3.V1.F3 for the selected connectors.
	D3.V2.F4: Integrate transitively shared services into a calling service	The architect has to select which services on the transitive link path are to integrated into the calling service. It must then be checked that this is enough to break up the transitive sharing. Then: Same as D3.V1.F4 for the selected services. The
	D3.V2.F5: Integrate transitively calling service into shared service	As in D3.V2.F4 the architect needs to select services, and a check that sharing is broken needs to be performed. Then: Same as D3.V1.F5 for the selected services.
D3.V3	*D3.V3.F1: Do not fix the violation*	The architect should have the option to *not* fix the violation, e.g. because it is not critical.
	D3.V3.F2: Replace cyclic relations via an intermediary component, e.g., API Gateway, Pub/Sub, Message Broker	The architect has to select if an existing intermediary component can be used for the fix, or a new one has to be created. Replace all connectors in a cycle with connectors to this component. Delete the cyclic connectors.
	D3.V3.F3: Remove connectors from a cycle until there is no cycle	The architect selects connectors that can be deleted in a cyclic path. It is then checked whether the cycle is broken by those deletions. Then the selected connectors are deleted. If selected by the architect, the steps from D3.V3.F2 can then be followed to introduce links via an intermediary component instead.
	D3.V3.F4: Integrate all functionality of services involved to a cycle into one service	Integrate the responsibility and functionality of the services in the cycle into one *integration service*, selected by the architect. This can also be a new service, introduced by the architect. Rewire the cyclic services' clients directly to the integration service. Here we add annotations that functionality has been added to the integration service, so that implementers later on can realize this functionality.

Table 10.2: Identified Fixes And Fix Algorithms

Model ID	Initial Model Assessments						Models Generated / Remaining Violation Instances per Refactoring Step				Resulting Suggested (Optimal) Models
	D1.V1	*D1.V2*	*D2.V1*	*D3.V1*	*D3.V2*	*D3.V3*	*Step 1*	*Step 2*	*Step 3*	*Step 4*	
BM1	0.33	1.00	1.00, 0.00	1.00	1.00	1.00	2 / 0	–	–	–	2
BM2	1.00	1.00	0.00, 0.00	1.00	1.00	1.00	3 / 1	2 / 0	–	–	4
BM3	1.00	1.00	0.00, 1.00	1.00	1.00	1.00	–	–	–	–	–
CO1	1.00	1.00	0.00, 0.00	1.00	1.00	1.00	3 / 1	2 / 0	–	–	4
CO2	1.00	1.00	1.00, 0.00	1.00	1.00	1.00	–	–	–	–	–
CO3	0.60	0.00	0.00, 1.00	1.00	1.00	1.00	2 / 0	–	–	–	2
CI1	1.00	1.00	0.00, 0.00	0.75	1.00	1.00	7 / 6	18 / 4	8 / 0	–	23
CI2	1.00	1.00	0.00, 0.00	1.00	1.00	1.00	3 / 1	2 / 0	–	–	4
CI3	1.00	1.00	0.00, 0.00	0.70	1.00	1.00	7 / 6	18 / 4	8 / 0	–	23
CI4	1.00	1.00	0.00, 0.00	1.00	1.00	1.00	3 / 1	2 / 0	–	–	4
EC1	1.00	1.00	0.00, 0.00	1.00	1.00	1.00	3 / 1	2 / 0	–	–	4
EC2	1.00	1.00	1.00, 0.00	1.00	1.00	1.00	–	–	–	–	–
EC3	0.00	0.00	0.00, 1.00	1.00	1.00	1.00	2 / 0	–	–	–	2
ES1	1.00	1.00	0.60, 0.00	0.73	1.00	1.00	7 / 1	2 / 0	–	–	8
ES2	0.00	1.00	0.00, 0.00	0.66	1.00	1.00	9 / 10	26 / 11	28 / 2	4 / 0	45
ES3	0.33	1.00	0.00, 0.00	0.66	1.00	1.00	9 / 10	26 / 11	28 / 2	4 / 0	45
FM1	1.00	1.00	0.00, 0.00	0.38	1.00	1.00	7 / 2	6 / 0	–	–	11
FM2	1.00	1.00	0.00, 1.00	0.70	1.00	1.00	4 / 0	–	–	–	4
FM3	1.00	1.00	0.00, 1.00	1.00	0.77	0.00	7 / 0	–	–	–	7
HM1	1.00	1.00	0.00, 0.42	0.80	1.00	1.00	7 / 6	18 / 4	8 / 0	–	23
HM2	1.00	1.00	0.80, 0.20	1.00	1.00	1.00	3 / 1	2 / 0	–	–	4
RM1	0.00	1.00	1.00, 0.00	1.00	1.00	1.00	2 / 0	–	–	–	2
RM2	1.00	1.00	0.00, 0.00	1.00	0.82	1.00	7 / 6	18 / 4	8 / 0	–	23
RM3	1.00	1.00	0.00, 0.00	1.00	0.82	0.00	10 / 7	30 / 7	14 / 0	–	38
RS	0.66	1.00	0.11, 0.11	0.63	1.00	1.00	9 / 14	37 / 14	61 / 5	10 / 0	89
TH1	1.00	1.00	0.00, 0.00	0.67	1.00	1.00	7 / 6	18 / 4	8 / 0	–	23
TH2	1.00	1.00	0.66, 0.00	1.00	1.00	1.00	3 / 1	2 / 0	–	–	4

Table 10.3: This table shows a) the architecture assessment (per decision/violation pair) of the original models used in our study, b) the number of models generated at each step of an iterative application of our algorithms, and c) the number of violation instances (generated models × violations per model) still remaining, or introduced, after each iteration, plus d) the resulting number of suggested (optimal) models at the end (cf. Figure 10.4 for a detailed example).

11 Improving Microservice Architecture Conformance to Design Decisions

This chapter employs the architectural design decisions featured in Chapter 7 and the method outlined in the preceding chapter to facilitate the semi-automated detection and rectification of conformance violations. The objective is to aid the software architect by presenting a range of viable options for resolution and creating models of "fixed" architectures.

11.1 Introduction

Microservices are one of many service-based architecture decomposition approaches (see e.g. [PJ16, PW09, Ric17, ZGK+07]). The chief features of microservices are that they communicate via message-based remote APIs in a loosely coupled fashion, and that they can be highly polyglot; ideally, microservices should not share their data with other services. This allows the rapid evolution of individual microservices independently of one another, and their independent deployment in lightweight containers or other virtualized environments. These features make microservices ideal for DevOps practices (see e.g. [Zim17, PZA+17]).

While a large body of literature has examined architectural patterns and recommended "best practices" in a microservice context [Ric17, ZSZ+19, Sko19], translating these theoretical insights into usable tools to assist the architectural evolution of actual microservice-based systems has lagged behind. While the theoretical tenets proposed in the literature are easy to grasp and maintain in small-scale systems, ensuring conformance in large, complex, as well as rapidly and independently evolving systems quickly becomes a laborious affair requiring considerable manual work and resulting in extensive overhead effort. Furthermore, patterns have mutual dependencies, meaning that improvement in one area can result in deterioration in another. Real-world architectures are also impacted by a number of non-microservice-specific requirements, which also can lead to unintended violations of microservice best practices.

This work provides a set of actionable solutions to violations on different aspects of microservice architectures, as part of a larger study on the topic. Three architectural design decisions (ADDs) were selected as representing very different aspects of architecting microservices, so as to demonstrate the wide applicability of our approach. Other ADDs have already been covered in our prior work. More specifically, for covering the best practices of client-system communication we chose the External API decision; for the guaranteed delivery of messages, a critical aspect of many business-critical microservice systems, we used the Inter-Service Message Persistence decision to examine the relevant recommended practices; finally, to cover the logging and monitoring practices that ensure observability of the microservices and their complex interactions, we used the End-to-End Tracing decision. In this context, we aim to study the following research

questions:

- **RQ4.3** What are the possible architecture violations related to the above-mentioned ADDs and how can they be automatically detected?
- **RQ4.4** What are the possible fixes for the violations found in RQ4.3 and how can architects be assisted in choosing the appropriate solutions and applying them?

We propose a novel architecture refactoring approach that uses empirically validated metrics proposed in our prior work [NZP+20b] to evaluate the degree of architecture conformance for each of the given ADDs. For every ADD design option, we define every possible violation and propose a corresponding, automated violation detection algorithm, as well as a set of possible fixes. For each microservice-based system, the sets of ADD options, violations, and fixes leads to a search tree of possible architecture designs that partly or entirely enforce conformance to best practices, which we can continually assess using our metrics.

To evaluate our approach we utilized a set of 24 models of microservice-based systems from third-party practitioners (see Table 3.1). For each of these, we implemented the automated violation detection and refactoring (fix) algorithms to detect the possible violations and to generate all the possible fixes for addressing each violation, resulting in a set of models. Using our metrics, we evaluated the improvements compared with the original version, as well as any outstanding issues. This process was iteratively repeated until all violations were resolved. Each of the violations found in the 24 models can be fully resolved leading to optimal metric values within *at most* 3 refactoring steps, usually with many suggested optimal models provided as options for architects to choose from.

This chapter is structured as follows: In Section 11.2 we analyze the ADDs examined in this work, the associated patterns and practices, and the corresponding metrics. Section 11.3 discusses and compares our approach to existing studies in the literature. Our research methods and the tools we have applied in our study are described in Section **??**, followed by a detailed explanation of our approach in Section 11.5. The evaluation process is given at Section 11.6, the results are discussed in Section 11.7, and the threats to validity in Section 11.8. Finally, in Section 11.9 we draw conclusions and discuss future work.

11.2 Background: Decisions and Metrics

In this section, we briefly introduce the three ADDs and the corresponding patterns and practices as decision options, based on our prior work. The decisions have been modeled based on an empirical study of existing best practices and patterns by practitioners [NZP+19], while the metrics used to assess the pattern conformance of each given system derive from [NZP+20b].

External API Decision. A fundamental decision in microservice-based systems is how external clients are connected to the system services. This can affect aspects related to loose coupling, releasability, independent development and deployment, and continuous delivery. The simplest method, but with the highest negative impact, occurs when the *clients can call into system services directly*, resulting in high coupling that impedes releasing, developing, and deploying

the clients and system services independently of each other. Another option, that solves possible problems caused by client-service direct connections, is the *API Gateway* [Ric17], which provides a common entry point for the system (Facade component) and all client requests are routed via this component. It is a specialized variant of a *Reverse Proxy*, which covers only the routing aspects of an *API Gateway* but not further API abstractions such as authentication, rate limiting, etc. (see [ZSZ+19]). The *Backends for Frontends* pattern [Ric17] is another variant of *API Gateway* that specializes in handling different types of clients (e.g., mobile and desktop clients). Alternatively, the *API Composition* pattern [Ric17] describes a service that shields other services from the clients by actively gathering and composing their data. In our previous work [NZP+20b], we have empirically defined two metrics that can be used to assess conformance to each of the decision options:

- *Client-side Communication via Facade utilization metric* measures how many unique client links are using the External API used by one of the Facade components (i.e. offered through patterns such as *API Gateway*, *Reverse Proxy*, *Backends for Frontends*) compared to the total number of unique client links.

- *API Composition utilization metric* measures the proportion of clients connected services which are possibly composing an *External API* using *API Composition*.

Inter-Service Message Persistence Decision. The persistence or missing persistence of the inter-service messages is another decision with considerable impact on the qualities of the system. Many real-world systems use *no inter-service message persistence*, while options that support message persistence are the *Messaging* pattern [HW03a], in which persistent message queuing is used to store a producer's messages until the consumer receives them, or alternatively *Stream Processing* [Sko19] components (e.g. Apache Kafka). Another option is *Interaction through a Shared Database*, since it supports some level of message persistence, but not the automated support of *Messaging*. A technique that is more microservice-relevant and able to support a lower level of persistence to *Messaging* or a *Shared Database* is the combination of the *Outbox* and the *Transaction Log Tailing* patterns [Ric17]. A persistence more tailored to event-driven or eventually consistent microservice architectures can be achieved following the *Event Sourcing* pattern [Ric17]. For this decision, too, we have empirically defined three metrics that can be used to assess conformance to each of the decision options:

- *Service Messaging Persistence utilization metric* measures the proportion of all service interconnections that are made persistent through a supporting technology (i.e. *Messaging* or *Stream Processing*).

- *Shared Database utilization metric* measures the proportion of all interconnections via a *Shared Database*.

- *Outbox/Event Sourcing utilization metric* measures the proportion of all interconnections with *Outbox/Event Sourcing*.

End-to-end Tracing Decision. End-to-end tracing is an important aspect in microservice architectures since they are usually highly distributed and polyglot systems with complex interactions. One option, like in the other decisions, is to offer *no tracing support*. Alternatively, traces can be recorded on either the services themselves or facade components (or both) via *Distributed Tracing* [Ric17]. A less comprehensive level of tracing can be achieved when service communication is routed through a central component, which stores some, but not all inter-service communication (e.g., *Publish/Subscribe*, *Message Broker* [HW03a], *API Gateway* or *Event Logging* [Ric17, Sko19]); the exception is *Event Sourcing*, which temporarily stores *all* service events.

For this decision, too, we have empirically defined three metrics that can be used to assess conformance to each of the decision options:

- *Services and Facades Support Distributed Tracing metric* measures the proportion of all services and facades that support distributed tracing.
- *Service Interaction via Central Component utilization w/o Event Sourcing metric* measures the proportion of all service interactions through a central component other than *Event Sourcing*.
- *Service Interaction via Central Component with Event Sourcing metric* measures the proportion of all service interactions through a central component via *Event Sourcing*.

11.3 Related Work

The fundamentals of the term "microservices" were first discussed by Fowler and Lewis [LF04], and fundamental tenets by Zimmermann [Zim17]. Richardson [Ric17] has published a collection of microservice patterns and practices, while a mapping study by Pahl and Jamshidi [PJ16] has summarized much of the previous literature on patterns. Skowronski [Sko19] has examined event-driven microservice architectures specifically, and microservice API patterns were studied by Zimmermann et al. [ZSZ+19].

A number of studies have focus on techniques for detecting design or architecture "bad smells" (violations). Taibi and Lenarduzzi [TL18] defined a list of microservice-specific smells, while Neri et al. [NSZB19] have presented an extensive examination of architectural smells for independent deployability, horizontal scalability, fault isolation, and decentralisation of microservices, as well as suggesting refactorings to resolve them. Most similar studies are more generic, but still useful. Le et al. [LCCM16] proposed a classification of architectural smells and their impact on different quality attributes. Catalogs of smells have been published by Garcia et al. [GPEM09a, GPEM09b] and Azadi et al. [AFT19]. Detection strategies for smell categories related to our study are discussed by Brogi et al. [BNS20], Le et al. [LLSM18], Marinescu [Mar04], and especially Neri et al. [NSZB19], along with suggested refactorings for resolving them. Although these works study various aspects of architecture violations detection, and some investigate aspects related to the microservice domain, none covers detecting and addressing violations specifically associated with the ADDs covered in this work (external

API, persistent messaging, and end-to-end tracing) in a microservice context, which our work investigates in detail.

As a result, we expect that our work produces more accurate detection of decision-specific violations and more targeted suggestions for fixes. On the other hand, our approach requires a model in which the component and connector roles in a microservice architecture have been modeled (as for instance done with stereotypes in the model introduced in Figure 11.2). That is, our work requires additional insight into a system's architecture, and some effort in encoding the corresponding models; however, this knowledge is at a relatively high level of abstraction and the resulting models are not impacted by changes in service implementation. We are currently working on a semi-automatic approach for architecture reconstruction and modelling that relies on reusable code abstractions and is thus suitable for complex systems with short delivery cycles.

11.4 Research and Modeling Methods

In this section, we summarize the main research methods applied in our study. These have been more extensively described in our previous work [NZPG21]. For reproducibility, all the code of the algorithms' implementation and the models produced in this study will be made available online, as an open-access dataset in a long-term archive[1].

11.4.1 Research Method

Figure 12.3 shows the structure of the research process of this study. In Section 11.2 we have already explained in detail the architectural decisions and the model-based metrics on which this study is based. In Section 11.5 we present precise definitions and algorithms a) for the detection of possible violations per decision option, and b) for the possible fixes (architecture refactorings) for each violation.

We have tested our approach by applying the algorithms to the 24 models in our data set. First all violations present in each model were detected, and then all possible fixes for each violation were applied in an iterative-exhaustive manner, i.e., on the resulting, refactored models for each violation fix, we again performed *all* violation detection algorithms and applied *all* possible refactorings, until either no more violations were detected, or we arrived at a refactored model identical to a previous version. In the latter case, which we did not encounter here, this would have meant that a violation could not be entirely resolved, as its fix introduced other violations. For each of the final models (the 'leaves' of the iteration tree), we assessed pattern conformance through our metrics on microservice coupling, to judge the improvement compared to the original model.

11.5 Architecture Refactoring Approach

From an abstract point of view, a microservice-based system is composed of components and connectors, with distinct sets of component types and connector types. This applies also to

[1] https://doi.org/10.5281/zenodo.5549978

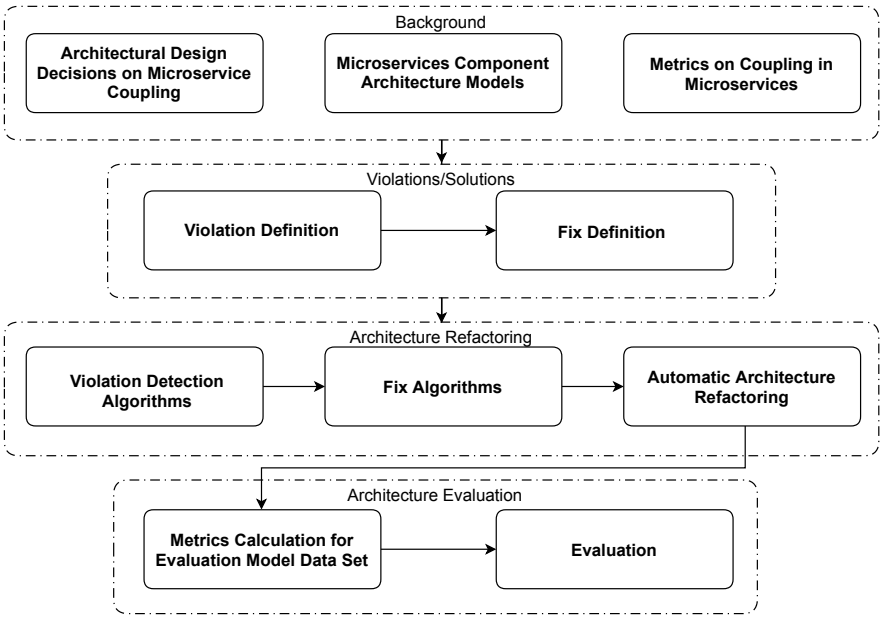

Figure 11.1: Overview diagram of the research method followed in this study

indirect or implicit relationships between components, such as indirect dependencies, which can be described as a special set of connectors. For example, in Figure 11.2, two components are indirectly linked via the API gateway.

We base our definitions of the violations and fixes on the notion of an architecture model consisting of a directed components and connectors graph. This can be expressed formally as: A microservice architecture model M is a tuple (CP, CN, CPT, CNT, ST) where:

- CP is a finite set of **component nodes**. The operation $components(M)$ returns all components in M.
- $CN \subseteq CP \times CP$ is an ordered finite set of **connectors**. $connectors(M)$ returns all connectors in M.
- CPT is a set of **component types**. The operation $services(M)$ returns all components of type $service$ in M. The operation $service\_connectors(M)$ returns all connectors of components of type $service$ in M.
- CNT is a set of **connector types**.
- ST is a finite set of **stereotype nodes**. The operation $cp\_stereotypes(CP)$ returns all stereotypes of component CP. The operation $cn\_stereotypes(CN)$ returns all stereotypes of connector CN. Stereotypes can be applied to components to denote their type, such as *Service*, *API Gateway*, etc. Stereotypes can be applied to connectors to denote their type, such as *Read_Data*, *RESTful HTTP*, or *Asynchronous*. Some are specialized with tagged values (details omitted here for space reasons).

- $cp\_annotations : CP \rightarrow \{String\}$ is a function that maps an component to its set of annotations. Annotations are used in our approach (in some of the fixes) to document aspects that need further consideration or maybe manual refactoring.
- $cn\_annotations : CN \rightarrow \{String\}$ is a function that maps a connector to its set of annotations.

Please note that we define many additional model traversal operations not detailed here for space reasons.

11.5.1 Violations and Detection Algorithms

Violation	***Violation Detection Algorithm Summary***
D1: External API	
D1.V1: Services are directly connected to clients	All services in the model are traversed, and it is checked whether services are directly connected to clients or web UIs. If this is the case, a violation is raised. Each service-client connector that is found is returned by the detector operation.
D2: Persistent Messaging for Inter-Service Communication	
D2.V1: Services communicate without using an intermediary component that is able to persist the communication (e.g., Message Brokers or a persistent Publish/Subscribe or Stream Processing or Event Sourcing or Outbox/Transaction Log Tailing or Database) and no persistent messaging occurs between them.	All service connectors in the model are traversed. If no intermediary component is found, the violation is raised and the list of all relevant connectors is returned by the detector operation.
D3: End-to-End Tracing	
D3.V1: Distributed Tracing is not supported on services and/or facades or services communicate without using a central intermediary component (e.g., Message Brokers or persistent Publish/Subscribe or Stream Processing or Event Sourcing or Outbox/Transaction Log Tailing or API Gateway)	All services, facades and the corresponding connectors in the model are traversed, and it is checked whether services and/or facades support tracing or whether an intermediary component is presented. If no intermediary component or tracing support on services/facades is found, the violation is raised and the list of all relevant connectors is returned by the detector operation.

Table 11.1: Identified Violations and Violation Detection Algorithms

Table 11.1 summarizes the possible violations we have identified for each of the decisions. The table also describes in detail how the algorithms that we use for detecting the violations in the models work. As a detailed example, Algorithm 1 detects the *Services communicate without using an intermediary component* violation of Decision D2. It returns a list of connected service pairs s_i and s_j, that are *not* connected via an intermediary component.

Algorithm 7: Services Communicate w/o Intermediary Component Violation

```
input: Model M
output: Set<Tuple> Component intermediary
begin
  violations ← ∅
  for s_i ∈ services(M):
```

```
  for s_j ∈ services(M):
    if (s_i, s_j) ∈ direct_service_connectors(M):
      violations ← violations ∪ (s_i, s_j)
  return violations
end
```

11.5.2 Fix Options and Algorithms

Table 11.2 details all the fixes for each identified violation, along with a summary of the fix algorithm. Please note that many algorithms can only be applied fully automatically with their default values. Many of them require human review and decision by the architect. For example, the architects can be presented with a choice of an intermediary component to use to replace services links.

Violation	*Fix*	*Fix and Fix Algorithm Summary*
D1: External API		
D1.V1	*D1.V1.F1: Do not fix the violation*	The architect should have the option to *not* fix the violation, e.g. because it is not critical.
	D1.V1.F2: Introduce a new API Gateway and connect client to services via it	Disconnect client(s) from the services and introduce a new API Gateway. Connect the client(s) to the API Gateway and the API to each former client-connected service.
	D1.V1.F3: Introduce API Composition service or service with reverse proxy capabilities and connect client(s) to the services via this component	Disconnect client(s) from the services and introduce a new API composition service. Connect the client(s) to the API composition service and the latter to each former client-connected service.
D2: Persistent Messaging for Inter-Service Communication		
D2.V1	*D2.V1.F1: Do not fix the violation*	The architect should have the option to *not* fix the violation, e.g. because it is not critical.
	D2.V1.F2: Remove the non-persistent connectors between services and replace them with persistent messaging-based connectors	Replace non-persistent interconnections with interactions via an intermediary component (e.g., API Gateway, Pub/Sub, Message Broker). The architect has to select if an existing intermediary component can be used for the fix, or a new one has to be created. Replace non-persistent interconnections with persistent interconnections via this component.
	D2.V1.F3: Remove the non-persistent connectors between services and replace them by writing to and reading from a common database	The architect has to select if an existing database can be used for the fix, or a new one has to be created. For each connector, introduce communication by writing to and reading from this database. Delete the non-persistent interconnections.
D3: End-to-End Tracing		
D3.V1	*D3.V1.F1: Do not fix the violation*	The architect should have the option to *not* fix the violation, e.g. because it is not critical.
	D3.V1.F2: Remove the connectors that don't support end-to-end tracing between services and replace them with interactions via an intermediary component (e.g., API Gateway, Pub/Sub, Message Broker)	The architect has to select if an existing intermediary component can be used for the fix, or a new one has to be created. Replace interconnections that don't support end-to-end tracing with interconnections via this component.
	D3.V1.F3: Connect services and facades that don't support end-to-end tracing with a tracing component (e.g., Zipkin)	The architect has to select if an existing tracing component can be used for the fix, or a new one has to be created. Introduce interconnections from service and facades to tracing component.

Table 11.2: Identified Fixes And Fix Algorithms

The Algorithms 2 and 3 respectively present the fixes F2 and F3, for Decision D2 and its Violation V1. For explanations of each fix, please study Table 11.2.

Algorithm 8: Remove the non-persistent connectors between services and replace them with persistent messaging-based connectors

```
input: Model M, Set<Tuple> violation, Component intermediary_component
output: –
begin
  for (s_i, s_j) ∈ violation:
    add_connector(s_i, intermediary_component,
      get_applicable_stereotypes(M, (s_i, s_j)))
    add_connector(intermediary, s_m,
      get_applicable_stereotypes(M, (s_i, s_j)))
  delete_direct_connector(M, (s_i, s_j))
end
```

Algorithm 9: Remove the non-persistent connectors between services and replace them by writing to and reading from a common database

```
input: Model M, Set<Tuple> violation, Component database
output: –
begin
  for (s_i, s_j) ∈ violation:
    add_connector(s_i, database,
      get_applicable_stereotypes(M, (s_i, s_j)))
    add_connector(s_j, database,
      get_applicable_stereotypes(M, (s_i, s_j)))
  delete_direct_connector(M, (s_i, s_j))
end
```

11.5.3 Example Application

In Figure 11.2 the model CI4 from Table 3.1 is shown as an illustrative example to demonstrate all three violations and possible fixes. In this model the *Cinema Catalog* service is connected directly with *Movie* and *Booking* services, causing D2.V1 and D3.V1, while *Client* is connected directly with *Cinema Catalog* service, causing D1.V1. In contrast, *Booking Payment* and *Notification* services are connected to each other and with the *Client* through the *API Gateway*, resulting in no violation. If we run our fix algorithms, some of the resulting refactoring suggestions are:

- *Applying Fix D1.V1.F2:* The architect can choose the existing *API Gateway* and connect *Client* to *Cinema Catalog* and *Movie* services through it. The current connectors are removed by this fix.

- *Applying Fix D1.V1.F3:* The architect can introduce an *API composition* service or service with reverse proxy capabilities and connect *Client* to *Cinema Catalog* and *Movie* services through it. The current connectors are removed by this fix.

- *Applying Fix D2.V1.F2:* All services with non-persistent connectors are disconnected and connected to a *Message-based persistent mechanism* (all interactions will be happening via

this component). For example, this fix can introduce a new *Pub/Sub intermediary component* (alternatively *Message Broker* or *API Gateway*), to which all involved services will be connected with *publish* and *subscribe* operations supporting persistent communications.

- *Applying Fix D2.V1.F3:* All services with non-persistent connectors are disconnected from each other as well as from their existing databases and connected to a new *shared database* with *read* and *write* operations.
- *Applying Fix D3.V1.F2: Cinema Catalog*, *Movie* and *Booking* services that don't support end-to-end tracing will be disconnected from each other and connected to a new (or existing) *intermediary component* (e.g., *Pub/Sub*, *Message Broker* or *API Gateway*).
- *Applying Fix D3.V1.F3:* A new *tracing component* (e.g., *Zipkin*) is introduced and connected to all services and the *API Gateway*.

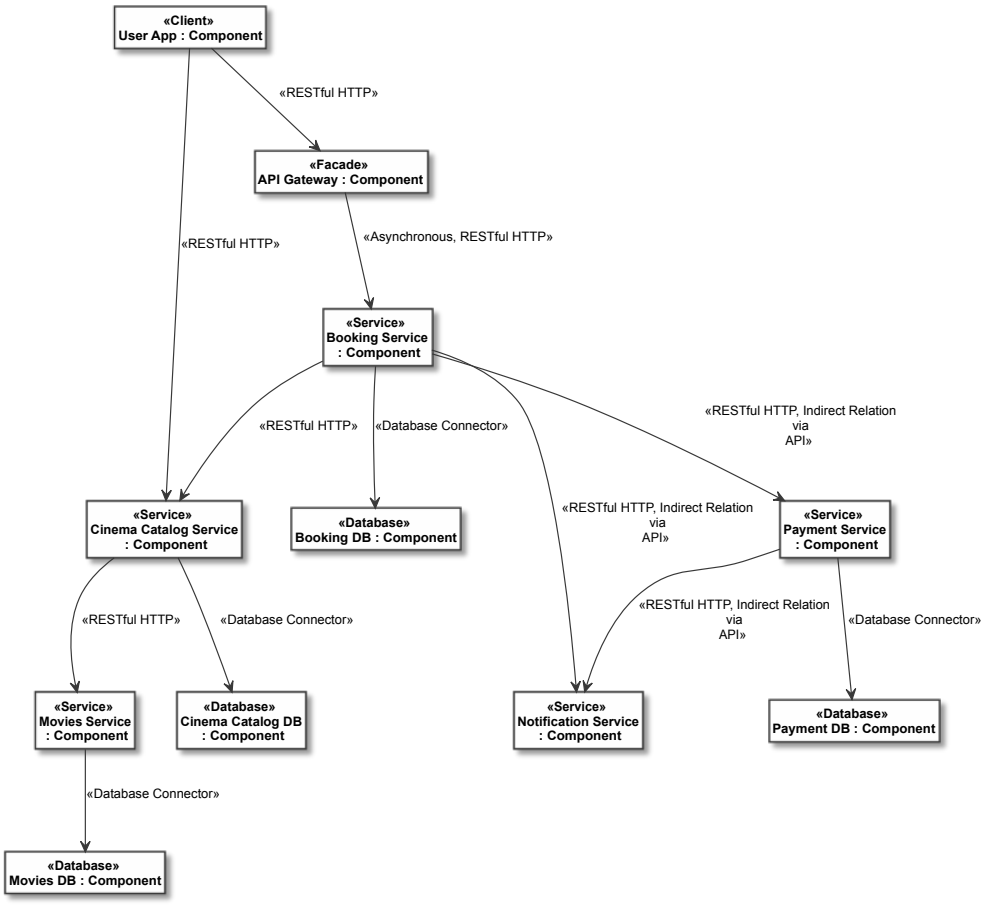

Figure 11.2: Example of an Architecture Component Model (CI4 in Table 3.1): this architecture violates all three ADDs

11.6 Iterative Application and Evaluation

To evaluate our work, we have fully implemented our algorithms for detecting violations and performing fixes, as well as generating the set of metrics described in Section 11.2 to measure the improvements and the presence of remaining violations, in our model set. In case multiple violations are present in a model, then the algorithms can be employed iteratively, until all violations have been fully resolved.

As an example, let us illustrate this exhaustive iterative refactoring for the previously mentioned CI4 Model (see Figure 11.2). CI4 violates all the three decisions as indicated by the corresponding decision-related measures in Table 11.3. The incremental refactoring process is illustrated in Figure 11.3. At the first iteration, there are three branches, indicating the respective violations. The first refactoring step produces 6 possible model variants, one for each fix option from Table 11.2. All resulting models have resolved the respective violation, but have the other two unresolved, requiring another refactoring step that produces 18 new model variants. In turn, 7 of the resulting models still violate D1.V1 and D2.V1, requiring a third step to be resolved. At the end of the third step, we have 29 suggested model variants (M1_1, M2_1, M2_3, M1_2_1–M1_2_2, M2_1_1–M2_2_2, M2_4_1–M2_4_2, M3_1, M3_2_1–M3_2_2, M4_1, M4_2_1–M4_2_2, M4_3_1–M4_3
_2, M5_1–M5_2, M4_4_1–M4_4_2, M6_1_1–M6_2_2, M6_2_1–M6_2_2, M6_3_1–M6_3_2, M6_4_1–M6_4_2) which all fully resolve the violations (i.e., scoring 1.00 in our assessment scale). The architect can choose the refactoring sequence, and from among those final optimal model variants, but can also choose to not apply certain fixes, e.g. due to other constraints that are outside of the scope of our study.

For evaluation purposes, we have performed this procedure for *all* 24 system models in Table 3.1. The resulting number of intermediary models and violation instances per step, and the number of final suggested models with an optimal assessment of 1.00, are given in Table 11.3, along with the initial violations and architecture assessment values for each model. Please note that the metrics reported here are the ones associated with each of the decisions in Section 11.2. Please also note that for each violation to be fixed, it is enough that at least one of the corresponding metrics is optimal (1.00). Obviously, the number of steps required to reach optimal models depends on a) the number of the violations present in the initial model and b) on the possible appearance of new violations during the refactoring process, which did not occur in the present case. As can be seen in Table 11.3, *all* models are *fully resolved*—i.e., all assessment metrics are 1.00—after *at most* three steps.

11.7 Discussion

To answer **RQ4.3** we have systematically specified a number of decision-based violations related to each possible decision option, summarized in Table 11.1. As we have empirically shown in our prior work [NZP+20b] that the metrics described in Section 11.2 can reliably distinguish favored or less favored design options, the role of the violation detectors is to find the precise locations in the models where the violations occur. For each system model in our evaluation dataset it was possible to suggest fixes that bring the architecture to optimal values, meaning that the algorithms

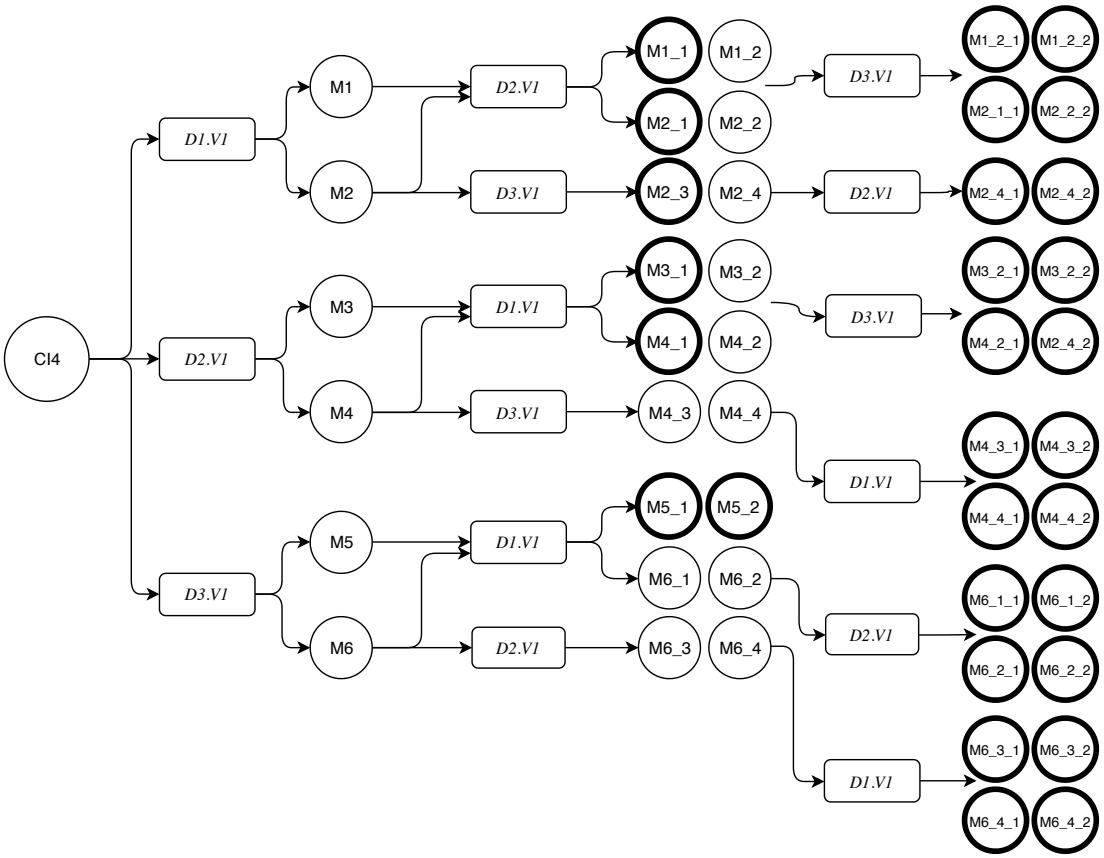

Figure 11.3: Example of an exhaustive iterative application of our approach in the CI4 model. Final (i.e., fully resolved) resulting models are thickly outlined.

have found the right place(s) to apply the fixes.

Regarding **RQ4.4** we defined a number of algorithms addressing every possible violation, with multiple fix options (cf. Table 11.2). If all options are tried out, this results in a search tree of possible architecture models, which can in turn be assessed, using our metrics, to measure improvements to the initial architecture and detect any remaining violations. We have shown (cf. Table 11.3) that an iterative approach results, within a few steps, in a sufficient variety of possible architecture models that remove all detected violations and ensure pattern conformance of the system architecture. The multiple optimal model variants that result from our approach give architects substantial levels of freedom in their design decisions. As detection is fully automated and human expertise is limited to the fix process, the approach is well suited to be run in a continuous delivery environment, which was one of our research goals.

Model ID	Initial Model Assessments			Models Generated / Remaining Violation Instances per Refactoring Step			Resulting Suggested (Optimal) Models
	D1.V1	*D2.V1*	*D3.V1*	*Step 1*	*Step 2*	*Step 3*	
BM1	1.00, 0.00	0.00, 0.00, 1.00	0.00, 0.00, 1.00	–	–	–	–
BM2	1.00, 0.00	0.00, 0.00, 0.00	0.00, 1.00, 0.00	2 / 0	–	–	2
BM3	1.00, 0.00	0.00, 0.00, 0.00	0.00, 1.00, 0.00	2 / 0	–	–	2
CO1	0.00, 0.00	0.00, 0.00, 0.00	0.00, 0.00, 0.00	6 / 9	18 / 11	22 / 0	29
CO2	1.00, 0.00	1.00, 0.00, 0.00	1.00, 1.00, 0.00	–	–	–	–
CO3	0.00, 0.00	0.00, 1.00, 0.00	1.00, 0.00, 0.00	2 / 0	–	–	2
CI1	1.00, 0.00	0.00, 0.00, 0.00	0.00, 0.14, 0.00	4 / 2	4 / 0	–	6
CI2	1.00, 0.00	0.00, 0.00, 0.00	0.00, 1.00, 0.00	2 / 0	–	–	2
CI3	0.00, 0.30	0.00, 0.00, 0.00	0.00, 0.00, 0.00	6 / 9	18 / 11	22 / 0	29
CI4	0.50, 0.10	0.00, 0.00, 0.00	0.00, 0.60, 0.00	6 / 9	18 / 11	22 / 0	29
EC1	0.25, 0.00	0.00, 0.00, 0.00	0.00, 1.00, 0.00	4 / 4	8 / 0	–	8
EC2	0.25, 0.00	1.00, 0.00, 1.00	0.00, 0.00, 1.00	2 / 0	–	–	2
EC3	0.25, 0.00	0.00, 1.00, 0.00	0.00, 0.00, 0.00	4 / 2	4 / 0	–	4
ES1	1.00, 0.00	0.60, 0.00, 0.60	0.00, 0.60, 0.00	4 / 2	4 / 0	–	6
ES2	1.00, 0.00	0.00, 0.00, 0.00	0.00, 0.45, 0.00	4 / 2	4 / 0	–	6
ES3	1.00, 0.00	0.00, 0.00, 0.00	0.00, 0.45, 0.00	4 / 2	4 / 0	–	6
FM1	0.00, 0.25	0.00, 0.00, 0.00	0.00, 0.00, 0.00	6 / 9	18 / 11	22 / 0	29
FM2	0.00, 0.50	0.00, 0.00, 0.00	1.00, 0.00, 0.00	4 / 4	8 / 0	–	8
HM1	0.00, 0.70	0.00, 0.00, 0.00	0.90, 0.00, 0.00	6 / 9	18 / 11	22 / 0	29
HM2	0.00, 0.70	0.80, 0.00, 0.80	0.90, 0.00, 0.80	6 / 9	18 / 11	22 / 0	29
RM	1.00, 0.00	1.00, 0.00, 0.00	0.14, 1.00, 0.00	–	–	–	–
RS	1.00, 0.00	0.11, 0.00, 0.00	0.62, 0.11, 0.00	4 / 2	4 / 0	–	6
TH1	0.25, 0.12	0.00, 0.00, 0.00	0.00, 0.00, 0.00	6 / 9	18 / 11	22 / 0	29
TH2	0.25, 0.04	0.66, 0.00, 0.66	0.00, 0.00, 0.66	6 / 9	18 / 11	22 / 0	29

Table 11.3: This table shows a) the architecture assessment (per decision/violation pair) of the original models used in our study, b) the number of models generated at each step of an iterative application of our algorithms, and c) the number of violation instances (generated models × violations per model) still remaining, or introduced, after each iteration, plus d) the resulting number of suggested (optimal) models at the end (cf. Figure 11.3 for a detailed example).

11.8 Threats to Validity

The basis material of our study derives from third-party sources: the solutions we propose are gathered from the best practices recommended in the published literature, and our evaluation dataset is a fairly representative set of systems (cf. Table 3.1), derived from nine different sources and published with the express purpose of demonstrating microservice architecture features. One possible threat to the internal validity of our algorithms is that they depend on the particular modelling approach we have adopted. However, our approach is by design abstract and generic, based on typical component-and-connector models used widely in the literature. The author team, with considerable experience in modeling methods, performed the system modeling as well as,

repeatedly and independently cross-checked all models. As the main modelling criterion was the ability to adequately represent the context of our systems, we cannot exclude that other teams might arrive at different interpretations, but we are confident that any resulting models would be broadly similar and compatible with our results. Furthermore, the algorithms we specified could easily be adapted to a different model, as they operate on the level of basic architectural constructs.

Nevertheless, some limitations remain. In order to remove the obstacles provided by the polyglot nature of microservice-based systems, we have chosen to apply our metrics and tools at a relatively high level of abstraction. We also limited our evaluation in the present paper to the patterns, metrics, and concerns applying to the given three ADDs, which in a real-world architecture would be insufficient. This point is addressed in previously published and ongoing parts of our work, which extend the coverage to additional ADDs, and aim to extend and test our approach in a larger set of patterns, design requirements, and more granular parameters. The same concern applies as to the lack of evaluation of the applicability of our approach on larger and more complex systems that are commonly found in industry, but which were not accessible to us for study. The lack of full automation is also a major obstacle to practical application, as the process still requires considerable input by the architect. At the same time, our approach can not match the ability of an experienced architect, familiar with the system, to devise a much more optimal solution. This is a limitation of all generic architecture assistance approaches, and one we intend to improve on. We want to emphasize that the present approach is a starting point from which the question of evaluating and improving microservice architectures can be examined, facilitating and building up to more complex and nuanced methods as more systems and decisions are modelled and tested. The generated models are also not optimal, as they are not evaluated, for example, on the coding/refactoring effort required to implement them. Nevertheless, the existence of a semi-automatic approach that detects and analyzes violations in an architecture remains of great value, since practitioners often ignore best practices, systems are often developed without a conscious effort to follow best practices, or are allowed to drift from the original architecture specifications over time.

11.9 Conclusion and Future Work

In this chapter we present a set of violations for three microservice-related ADDs. Building on previous work, we have defined automatic detectors, which return the location where the violations occur, a set of possible fixes for each violation, and automatic algorithms for refactoring the system in order to fix the violations. We have evaluated our approach on a set of 24 models of various degrees of pattern violations and architecture complexity, and have shown that our approach is capable of resolving these violations in at most 3 refactoring steps. Both metric calculation and violation detection are fully automated, but the choice of fixes and refactoring sequence remains with the human architect. Thus the approach is still flexible enough to let the architect make meaningful architectural design choices.

In our future work, we aim to broaden the set of ADDs and violations included in our approach, enrich it with runtime metrics and other architecture aspects such as deployment environments, and extend our model dataset to include larger and more complex systems. In addition, we hope

to experimentally validate our approach by employing it in real-world delivery pipelines as part of a feedback loop.

12 Detecting and Resolving IaC-Based Architecture Smells in Microservices

In this chapter, we utilize the architectural design decisions discussed in Chapter 8 and the methodology explained in Chapter 10 to enable the semi-automated detection and correction of conformance smells. Our objective is to assist software architects by offering them various practical remedies to correct these smells and generate "fixed" architecture models.

12.1 Introduction

Microservice-based systems often support rapid release techniques, resulting in frequent infrastructure and deployment modifications. Additionally, the infrastructure components that a system needs are growing rapidly [Nyg07]. Managing and organizing these pieces often impacts the development and deployment processes. Infrastructure as Code (IaC) facilitates automated management and provisioning of infrastructure components [Mor15]. By dividing deployment artifacts according to the duties of services and teams, IaC can also ensure that a deployed environment stays the same each time it is deployed in the same configuration [Mor15, ABDN+17]. Furthermore, it can help with coherence and maintain loose coupling by separating deployment artifacts according to the responsibilities of services and teams. It helps keep the architecture diagrams and the actual deployment consistent.

There have been several architectural patterns and other “best practices” for microservice-based systems [Ric17, ZSZ+19, Sko19] as well as microservice deployments [Mor15]. However, providing practical mechanisms to enforce such patterns and practices specifically for IaC-based deployments has received very little attention up to this point. This is troublesome because managing architecture compliance manually can be challenging, particularly in big complex architectures. Moreover, enhancing one best practice may cause problems with another since best practices are often interdependent. Thus, numerous additional system architectural and implementation constraints impact the architectures in ways that might result in unintentional or deliberate breaches of best practices for IaC-based deployments. In the context of DevOps and continuous delivery, it is anticipated that the architecture changes rapidly and often without central coordination. *Architecture smell* is a term used in software engineering to describe specific characteristics or traits of a software architecture that indicate a potential problem or suboptimal design [GPEM09b]. These smells can arise due to various factors, such as poor modularization, lack of cohesion, high coupling, or inefficient use of design patterns.

We have observed that there are multiple factors contributing to the increasing complexity of architecture. If infrastructure as code technologies are utilized to deploy these architectures, there is a significant possibility that architectural issues may be embedded in the IaC models without

the developers' immediate knowledge.

This study aims to provide actionable solutions to fix architectural smells of loose coupling-related IaC best practices. We focus on two major Architectural Design Decisions (ADD) in this scope, *System Coupling through Deployment Strategy* and *System Coupling through Infrastructure Stack Grouping*, that have been modeled based on an empirical study of existing best practices and patterns used by practitioners in our previous work [NZSB22].

We provide automated architecture refactoring tailored for architectural design in the context of IaC-related ADDs. We also employ the experimentally proven metrics suggested in our previous work [NZSB22]. These metrics allow us to analyze the degree to which an IaC deployment model adheres to preferred or less preferred design alternatives for each of the ADDs previously mentioned. We systematically specify each potential smell for every design option in the ADDs, and propose automated smell detection algorithms based on those specifications. Using the combination of available ADD options, the chosen option, potential smells, and detected smells, we can determine all possible next decision options by applying solutions to the smells. This results in a search tree of models for the next architecture iteration, which we individually evaluate using our metrics. Based on this, we can assess the conformance of IaC-based deployment models regarding architectural patterns and potential refactorings and provide an architect with all potential improvements. The purpose of smell detection is to discover the precise locations in the models where the smells occur.

This method is also intended to be continually applicable across each run of a continuous delivery pipeline.

This paper aims to study the following research questions:

- **RQ4.5** What are the potential coupling-related architectural smells in IaC-based deployment models related to System Coupling through Deployment Strategy and System Coupling through Infrastructure Stack Grouping, and how can they be automatically detected?
- **RQ4.6** What are the possible fixes for the architectural smells in IaC-based deployment models related to System Coupling through Deployment Strategy and System Coupling through Infrastructure Stack Grouping, and how can architects be supported in correcting them?

In total, 12 IaC-based deployment models (three case studies and nine variations) based on microservice-based systems that practitioners developed are used to evaluate our approach (see Table 12.1). We implemented automated smell detection and refactoring algorithms to detect potential smells and develop every solution that may be used to solve each smell. The improvements over the initial version are then measured using our metrics [NZSB22] on coupling aspects in IaC-based deployments. The results show that every smell can be addressed in no more than three refactoring steps, producing ideal metric values.

This paper is structured as follows: In Section 12.2, we explain the decisions in the focus of this paper. We also explain related patterns and practices, as well as the corresponding metrics, as the background of our work. Section 12.3 discusses and compares to related work. Next, we describe the research methods and the tools we have applied in our study in Section 12.4. Then, three case studies are explained in Section 12.5. We then describe the approach details

in Section 12.6. In Section 12.7, we explain the evaluation process of our work. Section 12.8 discusses the RQs regarding the evaluation results. In Section 12.9, we then analyze the threats to validity. Finally, in Section 12.10, we conclude and discuss future work.

12.2 Background

This section will briefly discuss two coupling-related ADDs and their associated options. This information is based on our previous research [NZSB22], in which we conducted an empirical study to identify the IaC-related best practices and patterns currently used by practitioners. We also examined the potential decision drivers, or the factors that influence the decision-making process, and developed metrics to evaluate how well a given system model adheres to our decision model's recommended patterns and practices. By analyzing the reported outcomes of these decisions, we can determine which options are more or less popular among microservice practitioners. We employed 9 IaC-based component and deployment architecture models for evaluation, which are used and extended in this work, listed in Table 12.1 and explained in Section 12.4.

12.2.1 Infrastructure Stack

Infrastructure stacks can be used to organize the deployment infrastructure, which refers to the set of hardware, software, and networking resources required to deploy and run applications, services, and systems in a production environment. According to [Mor15], an infrastructure stack is *a group of infrastructure resources defined, provisioned, and updated collectively*. A non-optimal structure can harm the system if coupling-related factors are not considered. For instance, the dependencies of system parts and teams and the independent deployability of system services might be impacted by grouping all declarations of the system's infrastructure resources in only one infrastructure stack.

Figure 12.1 shows the lifecycle of an infrastructure stack. The resources and services that an infrastructure platform offers are the elements of a stack, and they are described by source code. For instance, a stack might consist of computing resources (e.g., a virtual machine), storage resources (e.g., disk volume), and network resources (e.g., a subnet) [Mor15]. A stack management tool reads the source code for the stack and assembles the defined elements in the code to provision an instance of the infrastructure stack using a cloud platform's API [Mor15].

12.2.2 Architectural Design Decisions (ADDs)

ADD 1: System Coupling through Deployment Strategy

Maintaining the services' independence, scalability, and loose coupling is crucial when implementing a microservice-based system. The corresponding development teams should be able to construct and deploy a service swiftly, and services should be segregated. Another aspect to consider is resource use per service since certain services may restrict CPU or memory usage [Ric17]. Extra criteria should be guaranteed for each autonomous service, such as availability or behavior monitoring.

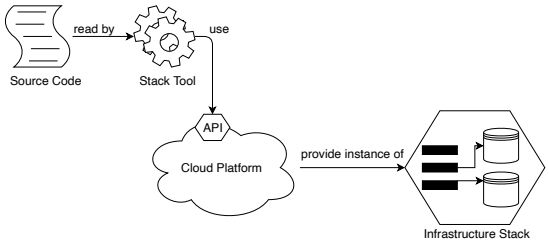

Figure 12.1: The lifecycle of an infrastructure stack. this figure is adopted from Morris book [Mor15]

The System Coupling through Deployment Strategy decision concerns how services are deployed in execution environments. The following decision options can be chosen: (i) *Multiple Services per Execution Environment*, where services are all deployed in the same execution environment making it problematic to change, build, and deploy the services independently. Execution Environment is used here to denote the environment in which a service runs, such as a VM, a Container, or a Host. Please note that execution environments can be nested. For instance, a VM can be part of a Production Environment, which runs on a Public Cloud Environment. Execution environments run on Devices (e.g., Cloud Server). The most recommended option is the (ii) *Single Service per Execution Environment* pattern [Ric17], in which each service is deployed in its execution environment and can be managed independently. In our previous work [NZSB22], we empirically identified two metrics that can be used to differentiate and assess the decision options' conformance:

- *Shared Execution Environment Connectors Metric (SEEC)* to measure the proportion of the shared connectors between services and execution environments.
- *Shared Execution Environment Metric (SEE)* to measure the proportion of the shared execution environments.

ADD 2: System Coupling through Infrastructure Stack Grouping

Another essential aspect of microservices deployment is the grouping of the infrastructure elements. The System Coupling through Infrastructure Stack Grouping decision concerns how grouping different resources into infrastructure stacks should reflect the development teams' responsibilities to ensure independent deployability and scalability. The following decision options can be chosen: *Monolith Stack* [Mor15], where all resources are grouped in a single stack. Another option is *Application Group Stack*, in which multiple services are deployed by one stack. A structuring that can work better with microservice-based systems is the *Service Stack*, in which one stack deploys one service and all related infrastructure resources. The *Micro Stack* pattern [Mor15] goes one step further by breaking the *Service Stack* into even smaller pieces and creating stacks for each infrastructure resource in a service (e.g., router, server, database, etc.). For this decision, we have empirically defined six metrics that can be used to assess conformance to each of the decision options:

- *Monolithic Stack Detection Metric (MSD)* to detect if a single stack is used to deploy all the infrastructure elements.
- *Application Group Stack Detection Metric (AGSD)* to detect if a single stack is used to deploy all system services.
- *Service-Stack Detection Metric (SES)* to detect if every service is deployed by its own stack.
- *Micro-Stack Detection Metric (MST)* to detect if every infrastructure element is deployed by its own stack.
- *Services per Stack Metric (SPS)* to measure how many services are deployed by a service-deploying stack on average.
- *Components per Stack Metric (CPS)* to measure how many components, on average, are deployed by a component-deploying stack.

12.3 Related Work

In this section, we provide details on and compare related works. We first discuss related studies for IaC-based best practices and patterns, then tool-based approaches for smell detection, and finally, approaches for evaluating the conformance of architectures.

12.3.1 Related Works on IaC-Based Best Practices and Patterns

As the industry adopts and popularizes IaC practices, many scientific studies are compiling or organizing IaC-related patterns, practices, smells, and anti-patterns. For example, a list of design and implementation language-specific smells for Puppet is presented by Sharma et al. [SFS16]. Kumara et al. [KGR+21] offer a comprehensive list of best and worst practices relating to implementation problems, design problems and smells of fundamental IaC concepts. Schwarz et al. [SSL18] provide a list of smells for Chef. Morris [Mor15] provides management recommendations for infrastructure as code. This book includes an extensive list of patterns and practices that fall under several categories and a complete discussion of technologies relevant to IaC-based practices. Our work follows the IaC-specific principles outlined in this book and those in [Ric17]. Many of these publications are less concerned with architecture decisions in the deployment architecture than our work is. In contrast to our research, they do not provide architecture conformance assessment or detect and resolve architecture smells.

12.3.2 Tool-based and Network Smell Detection Approaches

A tool-based approach for detecting smells in TOSCA models is proposed by Kumara et al. [KVM+20]. Sotiropoulos et al. [SMS20] develop a tool-based approach that identifies dependency-related issues by analyzing Puppet manifests and their system call trace. Van der Bent et al. [vdBHVG18] define metrics that also reflect best practices to assess Puppet

code quality. Saatkamp et al. [SBKL19] utilize architectural and design patterns to reorganize topology-driven deployment models to identify any issues obstructing a successful deployment. Their approach covers two aspects: (1) identifying problems in reorganized deployment models through architecture and design patterns and (2) automating problem detection by formalizing the issue and its context through implementing patterns. This work presents a method for identifying and implementing suitable solutions for issues in declarative deployment models in an automated manner. Saatkamp et al. [SBF+19] also present an approach that uses first-order logic to evaluate the applicability of solutions to a specific deployment model by expressing the required deployment context as a logical formula. Adaptation algorithms are also defined to operate on topological elements indicated by the deployment context to realize the solution in the deployment model. In [SBKL18] Saatkamp et al. demonstrate using formalized patterns to detect problems in two application scenarios. The Message Mover and Integration Provider patterns, relevant to restructured topology-based deployment models in distributed applications, show the approach's applicability in message-based systems. Reusable conditions to express pattern rules have also been defined. Although some of these works concentrate on the quality assurance of IaC systems, none of them, unlike our work, address and focus primarily on coupling-related issues in IaC deployment models and on architecture smell detection and fixes.

12.3.3 Related works on Frameworks and Metrics

An approach for automatically verifying declarative deployment models' conformity throughout design time is presented in [FBKL17, KBKL18]. The method enables modeling compliance rules as two fragments of a deployment model. One of the parts is a detector subgraph that decides whether the rule applies to a particular deployment model. Subgraph isomorphism compares the model fragments to the deployment model in question. In contrast to our study, this technique generally does not incorporate any particular compliance rules, like checking coupling-related ADDs in IaC models. It presupposes that the rule modeler can convert best practices into compliance rules with the desired format. Additionally, it doesn't indicate the severity of a rule violation; instead, it merely offers a Boolean result showing whether or not the rule is being broken. Weller et al. [WBSB] present the Deployment Model Abstraction Framework (DeMAF), a tool that allows the transformation of technology-specific deployment models into technology-independent deployment models modeled based on the Essential Deployment Metamodel (EDMM). This framework demonstrates the capability of abstracting deployment models in a technology-agnostic manner.

Numerous studies concentrate on methods for spotting design or architectural smells, but most do not specifically target the IaC domain. Garcia et al.'s approach [GPEM09a, GPEM09c] provides a format for a collection of offensive smells in architecture. Additionally, these findings provide potential methods for detecting these architectural smells. The relationship between smells and project problems was investigated by Le et al. [LLSM18]. Marinescu [Mar04] has proposed several detection techniques that use metrics-based heuristics to find design flaws. To detect architecture erosion or drift, Garcia et al. [GPM+11] describe a machine learning-based method for reconstructing an architectural perspective that includes a system's parts and connectors.

Although several of these publications examine features of the microservice domain and other

aspects of architecture smell detection, none address detecting and refactoring coupling-related smells in an IaC domain. This leads to our expectation that, in the context of loose coupling, our work yields more precise detections of decision-specific smells and more focused suggestions for fixes than this other research possibly could.

12.4 Research and Modeling Methods

This section summarizes the main research and modeling methods applied in our study. For reproducibility, all the code and models produced in this study are available online as an open-access data set in a long-term archive [1].

12.4.1 Research Method

The steps used in this study are shown in Figure **??**. We have already provided a detailed explanation of the architectural decisions and model-based metrics that served as the foundation for this study in Section 12.2. We offer explicit definitions and algorithms for detecting potential smells for each decision option and detailed definitions and methods for the possible fixes for each smell in Section 12.6.

Every smell associated with the ADDs listed in the previous section will be found using our method. Any suggested architecture refactorings will be applied to each model in our data set. For each smell fix, we ran all possible smell detection algorithms and refactorings on the resulting refactored models until no more smells were found or the refactored model matched a previous version. In the latter case, this shows that it is impossible to eliminate all smells because doing so would require creating new smells. To assess each final model's improvement over the initial model, we examined pattern conformance using metrics on IaC coupling.

12.4.2 Modeling Method

We used a dataset of 12 IaC-based deployment models (3 case studies and 9 variations) listed in Table 12.1 to evaluate our approach. This dataset consists of three sources of microservice-based systems and deployment artifacts. The fact that professionals with relevant expertise created the systems we discovered supports the notion that they serve as a solid example of the IaC coupling-related best practices enumerated in Section 12.2. Figure 12.2 shows the steps we followed for reconstructing the model from the source code. We conducted a complete manual static code analysis for the IaC models included in the repositories and the source code for the applications. To create our models, we used the Python modeling library CodeableModels described in Chapter 3.7. The result is a collection of meticulously designed software systems and IaC-based deployment models. Figure 12.4 shows an excerpt of an IaC-based deployment model.

[1] https://doi.org/10.5281/zenodo.7692017

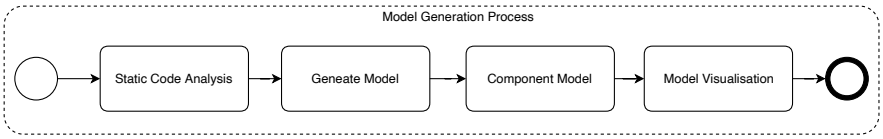

Figure 12.2: Overview diagram of the model generation process

12.5 Case Studies

This section briefly describes the case studies used to evaluate our approach. We studied three open-source microservice-based systems and created nine variants that introduce typical ADD smells of the ADDs described in Section 12.2. Table 12.1 summarizes the case studies and corresponding variants.

Case Study 1: eShopOnContainers Application The *eShopOnContainers* case study is a prototype reference application, realized by Microsoft, built on a microservices architecture and Docker containers that can be used with Azure and Azure cloud services. It features several independent microservices and accommodates various communication modes (e.g., synchronous and asynchronous via a message broker). The code repository also offers the necessary IaC scripts to work with ELK for logging and deployment on a Kubernetes cluster (Elasticsearch, Logstash, Kibana).

Case Study 2: Sock Shop Application The *Sock Shop* is a microservices reference application built by the company Weaveworks to show several microservice architectures and the company's technologies. The application showcases cloud-native and microservices technology. Services are deployed on *Docker* containers, and the system employs *Kubernetes* to manage containers. The system may be deployed on Amazon Web Services (AWS) using infrastructure scripts for Terraform. In our opinion, this is a common practice in the industry regarding microservice-based designs and IaC-based deployments.

Case Study 3: Robot-Shop Application *Robot-Shop* is a reference application by the company Instana that showcases polyglot microservice architectures with Instana monitoring. The IaC scripts are included. Kubernetes is used for container orchestration, and all system services are deployed on Docker containers. *Helm* is also supported for automated cluster *Kubernetes* construction, packaging, setup, and deployment. In addition, some services support *Prometheus* metrics and offer end-to-end monitoring.

12.6 Architecture Smells and Fix Options Definition

This section presents an overview of the various smells, possible solutions, and algorithms we have designed for detecting smells and implementing fixes. To further clarify our approach, we also include examples from the decision of "System Coupling via Infrastructure Stack Grouping."

Case Study ID	*Model Size*	*Description / Source*
CS1	68 components 167 connectors	E-shop application using pub/sub communication for event-based interaction and files for deployment on a Kubernetes cluster. All services are deployed in their infrastructure stack (from `https://github.com/dotnet-architecture/eShopOnContainers`).
CS1.V1	67 components 163 connectors	Variant of Case Study 1 in which half of the services are deployed on the same execution environment, and some infrastructure stacks deploy more than one service.
CS1.V2	60 components 150 connectors	Variant of Case Study 1 in which some services are deployed on the same execution environment and half of the non-services components are deployed by a component-deploying stack.
CS1.V3	60 components 150 connectors	Variant of Case Study 1 in which some services are deployed on the same execution environment and the non-services components are deployed by a component-deploying stack.
CS2	38 components 95 connectors	An online shop that demonstrates and tests microservice and cloud-native technologies and uses a single infrastructure stack to deploy all the elements (from `https://github.com/microservices-demo/microservices-demo`).
CS2.V1	40 components 101 connectors	Variant of Case Study 2 where multiple infrastructure stacks are used to deploy the system elements, as well as some services are deployed on the same execution environment.
CS2.V2	40 components 101 connectors	Variant of Case Study 2 where two infrastructure stacks are used to deploy the system elements (one for the services and one for the rest elements), as well as some services are deployed on the same execution environment.
CS2.V3	60 components 150 connectors	Variant of Case Study 2 in which some services are deployed on the same execution environment and the non-services components are deployed by a component-deploying stack.
CS3	32 components 118 connectors	Robot shop application with various kinds of service interconnections, data stores, and Instana tracing on most services, as well as an infrastructure stack that deploys the services and their related elements (from `https://github.com/instana/robot-shop`).
CS3.V1	56 components 147 connectors	Variant of Case Study 3 where some services are deployed in their infrastructure stack and some services are deployed on the same execution environment.
CS3.V2	56 components 147 connectors	Variant of Case Study 3 where all services are deployed in their infrastructure stack and all services are deployed on their execution environment.
CS3.V3	54 components 148 connectors	Variant of Case Study 3 where some services are deployed in their infrastructure stack and some services are deployed on their execution environment.

Table 12.1: Overview of modeled case studies and the variants (size, details, and sources), adapted from our previous work [NZSB22]

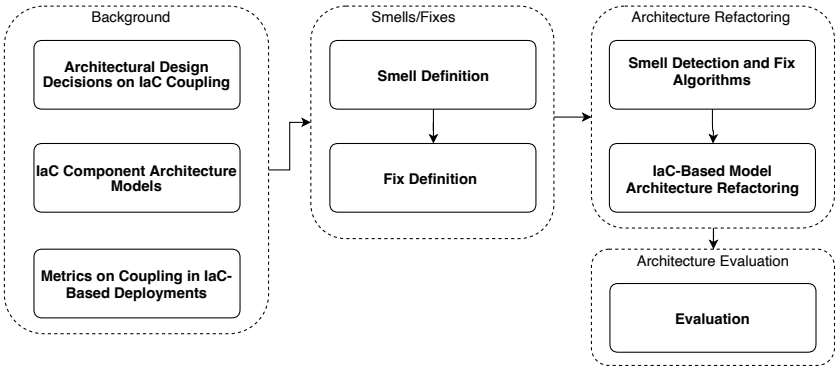

Figure 12.3: Overview diagram of the research method followed in this study (the diagram is adapted from our previous work [NZSB22])

Our previous work [NZSB22] presents a microservice-based architecture model description/-modeling of a directed graph of components and connectors. This model serves as the foundation for our definitions of smells and fixes.

A microservice decomposition and deployment architecture model M is a tuple $(N_M, C_M, NT_M, CT_M, c\_source, c\_target, nm\_connectors, n\_type, c\_type)$ where:

- N_M is a finite set of components and infrastructure **nodes** in Model M.
- $C_M \subseteq N_M \times N_M$ is an ordered finite set of **connector edges**.
- NT_M is a set of **component types**.
- CT_M is a set of **connector types**.
- $c\_source : C_M \to N_M$ is a function returning the component that is the **source** of a link between two nodes.
- $c\_target : C_M \to N_M$ is a function returning the component that is the **target** of a link between two nodes.
- $nm\_connectors : \mathbb{P}(N_M) \to \mathbb{P}(C_M)$ is a function returning the set of connectors for a set of nodes: $nm\_connectors(nm) = \{c \in C_M : (\exists n \in nm : (c\_source(c) = n \wedge c\_target(c) \in C_M) \vee (c\_target(c) = n \wedge c\_source(c) \in C_M))\}$.
- $n\_type : N_M \to \mathbb{P}(NT_M)$ is a function that maps each node to its set of **direct and transitive node types**. (For a formal definition of node types, see [ZNL17].)
- $c\_type : C_M \to \mathbb{P}(CT_M)$ is a function that maps each connector to its set of **direct and transitive connector types**. (For a formal definition of connector types, see [ZNL17].)

All deployment nodes are of type *Deployment_Node*, which has the subtypes *Execution_Environment* and *Device*. These have further subtypes, such as *VM* and *Container* for *Execution_Environment*,

and *Server*, *IoT Device*, *Cloud*, etc. for *Device*. Environments can also distinguish logical environments on the same infrastructure, such as a *Test_Environment* and a *Production_Environment*. Combining all types, e.g., *Production_Environment* and *VM*, is possible.

The microservice decomposition is modeled as nodes of type *Component* with component types such as *Service* and connector types such as *RESTful HTTP*.

The connector type *deployed_on* is used to denote a deployment relation of a *Component* (as a connector source) on an *Execution_Environment* (as a connector target). It is also used to denote the transitive deployment relation of *Execution_Environments* on other ones, e.g., a *Container* that is deployed on a *VM* or a *Test_Environment*. The connector type *runs_on* models the relations between execution environments and the devices they run on.

The type *Stack* is used to define deployments of *Devices* using the *defines_deployment_of* relation. Stacks include environments with their deployed components using the *includes_deployment_node* relation.

12.6.1 Smell Detection

Table 12.2 summarizes the possible smells we have detected for each ADD. It also describes how the algorithms we use to detect smells in models are based on meta-model definition introduced in Section 12.6. For example, Algorithm 12.1 describes the steps required for detecting the Smell *Services are Deployed on a Single Execution Environment* of ADD 1. It returns a list of smells, each represented by a set of service environment connectors in which two services s_m and s_j share an execution environment e_i.

Algorithm 10: Detect System Services are Deployed on a Single Execution Environment Smell

```
input: Model M
output: Set<Tuple>
begin
smells ← ∅
for s_m ∈ services(M):
  for s_j ∈ services(M):
    for e_i ∈ execution_environment(M):
      if ((s_m, e_i) ∈ service_env_connectors(M) ∧
          (s_j, e_i) ∈ service_env_connectors(M)):
        smells ← smells ∪ {(s_m, e_i), (s_j, e_i)}
  return smells
end
```

12.6.2 Fixes

Table 12.3 summarizes all possible fixes for each detected smell and the fix algorithm. Many of the fixes require human review and sometimes a human decision to be applicable. For instance, the architect may be faced with a decision of which infrastructure stack is better suitable to the application requirements. For example, Algorithm 12.2 shows one of the fix algorithms, integrating services deployed in the same execution environments needed to realize D1.S1.F3.

Algorithm 11: Integrate Services Deployed in the Same Execution Environment (D1.S1.F3)

```
input: Model M, Execution_Environment env
```

Smells	*Smell Detection Algorithm Summary*
D1: System Coupling through Deployment Strategy	
D1.S1: System services are running/deployed on a single execution environment [Host/VM/Container].	All service connectors in the model are traversed. If at least two services are deployed on the same execution environment, an instance of the smell is found. The detector operation returns each such service-execution connector that is found.
D2: System Coupling through Infrastructure Stack Grouping	
D2.S1: All infrastructure elements and services are part of a single infrastructure stack.	All infrastructure elements connectors in the model are traversed. If only one infrastructure stack is found to be used by them, an instance of this smell is found. The detector operation returns the list of all relevant model elements.
D2.S2: Two or more services are part of a single infrastructure stack.	All service and stack connectors in the model are traversed. The smell is found if multiple services are clustered in groups on at least one of the stacks. The detector operation returns the list of all relevant model elements.
D2.S3: If Service Stack *is* True*: Infrastructure elements (e.g., databases, routers, etc.) that services depend on are not part of their service stack.*	All infrastructure elements connectors in the model are traversed. Suppose non-service components (e.g., databases) are connected to a different stack than their services. In that case, the smell is found. The detector operation returns the list of all such model elements.
D2.S4: If Service Stack *is* True*: Infrastructure elements (e.g., databases, routers, etc.) that services are not dependent on are part of their service stack.*	All infrastructure elements connectors in the model are traversed. Suppose non-service components (e.g., databases) are connected to a stack they are not dependent on. In that case, the smell is found, and the detector operation returns the list of all relevant model elements.

Table 12.2: Detected Smells and Smell Detection Algorithms

```
output: –
begin
  new_service ← Null
  first ← True
  integration_annotations ← ∅

  for s ∈ get_services(M, env):
    if first:
        new_service = create_service(M,
          get_service_name(M, s),
          get_applicable_stereotypes(M, s))
        first ← False
    else:
        set_service_name(M, new_service,
          get_service_name(M, new_service) + " + " +
          get_service_name(M, s))
        add_applicable_stereotypes(M, new_service, s)

    integration_annotations ← integration_annotations
        ∪ {"integrated functionality from: " +
        get_service_name(M, s)}
    add_connector(new_service, env,
      get_applicable_stereotypes(M, (s, env)))
    delete_service(s)

  add_annotations(M, new_service, integration_annotations)
end
```

Smell	*Fix*	*Fix Summary*
D1: System Coupling through Deployment Strategy		
D1.S1	*D1.S1.F1: Do not fix the smell*	The architect has the option to *not* fix the smell, e.g. because it has no significant impact on system deployment.
	D1.S1.F2: Deploy each service in a separate execution environment	Disconnect services from the execution environment and introduce a new execution environment for each service. Connect the services to the execution environments.
	D1.S1.F3: Integrate the services deployed on the same execution environment into one service	Disconnect services from the execution environment. Merge the services using the same execution environment into a single service using that execution environment. Connect the new service to the execution environment. Here we add annotations that functionality has been added to one service so that implementers, later on, can realize this functionality. The architect must check whether such integration is possible and can provide developers with annotations about the envisaged service integration details.
	D1.S1.F4: Introduce VMs/Containers in a Host/VM to separate services in different execution environments	Disconnect services from the execution environment and introduce new VMs/containers for each service in the same host. Connect the services to the VMs/containers.
D2: System Coupling through Infrastructure Stack Grouping		
D2.S1	*D2.S1.F1: Do not fix the smell*	The architect has the option to *not* fix the smell, e.g. because it has no significant impact on system deployment.
	D2.S1.F2: Create separate infrastructure stacks for each service and additional infrastructure elements	Disconnect services from the infrastructure stack. Introduce new infrastructure stacks for each service and additional infrastructure elements. Connect the services and elements to their stack.
D2.S2	*D2.S2.F1: Do not fix the smell*	The architect has the option to *not* fix the smell, e.g. because it has no significant impact on system deployment.
	D2.S2.F2: Create separate infrastructure stacks for each service	Disconnect services from the infrastructure stack. Introduce new infrastructure stacks for each service and connect the services to their stack.
D2.S3	*D2.S3.F1: Do not fix the smell*	The architect has the option to *not* fix the smell, e.g. because it has no significant impact on system deployment.
	D2.S3.F2: Move the service-dependent infrastructure elements in the same service stack	Disconnect service-dependent infrastructure elements from the infrastructure stack. Connect service-dependent infrastructure elements to the infrastructure stacks they are dependent on.
D2.S4	*D2.S4.F1: Do not fix the smell*	The architect has the option to *not* fix the smell, e.g. because it has no significant impact on system deployment.
	D2.S4.F2: Move the service-independent infrastructure elements in the dependent service stack	Disconnect service-independent infrastructure elements from the infrastructure stack. Connect service-independent infrastructure elements to the infrastructure stacks they are dependent on.

Table 12.3: Detected Fixes And Fix Algorithms

12.6.3 Smell Detection and Fixes Example

Figure 12.4 shows an excerpt model of CS1.V2 from Table 12.1. As an illustrative example, we use it here to demonstrate the *System services are running/deployed on a single execution environment [Host/VM/Container](D1.S1)* smell. All services are deployed on a single *Container* in this model. Here, the *Docker Container 0.0.0.3* is considered as *shared execution environment*, causing the corresponding smell. It would be triggered in our approach by providing a bad metric value, which would trigger the detailed detection, which would return the $\{(\textit{Catalog, Docker Container 0.0.0.3}), (\textit{Basket, Docker Container 0.0.0.3}), (\textit{Order, Docker Container 0.0.0.3})\}$ set of tuples. If we run our fix algorithms, the resulting model fix suggestions are:

- *Applying Fix D1.S1.F2: Catalog*, *Order* and *Basket* services will be disconnected from the

execution environment. The fix will introduce different execution environments for each service, which will be connected to its own execution environment.

- *Applying Fix D1.S1.F3:* The *Catalog*, *Order* and *Basket* services can be integrated into one new service.
- *Applying Fix D1.S1.F4:* The fix will introduce new execution environments for each service as part of the same host. *Catalog*, *Order*, and *Basket* services will be disconnected from the execution environment, and each service will be connected to its execution environment.

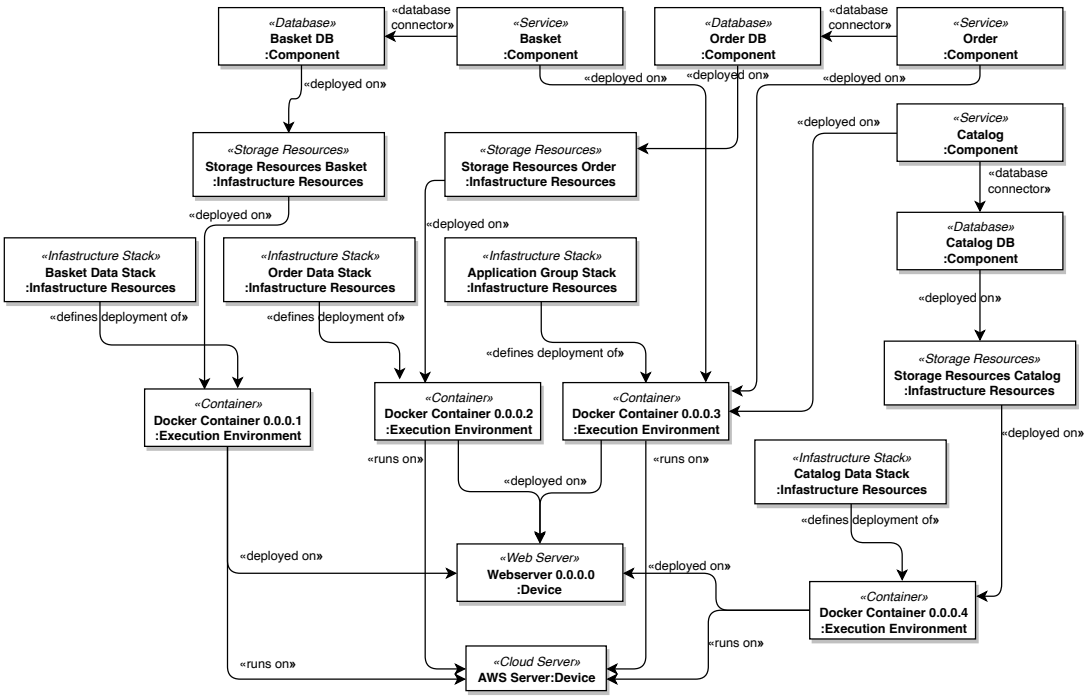

Figure 12.4: Excerpt of an Architecture Component Model of Case CS1.V2 in Table 3.1.

12.7 Evaluation

To evaluate our work, we have fully implemented our algorithms for detecting smells and performing fixes, and generating the metrics described in Section **??** to measure the improvements and presence of remaining smells in our model set. If multiple smells are present in a model, then the algorithms can be employed iteratively until all smells have been fully resolved.

For example, let us illustrate the exhaustive, iterative refactoring for the CS1.V2 Model (see Figure 12.4). CS1.V2 has the following smells—"System services are running/deployed on a single execution environment" (D1.S1), "Two or more services are part of the same infrastructure stack" (D2.S2), and "Infrastructure elements (e.g., databases, routers, etc.) that services depend

on are not part of their service stack" (D2.S3) as indicated by the respective measures in Table 12.2. There are two branches at the first iteration step of the refactoring process in Table 12.4. The first iteration step results in 4 possible model variants, one for each fix option from Table 12.3. In these models, the corresponding smells have been fixed. However, all the new models still contain a smell. M1–M3 still has the D2.S2 smell since this is not resolved in this branch. M4 still has the D1.S3 smell.

The second iteration step results in four further models. In turn, the resulting models M1.A, M2.A, and M3.A, now contain the additional smells D2.S3. At the end of the third step, we have four suggested model variants, all optimally resolving the smells. The architect can choose the refactoring sequence from among these final optimal model variants but can also choose not to apply specific fixes, e.g., due to other constraints outside our study's scope.

We followed this technique for *all* 12 system models in Table 12.1 to evaluate them. In Table 12.5, along with the starting smells and architecture evaluation values for each model, are the number of intermediate models and smell cases at every step, as well as the number of final suggested models with an optimal metric assessment. Please note that the metrics below correspond to each of the smells; D1.S1 has two related metrics, and D2.S2 has three metrics, one for detecting the *Application Group Stack* pattern, one for detecting the *Service Stack* pattern, and one to measure the proportion of this.

The number of smells in the starting model and the potential emergence of additional smells throughout the refactoring process determines the number of steps necessary to attain ideal models. All models are fully resolved, or all assessment metrics have optimal values after a maximum of three phases, as shown in Table 12.5.

	CS1.V2		
Step 1	Smells	D1.S1	D2.S2
	Produced Component Models (Fixes)	M1, M2, M3	M4
Step 2	Smells	D2.S2	D2.S3
	Produced Component Models (Fixes)	M1.A, M2.A, M3.A	**M4.A**
Step 3	Smells	D2.S3	No additional smell
	Produced Component Models (Fixes)	**M1.A-1, M2.A-2, M3.A-3**	**–**
Total	**4 Optimal Component Models**		

Table 12.4: Example of an exhaustive iterative application of our approach in the CS1.V2 model. Final (i.e. optimally resolved) resulting models are rendered in boldface font.

12.8 Discussion of Research Questions

For each potential alternative option, we methodically detected several decision-based smells, which are included in Table 12.2 to address **RQ1**. The purpose of the smell detectors is to discover the precise locations in the models where the smells occur because we have empirically demonstrated in our prior work [NZSB22] that the metrics described in Section 12.2 can reliably distinguish preferred or less preferred design options. For each system model in our evaluation dataset, proposing corrections to improve the architecture was possible. This indicates that the algorithms had correctly detected the resolution's proper location or locations.

Model ID	Initial Model Assessments				Models Generated / Remaining smell Instances per Refactoring Step			Resulting Suggested Models
	D1.S1	*D2.S1*	*D2.S2*	*D2.S3, D2.S4*	*Step 1*	*Step 2*	*Step 3*	
CS1	0.00, 0.00	False	True, False, 1.00	True, 1.00	1 / 1	1 / 0	–	1
CS1.V1	0.71, 0.50	False	False, False, 0.20	True, 1.00	4 / 3	3 / 0	–	4
CS1.V2	0.42, 0.20	False	False, False, 0.57	False, 0.50	4 / 4	4 / 3	3 / 0	4
CS1.V3	0.57, 0.25	False	False, False, 0.42	False, 0.33	4 / 4	4 / 3	3 / 0	4
CS2	0.00, 0.00	True	False,False, 0.00	False, 0.00	1 / 1	1 / 0	–	1
CS2.V1	0.25, 0.14	False	False,False, 0.12	False, 1.00	4 / 3	3 / 0	–	4
CS2.V2	0.62, 0.40	False	False,True, 0.00	False, 0.00	4 / 4	6 / 0	–	6
CS2.V3	0.25, 0.14	False	False,False, 0.12	False, 0.33	4 / 4	4 / 3	3 / 0	4
CS3	0.00, 0.00	False	False,False, 0.00	False, 0.00	1 / 1	1 / 0	–	1
CS3.V1	0.37, 0.16	False	False,False, 0.62	False, 1.00	4 / 4	6 / 0	–	6
CS3.V2	0.00, 0.00	False	True,False, 1.00	True, 1.00	1 / 1	1 / 0	–	1
CS3.V3	0.25, 0.14	False	False,False, 0.75	False, 0.33	4 / 4	4 / 3	3 / 0	4

Table 12.5: This table shows the results of evaluating the initial models used in our study. It includes the number of models created at each step of applying our algorithms in an iterative process, the number of smell instances (calculated by multiplying the number of generated models by the number of smells per model) that remained or were introduced in each iteration, and the final count of recommended (optimal) models.

We built a variety of methods for **RQ2** that addressed every conceivable smell and provided several correction alternatives (see Table 12.3). A search tree of potential architecture models is produced if every option is tested (such as the one shown in Table 12.4). This search tree may then be evaluated using metrics to gauge how much the basic architecture has improved and detect unresolved issues. We have demonstrated that an iterative method of employing our algorithms successively yields, within a few steps, a variety of potential architectural models that eliminate all smells detected and guarantee pattern conformity of the system architecture (see Table 12.5). The numerous ideal model versions produced by our method offer architects a great deal of design flexibility. The approach is suited to be used in a continuous delivery environment, which was one of our study's aims. This is because detection is automated, and human expertise is only used in the fix process.

12.9 Threats to Validity

The information and solutions presented in our study are based on published literature and best practices in the field. Our evaluation dataset consists of a representative collection of systems drawn from three different sources and specifically selected to demonstrate various features of IaC architectures (see Table 12.1). While our method is based on traditional component-and-connector models, widely used in the literature, and modified to include deployment aspects, it is designed to be abstract and general.

To ensure our results' accuracy and reliability, the authors' team carried out the modeling process, and all models were independently cross-checked. The authors have extensive expertise in modeling methodologies and are confident that alternative interpretations of the models would still be generally similar and compatible with our findings. However, it should be noted that our method depends on a specific modeling strategy and may not apply to all architectures.

One of the main limitations of our study is that it only considers two specific ADDs and the associated trends, metrics, and issues. In real-world architectures, it would be necessary to consider a broader range of ADDs to evaluate the architecture fully. Additionally, our measurements and tools were applied at a relatively high level of abstraction to accommodate different IaC technologies, such as Ansible, Terraform, and Puppet.

Another potential limitation of our technique is its ability to effectively address larger, more complex systems commonly found in industry but which we could not include in our research. While our method is automated to some extent, it still requires input and guidance from the architect, which may make it challenging to implement in practice. Additionally, our method cannot match the expertise and ability of a skilled architect to design a more optimal solution. This is a common limitation of generic architecture assistance techniques that we aim to address in future research.

We want to emphasize that our current approach is just a starting point for examining the issue of evaluating and improving IaC architectures. The models we have produced do not yet consider factors such as the amount of code or rewriting needed to implement them. Despite these limitations, it is still valuable to have a semi-automatic approach that can detect and analyze violations of architectural best practices, even if some of these issues may be inevitable in practice. Practitioners may not always adhere to best practices, and systems may be developed without a deliberate effort to follow them or may drift from their original specifications over time.

12.10 Conclusion and Future Work

In this chapter, we investigate the use of coupling-related architectural design decisions in infrastructure as code architectures and their impact on the system's overall design. We identify specific "smells" that may indicate issues and have developed automated detectors to locate the source of these smells within the model. We have created a set of potential solutions for each detected smell to address these smells and ensure adherence to best practices in microservice-based architectures.

We conducted three case studies on open-source microservice-based systems to evaluate our approach. We introduced smells or refactorings to 9 variants of these systems to test the performance of our smell detection algorithms in more complex scenarios. Our analysis of these models, which ranged in architectural complexity and the presence of patterns and smells, showed that our technique could eliminate smells in just three refactoring steps, most of which could be automated.

One of the main advantages of our method is its fully automated metric computation and smell detection, which makes it suitable for incorporation into a continuous delivery pipeline as an additional “architecture evaluation” step. While the proposed fixes on the IaC-based models are automated, architects still have the flexibility to make design choices and provide feedback as

needed.

We plan to expand the range of ADDs and smells supported by our method and improve it by including runtime metrics and other design components. We also plan to increase the size and complexity of our model dataset and empirically validate our approach through its use in real delivery pipelines as part of a feedback loop.

Part V
Conclusion

13 Conclusion

This doctoral thesis outlines several accomplishments related to the codification of architecture knowledge, including the development of a reusable architectural design decision model, accurate abstraction of highly polyglot systems, assessment of architecture conformance to best patterns and practices in microservices and IaC-based deployments, a unified framework for modeling and analyzing such systems, as well as automatic violation detection and fix suggestions.

In Part II, we report on a qualitative study of data management practices in microservice architectures. The study uses a model-based approach to provide a systematic and consistent representation of industry practices. The study concludes that existing knowledge sources on microservice data management practices are inconsistent and incomplete, and a systematic and unbiased study of many sources, integrated via formal modeling, can provide a more complete and rigorous account of current practices.

In Part III, we report on a study that combines design science and case study research to study methods for comprehending the component architecture of highly polyglot systems, as exemplified by state-of-the-art microservices systems. The study recommends the reusable detector approach unless a project is certain to use all detectors a very few times at most. The study also tested the adaptability of the same approach to different tasks, which are well-suited to be used in combination with the architecture abstraction detectors, potentially greatly reducing the already small manual overhead required by the method.

Furthermore, Part III presents a method that automatically assesses different tenets in microservice and IaC decisions based on a microservice system's component model. The method utilizes a minimal set of component model elements to model the key aspects of decision options, and derives metrics for each decision option. The method also employs ordinal regression analysis to develop a prediction model. Statistical analysis reveals that each decision-related metric closely aligns with manual, pattern-based assessments.

Part IV describes a method for detecting and fixing violations in microservice and IaC-based architectures by mapping them to existing metrics and defining automatic detectors. The approach is evaluated on various models and is shown to be capable of resolving violations in a small number of automated refactoring steps. The approach can be used as an additional "architecture assessment test" in a continuous delivery pipeline and is flexible enough to allow architects to make design choices.

We have identified four research questions (RQ1-RQ4) (see Section 3.2) related to microservice-based systems and proposed contributions to address these questions. RQ1 pertains to the lack of a comprehensive understanding of data management patterns and practices in microservice architecture, while RQ2 focuses on preventing architecture model drift and erosion. RQ3 involves evaluating the conformity of microservice-based systems with established patterns and best practices, and identifying potential architecture violations resulting from design decisions.

Finally, RQ4 concerns providing actionable feedback to rectify these violations and improve the system architecture.

In order to investigate our research questions, we have identified five problem areas (P1-P5) (refer to Section 3.3) that pertain to different aspects of the microservice domain. These areas encompass data gathering, modeling, analysis, assessment, refactoring, and comprehension of existing practices and theoretical foundations. To address these problem areas, we propose a set of contributions (C1-C6) discussed in Section 3.4.

The first problem area, P1, which corresponds to RQ1, is addressed by C1. C1 involves the creation of a comprehensive pattern catalog that encompasses data management patterns and practices. This catalog explores the interrelationships between patterns and practices and investigates their associated impact and drivers (Chapter 4).

The second and fourth problem areas, P2 and P4, are linked to RQ1, RQ2, and RQ3. They are addressed by C2, which introduces detectors capable of analyzing relevant sections of source code. These detectors generate model abstractions to handle the diverse nature of microservice-based systems. Furthermore, the detectors aid in reconstructing existing microservice-based systems (Chapter 5).

The third problem area, P3, pertains to RQ3 and RQ4 and is tackled by C3. C3 involves an iterative study of various knowledge sources related to microservices. This study refines a meta-model and creates multiple model instances of microservice-based systems and Infrastructure as Code (IaC)-based deployments. The investigation aims to explore the ontology and evaluate the efficiency of the meta-model (Chapters 5, 6, 7, 8 and 9).

Both problem areas P3 and P5, connected to RQ3 and RQ4, are addressed by C4 and C5. These contributions involve the implementation of detectors capable of identifying decision-based violations. Additionally, metrics introduced in Chapters 6, 7, 8 and 9 are used in conjunction with these detectors to pinpoint the elements (components and connectors) involved in specific violations (Chapters 10, 11, and 12). Furthermore, the fifth problem area, P5, related to RQ3 and RQ4, is addressed by C6. C6 provides actionable options (fixes) for each violation, aiming to enhance the system architecture as part of a feedback loop (Chapters 10, 11, and 12).

Overall, the contributions of this doctoral thesis address all the defined problems, answer the RQs, offer a comprehensive understanding of microservice-based systems and deployments, and provide valuable insights into improving architecture conformance and rectifying architecture violations.

Bibliography

[AAE16] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *IEEE 9th Int. Conf. on Service-Oriented Computing and Applications (SOCA)*, pages 44–51. IEEE, 2016.

[AAE18a] N. Alshuqayran, N. Ali, and R. Evans. Towards micro service architecture recovery: An empirical study. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 47–4709, April 2018.

[AAE18b] N. Alshuqayran, N. Ali, and R. Evans. Towards micro service architecture recovery: An empirical study. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 47–4709, 2018.

[ABDN+17] Matej Artac, Tadej Borovssak, Elisabetta Di Nitto, Michele Guerriero, and Damian Andrew Tamburri. Devops: Introducing infrastructure-as-code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 497–498, 2017.

[AFT19] Umberto Azadi, F. Fontana, and D. Taibi. Architectural smells detected by tools: a catalogue proposal. *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 88–97, 2019.

[AGG06] Edward B. Allen, Sampath Gottipati, and Rajiv Govindarajan. Measuring size, complexity, and coupling of hypergraph abstractions of software: An information-theory approach. *Software Quality Journal*, 15:179–212, 2006.

[AGG07] Edward B. Allen, Sampath Gottipati, and Rajiv Govindarajan. Measuring size, complexity, and coupling of hypergraph abstractions of software: An information-theory approach. *Software Quality Journal*, 15(2):179–212, 2007.

[AhS09] Yousef Al-houmaily and George Samaras. Two-phase commit. In *Encyclopedia of Database Systems*, pages 3204–3209. 2009.

[APW+11] Antti Airola, Tapio Pahikkala, Willem Waegeman, Bernard De Baets, and Tapio Salakoski. An experimental comparison of cross-validation techniques for estimating the area under the roc curve. *Computational Statistics & Data Analysis*, 55(4):1828–1844, 2011.

[aut21] auth0Docs. Single sign-on (sso). `https://auth0.com/docs/authenticate/single-sign-on`, 2021.

[AWS21] AWS Documentation. Security groups for your vpc. https://docs.aws.amazon.com/vpc/latest/userguide/VPC_SecurityGroups.html, 2021.

[BBM96] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[BCH+95] Barry Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby. COCOMO 2.0. *Ann. Softw. Eng.*, 1(1):1–24, 1995.

[BD02] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[BNS20] Antonio Brogi, Davide Neri, and Jacopo Soldani. Freshening the air in microservices: Resolving architectural smells via refactoring. In Sami Yangui, Athman Bouguettaya, Xiao Xue, Noura Faci, Walid Gaaloul, Qi Yu, Zhangbing Zhou, Nathalie Hernandez, and Elisa Y. Nakagawa, editors, *Service-Oriented Computing – ICSOC 2019 Workshops*, pages 17–29, Cham, 2020. Springer International Publishing.

[BWZ17] Justus Bogner, Stefan Wagner, and Alfred Zimmermann. Towards a practical maintainability quality model for service-and microservice-based systems. pages 195–198, 09 2017.

[CDMS10] Anna Corazza, Sergio Di Martino, and Giuseppe Scanniello. A probabilistic based approach towards software system clustering. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 88–96, Washington, DC, USA, 2010. IEEE Computer Society.

[Cha14] Kathy Charmaz. *Constructing grounded theory*. Sage, 2014.

[Che18] L. Chen. Microservices: Architecting for continuous delivery and devops. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 39–397, April 2018.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[Clo18] Cloud Security Alliance. Continuous monitoring in the cloud. https://cloudsecurityalliance.org/blog/2018/06/11/continuous-monitoring-in-the-cloud/, 2018.

[Clo21] Cloud Security Alliance. Five approaches for securing identity in cloud infrastructure. https://cloudsecurityalliance.org/blog/2021/05/20/five-approaches-for-securing-identity-in-cloud-infrastructure/, 2021.

[Cop96] J. Coplien. *Software Patterns: Management Briefings*. SIGS, New York, 1996.

[CS90] Juliet Corbin and Anselm L. Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13:3–20, 1990.

[DDPT20] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian Andrew Tamburri. Toward a catalog of software quality metrics for infrastructure code. *Journal of Systems and Software*, 170:110726, 2020.

[DDT20] Stefano Dalla Palma, Dario Di Nucci, and Damian A. Tamburri. Ansiblemetrics: A python library for measuring infrastructure-as-code blueprints in ansible. *SoftwareX*, 12:100633, 2020.

[DP09] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, Jul-Aug 2009.

[ELBH18] Thomas Engel, Melanie Langermeier, Bernhard Bauer, and Alexander Hofmann. Evaluation of microservice architectures: A metric and tool-based approach. In Jan Mendling and Haralambos Mouratidis, editors, *Information Systems in the Big Data Era*, pages 74–89, Cham, 2018. Springer International Publishing.

[FBKL17] Markus Philipp Fischer, Uwe Breitenbücher, Kálmán Képes, and Frank Leymann. Towards an approach for automatically checking compliance rules in deployment models. In *Proceedings of The Eleventh International Conference on Emerging Security Information, Systems and Technologies (SECURWARE)*, pages 150–153. Xpert Publishing Services (XPS), 2017.

[FEH15] Jr. Frank E. Harrell. *Regression Modeling Strategies: With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis*. Springer, 2nd edition, 2015.

[FKH17] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. Software Landscape and Application Visualization for System Comprehension with ExplorViz. *Information and Software Technology*, 87:259–277, July 2017.

[FML17] P. D. Francesco, I. Malavolta, and P. Lago. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 21–30, April 2017.

[Fow11] Martin Fowler. Command and Query Responsibility Segregation (CQRS) pattern, 2011.

[GAK99] George Yanbing Guo, Joanne M Atlee, and Rick Kazman. A software architecture reconstruction method. In *Software Architecture*, pages 15–33. Springer, 1999.

[GCD+17] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle. Microart: A software architecture recovery tool for maintaining microservice-based systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 298–302, 2017.

[GCF+17] G. Granchelli, M. Cardarelli, P. D. Francesco, I. Malavolta, L. Iovino, and A. D. Salle. Towards recovering the software architecture of microservice-based systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 46–53, April 2017.

[GFM17] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. Guidelines for including the grey literature and conducting multivocal literature reviews in software engineering. *CoRR*, 2017.

[GIM13] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. A comparative analysis of software architecture recovery techniques. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 486–496, Piscataway, NJ, USA, 2013. IEEE Press.

[GKN15] Ian Gorton, John Klein, and Albert Nurgaliev. Architecture knowledge for evaluating scalable databases. In *Proc. of the 12th Working IEEE/IFIP Conference on Software Architecture*, pages 95–104, 2015.

[GL14] Dharmalingam Ganesan and Mikael Lindvall. Adam: External dependency-driven architecture discovery and analysis of quality attributes. *ACM Trans. Softw. Eng. Methodol.*, 23(2):17:1–17:51, April 2014.

[GM14] Maayan Goldstein and Dany Moshkovich. Improving software through automatic untangling of cyclic dependencies. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, page 155–164, New York, NY, USA, 2014. Association for Computing Machinery.

[Goo21] Google Cloud. Using api keys. `https://cloud.google.com/docs/authentication/api-keys`, 2021.

[GPEM09a] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 255–258, 2009.

[GPEM09b] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Toward a catalogue of architectural bad smells. In Raffaela Mirandola, Ian Gorton, and Christine Hofmeister, editors, *Architectures for Adaptive Software Systems*, pages 146–162, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[GPEM09c] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Toward a catalogue of architectural bad smells. In Raffaela Mirandola, Ian Gorton, and Christine Hofmeister, editors, *Architectures for Adaptive Software Systems*, pages 146–162, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[GPM+11] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Yuanfang Cai. Enhancing architectural recovery using concerns. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 552–555, 2011.

[GS67] Barney G. Glaser and Anselm L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. de Gruyter, 1967.

[Gup17] Arun Gupta. Microservice design patterns. http://blog.arungupta.me/microservice-design-patterns/, 2017.

[HCN98] R. Harrison, S. J. Counsell, and R. V. Nithi. An evaluation of the mood set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, 24(6):491–496, 1998.

[HF10] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[HNZ17] Thomas Haitzer, Elena Navarro, and Uwe Zdun. Reconciling software architecture and source code in support of software evolution. *Journal of Systems and Software*, 123:119–144, 2017.

[HS17] Wilhelm Hasselbring and Guido Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. In *Proceedings 2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 243–246, Gothenburg, Sweden, April 2017. IEEE.

[HW03a] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns*. Addison-Wesley, 2003.

[HW03b] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

[HWB17] Stefan Haselböck, Rainer Weinreich, and Georg Buchgeher. Decision models for microservices: Design areas, stakeholders, use cases, and requirements. In *Software Architecture. ECSA 2017*, Lecture Notes in Computer Science, vol 10475, pages 155–170. Springer International Publishing, 2017.

[HZ14] Thomas Haitzer and Uwe Zdun. Semi-automated architectural abstraction specifications for supporting software evolution. *Science of Computer Programming*, 90:135–160, 2014.

[HZHD15] Carsten Hentrich, Uwe Zdun, Vlatka Hlupic, and Fefie Dotsika. An Approach for Pattern Mining Through Grounded Theory Techniques and Its Applications to Process-driven SOA Patterns. In *Proc. of the 18th European Conference on Pattern Languages of Program*, pages 9:1–9:16, 2015.

[KBKL18] Christoph Krieger, Uwe Breitenbücher, Kálmán Képes, and Frank Leymann. An Approach to Automatically Check the Compliance of Declarative Deployment Models. In *Papers from the 12th Advanced Summer School on Service-Oriented Computing (SummerSoC 2018)*, pages 76–89. IBM Research Division, Oktober 2018.

[KGR+21] Indika Kumara, Martín Garriga, Angel Urbano Romeu, Dario Di Nucci, Fabio Palomba, Damian Andrew Tamburri, and Willem-Jan van den Heuvel. The do's and don'ts of infrastructure code: A systematic gray literature review. *Information and Software Technology*, 137:106593, 2021.

[KH19] Holger Knoche and Wilhelm Hasselbring. Drivers and barriers for microservice adoption – a survey among professionals in germany. *Enterprise Modelling and Information Systems Architectures (EMISAJ) – International Journal of Conceptual Modeling*, 14(1):1–35, 2019.

[KMNL06] Jens Knodel, Dirk Muthig, Matthias Naab, and Mikael Lindvall. Static evaluation of software architectures. *Software Maintenance and Reengineering, European Conference on*, pages 279–294, 2006.

[Kub21] Kubernetes Documentation. Ingress traffic control. `https://kubernetes.io/docs/concepts/services-networking/ingress/`, 2021.

[KVM+20] Indika Kumara, Zoe Vasileiou, Georgios Meditskos, Damian A. Tamburri, Willem-Jan Van Den Heuvel, Anastasios Karakostas, Stefanos Vrochidis, and Ioannis Kompatsiaris. Towards semantic detection of smells in cloud infrastructure code. In *Proceedings of the 10th International Conference on Web Intelligence, Mining and Semantics*, WIMS 2020, page 63–67, New York, NY, USA, 2020. Association for Computing Machinery.

[LCCM16] D. M. Le, C. Carrillo, R. Capilla, and N. Medvidovic. Relating architectural decay and sustainability of software systems. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 178–181, 2016.

[LF04] James Lewis and Martin Fowler. Microservices: a definition of this new architectural term. `http://martinfowler.com/articles/microservices.html`, March 2004.

[LLSM18] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural decay in open-source software. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 176–17609, 2018.

[LSZ12] Ioanna Lytra, Stefan Sobernig, and Uwe Zdun. Architectural Decision Making for Service-Based Platform Integration: A Qualitative Multi-Method Study. In *Joint 10th Working IEEE/IFIP Conf. on Software Architecture & 6th European Conf. on Software Architecture (WICSA/ECSA), Helsinki, Finland*. IEEE Comp. Soc., 2012.

[Mar04] R. Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 350–359, 2004.

[MC] Caroline Davis Michael Cowles. On the origins of the .05 level of statistical significance. In *American Psychologist, 37(5), 553–558*.

[McC04] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Best Practices for Developers. Microsoft Press, Redmond, WA, 2 edition, 2004.

[MMW02] Kim Mens, Tom Mens, and Michel Wermelinger. Maintaining software through intentional source-code views. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, SEKE '02, pages 289–296, New York, NY, USA, 2002. ACM.

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '95, pages 18–28, New York, NY, USA, 1995. ACM.

[MNS01] G.C. Murphy, D. Notkin, and K.J. Sullivan. Software reflexion models: bridging the gap between design and implementation. *IEEE Trans. Softw. Eng.*, 27(4):364–380, 2001.

[Mor15] Kief Morris. *Infrastructure as Code: Dynamic Systems for the Cloud*. O'Reilly, 2015.

[New15] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly, 2015.

[NSZB19] Davide Neri, Jacopo Soldani, Olaf Zimmermann, and Antonio Brogi. Design principles, architectural smells and refactorings for microservices: a multivocal review. *SICS Software-Intensive Cyber-Physical Systems*, 35(1-2):3–15, Sep 2019.

[Nyg07] Michael Nygard. *Release It! Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007.

[NZP+19] Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Daniel Schall, Fei Li, and Sebastian Meixner. Supporting architectural decision making on data management in microservice architectures. In *13th European Conference on Software Architecture (ECSA) - 2019*, September 2019.

[NZP+20a] Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Sebastian Meixner, and Sebastian Geiger. Assessing architecture conformance to coupling-related patterns and practices in microservices. In *14th European Conference on Software Architecture (ECSA), 2020*, September 2020.

[NZP+20b] Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Sebastian Meixner, and Sebastian Geiger. Metrics for assessing architecture conformance to microservice architecture patterns and practices. In *18th International Conference on Service Oriented Computing (ICSOC 2020)*, December 2020.

[NZP+21] Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Patric Genfer, Sebastian Geiger, Sebastian Meixner, and Wilhelm Hasselbring. Detector-based component

model abstraction for microservice-based systems. *Computing*, 103:2521–2551, August 2021.

[NZPG21] Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, and Sebastian Geiger. Semi-automatic feedback for improving architecture conformance to microservice patterns and practices. In *18th IEEE International Conference on Software Architecture(ICSA 2021)*, March 2021.

[NZSB22] Evangelos Ntentos, Uwe Zdun, Jacopo Soldani, and Antonio Brogi. Assessing architecture conformance to coupling-related infrastructure-as-code best practices: Metrics and case studies. In *16th European Conference on Software Architecture*, Software Architecture - 16th European Conference, ECSA 2022, Prague, Czech Republic, September 19?23, 2022, Proceedings, pages 101–116, September 2022.

[Okt21] Okta. Token-based authentication. https://www.okta.com/identity-101/what-is-token-based-authentication/, 2021.

[OWA21a] OWASP Cheat Sheet Series. Authentication cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html#logging-and-monitoring, 2021.

[OWA21b] OWASP Cheat Sheet Series. Infrastructure as code security cheatsheet. https://cheatsheetseries.owasp.org/cheatsheets/Infrastructure_as_Code_Security_Cheat_Sheet.html, 2021.

[OWA21c] OWASP Cheat Sheet Series. Transport layer protection cheat sheett. https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Cheat_Sheet.html#ssl-vs-tls, 2021.

[PD18] Waligora M. Piasecki, J. and V Dranseika. Google search as an additional source in systematic reviews. *Sci Eng Ethics 24*, 55(4):809–810, 2018.

[Per17] Matthieu Perrin. Overview of existing models. In Matthieu Perrin, editor, *Distributed Systems*, pages 23–52. Elsevier, 2017.

[PGLCX16] Marcus Pendleton, Richard Garcia-Lebron, Jin-Hee Cho, and Shouhuai Xu. A survey on systems security metrics. *ACM Comput. Surv.*, 49(4), December 2016.

[PJ16] Claus Pahl and Pooyan Jamshidi. Microservices: A systematic mapping study. In *6th International Conference on Cloud Computing and Services Science*, pages 137–146, 2016.

[PRG18] R. Pietrantuono, S. Russo, and A. Guerriero. Run-time reliability estimation of microservice architectures. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 25–35, Oct 2018.

[PTV+10] Leonardo Passos, Ricardo Terra, Marco Tulio Valente, Renato Diniz, and Nabor das Chagas Mendonca. Static architecture-conformance checking: An illustrative overview. *IEEE software*, 27(5):82–89, 2010.

[PW92] Dewayne E Perry and Alexander L Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[PW09] Cesare Pautasso and Erik Wilde. Why is the web loosely coupled?: a multi-faceted metric for service design. In *18th Int. Conf. on World wide web*, pages 911–920. ACM, 2009.

[PZA+17] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis. Microservices in practice, part 1: Reality check and service design. *IEEE Software*, 34(1):91–98, Jan 2017.

[PZL08] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. RESTful Web Services vs. Big Web Services: Making the right architectural decision. In *Proc. of the 17th World Wide Web Conference (WWW)*, pages 805–814, April 2008.

[Ric17] Chris Richardson. A pattern language for microservices. `http://microservices.io/patterns/index.html`, 2017.

[RKH14] Aditya Kaushal Ranjan, Vijay Kumar, and Muzzammil Hussain. Security analysis of tls authentication. In *2014 International Conference on Contemporary Computing and Informatics (IC3I)*, pages 1356–1360, 2014.

[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)*, pages 21–30, March 2019.

[Sar03] Kamran Sartipi. Software architecture recovery based on pattern matching. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 293–, Washington, DC, USA, 2003. IEEE Computer Society.

[SBF+19] Karoline Saatkamp, Uwe Breitenbücher, Michael Falkenthal, Lukas Harzenetter, and Frank Leymann. An approach to determine & apply solutions to solve detected problems in restructured deployment models using first-order logic. In *International Conference on Cloud Computing and Services Science*, 2019.

[SBKL18] Karoline Saatkamp, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Application scenarios for automated problem detection in tosca topologies by formalized patterns. In *Papers From the 12th Advanced Summer School on Service-Oriented Computing (SummerSOC'18)*, pages 43–53. IBM Research Division, 2018.

[SBKL19] Karoline Saatkamp, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. An approach to automatically detect problems in restructured deployment models based on formalizing architecture and design patterns. *SICS Softw.-Inensiv. Cyber-Phys. Syst. 34*, page 85–97, 2019.

[SFS16] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does your configuration code smell? In *Proceedings of the 13th International Conference on Mining Software*

Repositories, MSR '16, page 189–200, New York, NY, USA, 2016. Association for Computing Machinery.

[sid17] Sidecar pattern, 2017.

[Sko19] Jason Skowronski. Best practices for event-driven microservice architecture. https://hackernoon.com/best-practices-for-event-driven-microservice-architecture-e034p21lk, 2019.

[SMS20] Thodoris Sotiropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. Practical fault detection in puppet programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 26–37, New York, NY, USA, 2020. Association for Computing Machinery.

[SROV06] Christoph Stoermer, Anthony Rowe, Liam O'Brien, and Chris Verhoef. Model-centric software architecture reconstruction. *Softw. Pract. Exp.*, 36(4):333–363, April 2006.

[SSL18] Julian Schwarz, Andreas Steffens, and Horst Lichter. Code smells in infrastructure as code. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 220–228, 2018.

[The21] The Security Skeptic. Firewall best practices - egress traffic filtering. https://securityskeptic.typepad.com/the-security-skeptic/firewall-best-practices-egress-traffic-filtering.html, 2021.

[TL18] D. Taibi and V. Lenarduzzi. On the definition of microservice bad smells. *IEEE Software*, 35(3):56–62, May 2018.

[vDB11] Markus von Detten and Steffen Becker. Combining clustering and pattern detection for the reengineering of component-based software systems. In *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*, QoSA-ISARCS '11, pages 23–32, New York, NY, USA, 2011. ACM.

[vdBHVG18] Eduard van der Bent, Jurriaan Hage, Joost Visser, and Georgios Gousios. How good is your puppet? an empirically defined and validated quality model for puppet. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 164–174, 2018.

[VDHK+04] Arie Van Deursen, Christine Hofmeister, Rainer Koschke, Leon Moonen, and Claudio Riva. Symphony: View-driven software architecture reconstruction. In *4th Working IEEE/IFIP Conf. on Software Architecturen(WICSA 2004)*, pages 122–132. IEEE, 2004.

[vHAH12] U. van Heesch, P. Avgeriou, and R. Hilliard. A documentation framework for architecture decisions. *J. Syst. Softw.*, 85(4):795 – 820, 2012.

[VLS14] M. Vianden, H. Lichter, and A. Steffens. Experience on a microservice-based reference architecture for measurement systems. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 183–190, Dec 2014.

[WBH+20] Michael Wurster, Uwe Breitenbücher, Lukas Harzenetter, Frank Leymann, and Jacopo Soldani. Tosca lightning: An integrated toolchain for transforming tosca light into production-ready deployment technologies. In Nicolas Herbaut and Marcello La Rosa, editors, *Advanced Information Systems Engineering*, pages 138–146, Cham, 2020. Springer International Publishing.

[WBSB] Marcel Weller, Uwe Breitenbücher, Sandro Speth, and Steffen Becker. The deployment model abstraction framework.

[WHH03] Claes Wohlin, Martin Höst, and Kennet Henningsson. *Empirical Research Methods in Software Engineering*, pages 7–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[Wie14] Roel J Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[Wol16] Eberhard Wolff. *Microservices: flexible software architecture*. Addison-Wesley Professional, 2016.

[WRH+12] Claes Wohlin, Per Runeson, Martin Hoest, Magnus C. Ohlsson, Bjoern Regnell, and Anders Wesslen. *Experimentation in Software Engineering*. Springer, 2012.

[Yin02] Robert K. Yin. Case study research: Design and methods, 3rd edition (applied social research methods, vol. 5). 12 2002.

[ZGK+07] Olaf Zimmermann, Thomas Gschwind, Jochen Küster, Frank Leymann, and Nelly Schuster. Reusable architectural decision models for enterprise application development. In *Int. Conf. on the Quality of Software Architectures*, pages 15–32. Springer, 2007.

[Zim17] Olaf Zimmermann. Microservices tenets. *Computer Science - Research and Development*, 32(3):301–310, Jul 2017.

[ZKL+09] Olaf Zimmermann, Jana Koehler, Frank Leymann, Ronny Polley, and Nelly Schuster. Managing architectural decision models with dependency relations, integrity constraints, and production rules. *J. Syst. Softw.*, 82(8):1249–1267, 2009.

[ZNL17] Uwe Zdun, Elena Navarro, and Frank Leymann. Ensuring and assessing architecture conformance to microservice decomposition patterns. In Michael Maximilien, Antonio Vallecillo, Jianmin Wang, and Marc Oriol, editors, *Service-Oriented Computing*, pages 411–429, Cham, 2017. Springer International Publishing.

[ZSZ+18] Uwe Zdun, Mirko Stocker, Olaf Zimmermann, Cesare Pautasso, and Daniel Lübke. Supporting architectural decision making on quality aspects of microservice apis. In *16th International Conference on Service-Oriented Computing (ICSOC 2018)*, Hangzhou, Zhejiang, China, November 2018. Springer.

[ZSZ+19] Olaf Zimmermann, Mirko Stocker, Uwe Zdun, Daniel Luebke, and Cesare Pautasso. Microservice API patterns. https://microservice-api-patterns.org, 2019.

[ZZGL08] O. Zimmermann, U. Zdun, T. Gschwind, and F. Leymann. Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method. In *Software Architecture, 2008. WICSA 2008. Seventh Working IEEE/IFIP Conference on*, pages 157–166, 2008.