FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Dissertation

Designing Evolvable Web Services

Paul Schmiedmayer

FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Designing Evolvable Web Services

Paul Schmiedmayer

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Prof. Dr. Florian Matthes
Prüfende der Dissertation:	1. Prof. Dr. Bernd Brügge
	2. Prof. Dr. Stefan Wagner

Die Dissertation wurde am 14.02.2022 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 30.05.2022 angenommen.

Abstract

The development of distributed systems presents numerous challenges. Advancements in web technologies have made web development a de-facto standard in distributed systems. Designing web services involves incorporating various middleware- and protocol-types for inter-process communication. Deploying distributed systems is influenced by deployment mechanisms and several architectural styles, such as cloud- and fog-based architectures. Especially the rate of progress in these technologies and execution environments has made maintenance and evolution of web services a complicated endeavor.

The focus of this dissertation lies on a framework that enables developers to design evolvable web services. The framework consists of a web service metamodel and the Apodini ecosystem to provide mechanisms and tools in three areas: web service interface evolution, web service API evolution, and web service deployment evolution. The metamodel builds the foundation to investigate web API change patterns and deployment evolution-related challenges in the Web of Things, Function as a Service, and observability domains.

The Apodini ecosystem provides an internal domain-specific language to enable web service interface type-independent development. It consists of three major components. Interface Exporters enable an evolvable extension mechanism to support multiple web API types in the lifetime of a web service. Apodini Migrator generates migration guides and stable client libraries to mitigate web service API evolution. Apodini Deployer forms the basis for Deployment Providers that enable constraint-based deployment mechanisms and web service partitioning for cloud- and fog-based architectures.

The framework was empirically validated using a design science approach based on single-case mechanism experiments. The extensibility of Apodini was proven by using five web API types: gRPC, WebSocket, GraphQL, HTTP, and RESTful APIs. The applicability of Apodini evolution mechanisms was demonstrated in five experiments covering a wide range of application domains such as sports and health science, mobile applications, smart city Internet of Things environments, and water quality measurement systems.

Zusammenfassung

Die Entwicklung verteilter Systeme ist mit vielen Herausforderungen verbunden. Fortschritte in Webtechnologien haben Webdienste zu einem De-facto-Standard in verteilten Systemen gemacht. Das Entwerfen dieser Webdienste macht es notwendig, eine Vielzahl von Middleware- und Protokolltypen für die prozessübergreifende Kommunikation einzubeziehen. Die Bereitstellung dieser verteilten Systeme wird durch die breite Auswahl an Bereitstellungsmechanismen und Architekturstilen, wie cloud- und fogbasierte Architekturen, beeinflusst. Insbesondere die Geschwindigkeit, mit der Webtechnologien voranschreiten, hat die Wartung und Weiterentwicklung von Webdiensten zu einem komplexen Unterfangen gemacht.

Im Mittelpunkt dieser Dissertation steht eine Grundstruktur, die es Entwicklern ermöglicht, weiterentwickelbare Webdienste zu entwerfen. Sie besteht aus einem Webdienst-Metamodell und dem Apodini-Ökosystem, das Mechanismen und Werkzeuge in drei Bereiche aufteilt: Webdienst-Schnittstellen-Evolution, Web-API-Evolution und Webdienst-Bereitstellungs-Evolution. Das Webdienst-Metamodell bildet die Grundlage der Untersuchung von Web-API-Änderungsmustern und Herausforderungen im Zusammenhang mit der Entwicklung von Webdiensten in den Bereichen Web der Dinge, Function as a Service und Beobachtung von Webdiensten.

Das Apodini-Ökosystem bietet eine interne domänenspezifische Sprache, die eine von Webdienst-Schnittstellen unabhängige Entwicklung ermöglicht. Es besteht aus drei Hauptkomponenten. Interface-Exporter ermöglichen einen Erweiterungsmechanismus zur Unterstützung mehrerer Web-API-Typen während der Lebensdauer eines Webdienstes. Apodini-Migratoren generieren automatisch Migrations-Leitfäden und stabile Client-Bibliotheken, um die Änderungen von Webdienst-APIs abzufangen. Apodini-Deployer bildet die Grundlage für Deployment-Provider, die ein randbedingungsbasiertes Bereitstellen und ein Partitionieren von Webdiensten für cloud- und fogbasierte Architekturen ermöglichen.

Apodini wurde empirisch mit einem Design-Science-Ansatz validiert, der auf Einzelfall-Mechanismus-Experimenten basiert. Die Erweiterbarkeit von Apodini wurde anhand von fünf Web-API-Typen belegt: gRPC, WebSocket, GraphQL, HTTP und RESTful-APIs. Die Anwendbarkeit der Apodini-Mechanismen wurde in fünf Experimenten bestätigt, die ein breites Spektrum von Anwendungsbereichen abdecken: Sport- und Gesundheitswissenschaften, mobile Anwendungen, sowie Internet-der-Dinge-Systeme für intelligente Städte und Wasserqualitätsmessung.

Acknowledgements

First, I would like to express my deepest gratitude to Professor Bernd Brügge. His enthusiasm for teaching and software engineering research is truly inspiring and has motivated me throughout my time at the Technical University of Munich. I am grateful for the countless possibilities he enabled during my studies and time as a doctoral candidate. I can not imagine having a better supervisor and mentor. Furthermore, I want to thank Professor Stefan Wagner, Professor Stephan Jonas, and Professor Pramod Bhatotia for their input and encouragement for this dissertation.

I want to thank my colleagues for the great conversations we had along the way, and the friendships we made that are here to last. I want to especially thank Lukas Alperowitz and Dora Dzvonyar for giving me the opportunity to join the chair. Thank you to Jan Philip Bernius, Dominic Henze, Nadine von Frankenberg und Ludwigsdorff, and Lara Marie Reimer for their continuous support, the great times, and the positive energy we shared. In addition, I want to thank Mariana Avezum, Florian Bodlée, Jan Ole Johanßen, Marko Jovanović, Jens Klinker, Stephan Krusche, Andreas Seitz, and all colleagues I had the pleasure to work with for the great collaborations we had throughout different projects and courses we organized. I am also grateful for Helma Schneider and the system administrators at our chair, particularly Florian Angermeir, Matthias Linhuber, Vincent Picking, Robert Jandow, Philipp Eichstetter, and the multimedia group for always being available and making sure that everything is working and well organized.

I also want to thank all students that supported my research along the way. The Apodini ecosystem would not be possible without the support of Andreas Bauer, Eldi Cano, Lukas Kollmer, Paul Kraft, Max Obermeier, Mathias Quintero, Andre Weinkötz, Philipp Zagar, and all Apodini contributors, including the members of the Server-Side Swift practical course.

Last but not least, I want to thank my friends and family for their continuous support. Your friendship, encouragement, and advice mean the world to me. This dissertation would not have been possible without you.

Contents

Abstract v

Zusammenfassung vii

Conventions xv

I Prelude 1

1 Introduction 3

1.1 Research Process and Research Goals 9

1.1.1 Design Problems 13

1.1.2 Knowledge Questions 14

1.2 Dissertation Organization 16

2 Knowledge Context 21

2.1 Software Engineering 21

2.2 Distributed Systems 25

2.2.1 Web Service Interface Types 27

2.2.2 Domain-Specific Languages for Web Services 31

II Problem Investigation 35

3 Web Service Interface Evolution 37

3.1 Related Work 40

3.1.1 Service Definition Languages 40

3.1.2 Adapters 42

3.1.3 Model-Based Approaches 44

3.2 Web Service Interface Metamodel 46

3.2.1 Web Service Interface Metamodel 47

3.2.2 Metamodel Conformant Web API Types 49

4 **Web Service API Evolution** 55
4.1 Related Work 58
4.1.1 Local API Evolution 59
4.1.2 Web API Evolution Strategies 60
4.1.3 Web API Change Identification 61
4.1.4 Web API Evolution Migration 63
4.1.5 Protocol-Enabled API Evolution 64
4.2 Web Service API Change Classification 66
4.2.1 Web Service API Evolution Patterns 66
4.2.2 API Change Classifications Comparisons 68

5 **Web Service Deployment Evolution** 73
5.1 Web Service Deployment Evolution Domains 76
5.1.1 Cloud-Based Deployments 77
5.1.2 Observability 79
5.1.3 Web of Things 80
5.2 Web Service Metadata Annotations 83
5.2.1 Web Service Metadata Annotation Model 83
5.2.2 Web Service Metadata Annotation Domains 85

III Treatment Design 91

6 **System Design** 93
6.1 Design Goals 95
6.2 Control Flow 99
6.2.1 Interface Exporter 99
6.2.2 Migrator 100
6.2.3 Deployer 101
6.3 Software Architecture 103
6.3.1 Apodini 103
6.3.2 Interface Exporter 104
6.3.3 Migrator 106
6.3.4 Deployer 107

7 **Object Design** 109
7.1 Domain-Specific Language Components 110
7.1.1 Swift-based Apodini DSL Interface 112
7.1.2 Kotlin-based Apodini DSL Interface 114
7.2 Semantic Model 115
7.3 Migration Guide 117

7.4 Deployment Structure 121
7.5 Cross Deployment Node Communication 123

IV Treatment Validation **125**

8 Apodini Interface Exporter **127**
8.1 gRPC Interface Exporter 129
8.2 WebSocket Interface Exporter 134
8.3 GraphQL Interface Exporter 138
8.4 HTTP Interface Exporter 142
8.5 RESTful Interface Exporter 147
8.5.1 OpenAPI Document Generation 153
8.6 Summary 155

9 Web Service Instantiations **157**
9.1 Basketball Player Health Monitoring System 159
9.2 Event Management Plattform 162
9.3 Expense and Income Tracking Application 168
9.3.1 Localhost Deployment Provider 171
9.3.2 AWS Lambda Deployment Provider 172
9.4 Smart City IoT System 175
9.5 Water Quality Measurement System 180
9.6 Summary 185

V Epilog **187**

10 Conclusion and Future Work **189**

List of Figures **191**

List of Tables **193**

List of Listings **195**

Bibliography **197**

Conventions

This work was produced in conformance to the *Code of Conduct for Safeguarding Good Academic Practice and Procedures in Cases of Academic Misconduct at the Technical University of Munich* [274] as well as the *Technical University of Munich Citation Guide* [275]. Hyperlinks in footnotes were all last accessed on February 9, 2022. The conventions are based on the conventions introduced in *Continuous User Understanding in Software Evolution* by Jan Ole Johanßen [144] and a dissertation template provided by Jan Ole Johanßen.

The dissertation is written in an inclusive and gender-neutral writing style. We use the singular gender-neutral personal pronoun ”they” and derivative forms such as ”them”, ”their”, ”theirs”, and ”themselves” in this dissertation.

Titles of publications and referenced artifacts mentioned in the text are written in *italic font*. Direct quotes are written in *”italic font surrounded by quotation marks”*. Introduced terms are written in a **bold** font. We use a `monospace font` to distinguish code fragments.

Multiplicity in UML models use a default value of 1 if no multiplicity is defined.

We understand that company names and product names mentioned in this dissertation are registered trademarks and refer to further information using footnotes if beneficial and omit trademark symbols.

Definitions are presented in a dark gray box with a full border:

Definition X – Definition Convention:
A defintion.

Research goals are highlighted using a box in the following style:

Goal Convention

Refinements of important extracts of text such as design problems and knowledge questions are highlighted in the following style:

Refinement Convention

Part I

Prelude

THE prelude introduces the research process and knowledge context of this dissertation. Chapter 1 provides an introduction to web service evolution and motivates the problem context. The chapter features foundational definitions for types of web service evolution and conflicting stakeholder goals related to web service evolution.

Based on these insights, we present our research goals, design problems, and knowledge questions according to the design science methodology detailed in *Design Science Methodology for Information Systems and Software Engineering* by Roel J. Wieringa [309]. The dissertation outline combining the software engineering approach defined in ISO/IEC/IEEE 12207 [15] and design science methodologies defined by Wieringa [309] is detailed in Section 1.2.

Chapter 2 provides an overview of the knowledge context surrounding the research described in this dissertation. The chapter provides an overview of the used software engineering techniques and fundamental knowledge about distributed systems development.

Chapter 1

Introduction

Software engineering consists of rational-driven problem-solving processes that incorporate modeling and knowledge acquisition to deal with the complexity of developing software systems [62]. The processes are evolutionary as software has to continuously adapt and evolve to incorporate changes in requirements, application domains, and technologies during its lifetime [267]. Software evolution encompasses the challenges of adapting, extending, and maintaining a system's capabilities by incorporating changes that affect the software system [232, 267].

Lehman defines eight laws of software evolution that summarize the forces software systems are exposed to and shape the evolutionary process of software engineering [173]. The laws express the need for a software system to continuously change if it should continue to stay useful, which results in an increased complexity if it is not counteracted [171]. Service-oriented computing counteracts increasing complexity by creating computational abstractions using services that are invocable in a technology-neutral way, are loosely coupled, and location-transparent [56, 211].

Service-oriented architectures promote designing software systems around clearly partitioned and abstracted resources, creating the concepts of service providers, service clients, and service registries [89, 211]. Bogner et al. demonstrate that service and microservice-based architectures both encourage the usage of principles and design patterns beneficial to software maintainability and evolvability [46, 47, 48, 49, 50]. Enabled by web technologies, web services are a de-facto standard in service-oriented architectures, offered using web-based middleware- and protocol-types [19, 89]. The UDDI consortium defines web services as *"self-contained, modular business applications that have open, Internet-oriented, standards-based interfaces"* [283].

Web services provide several advantages when dealing with software evolution as *"any changes to the service implementations that do not impact their interfaces are completely transparent to the overall system"* [100], but also come with evolution-related challenges *"due to the fundamentally distributed nature of serviceoriented systems"* [100].

We establish three main stakeholder groups interacting with web services to explore these challenges: web service developers, web service hosting providers, and web service clients. Similar classifications in web service evolution research such as the influence factors by Treiber et al. (hosting environment, consumer/service integrator, developer, and provider) [279]; or the perspectives on web services by Treiber et al. (provider, developer, service integrator, and user) [280] support our classification. Bondel et al. also identify similar three main stakeholder groups for web API management complemented with the end-user stakeholder group and addition stakeholder groups interacting with the API provider [51]. In addition to the three stakeholder groups, web service evolution is also influenced by constraints, functional and non-functional requirements such as scalability, observability, privacy, security, sustainability, and other aspects discussed in this dissertation.

Definition 1 – Web Service Developer:
A web service developer designs and builds web services. The web service developer is involved in all software engineering activities, from understanding the problem domain, requirements elicitation, system design, implementation to testing and deploying the web service.

A **web service developer** can have multiple specializations, such as a software architect, requirements engineer, or security expert. While all these specialization are essential in developing web services, the web service developer stakeholder encompasses all these specializations, clearly defining the boundaries to the other two main stakeholder groups. For this dissertation, we use the terms **web service developer** and **developer** interchangeably.

While web services have been traditionally deployed on-premise, in the past decade, emerging service models such as Software as a Service (SaaS), Platform as a Service (PaaS), and Function as a Service (FaaS) do no longer require managing the underlying hardware infrastructure, by taking advantage of the benefits of cloud computing technologies like resource pooling [188, 209]. Virtual Machines (VMs), container technologies, and orchestration tools build the backbone of modern deployment strategies and abstract away hardware configurations [209, 210]. A **web service hosting provider** supplies cloud deployment models as a service, so service developers do not need to provide and maintain this infrastructure.

Definition 2 – Web Service Hosting Provider:
A web service hosting provider provides and maintains the infrastructure of hosting the developed web service as defined by the web service deployment structure.

For this dissertation, the terms **web service hosting provider** and **hosting provider** are used interchangeably. The web service developer defines and evolves the web service deployment structure in accordance with the deployment possibilities of web service hosting providers and constraints defined by potential execution environments. A UML deployment diagram, e.g., a hardware/software mapping developed during system design, can express this deployment structure [62]. A hardware/software mapping defines the responsibility of each hardware node or virtualization of a hardware node and how the communication between these nodes is realized [62]. As the communication between nodes incorporating software components is essential during web service development, we extend the functionality of a hardware/software mapping to define the middleware and protocol mapping between the components. As a result, we rename the hardware/software mapping to an execution-environment/software/protocol mapping. This highlights the importance of middleware- and protocol-types used to communicate between the hardware-independent execution environments enabled by hardware virtualization techniques.

Definition 3 – Deployment Structure:
A deployment structure defines the static and dynamic mapping of software components to execution environments and can be described by execution-environment/software/protocol mappings.

The **deployment structure** and documenting execution-environment/software/-protocol mappings are evolving based on requirements such as scalability, computation power, energy, sustainability and other and deployment-related constraints. Software architectures such as fog computing architectures require dynamic execution-environment/software/protocol mappings due to the need for ubiquitous, scalable, dynamically reconfigurable, and context-aware redeployments based on changes in the execution environments [139]. Section 2.2 provides a historical overview of distributed systems development leading up to edge and fog computing while Chapter 5 provides in-depth insights into the problem context around web service deployment evolution.

Definition 4 – Web Service Client:
A web service client discovers, consumes, and depends on the web service by interacting with its interface. The client queries the service and might produce content using web-based middleware- and protocol-types.

The **web service client** is the third stakeholder group we define interacting with web services. In contrast to the web service developer and the web service hosting provider, the web service client only interacts with the web service using its

interface and does not interact with the underlying models, source code, and executables. A **web service interface** defines the abstract definition of functionality defining the interface between the web service and the web service client [19, 36]. The web service interface is instantiated using a **web service application programming interface (API)** [19]. In contrast to the abstract definition of the web service interface, we define a web service API as bound to specific web-based middleware- and protocol-types.

Definition 5 – Web Service Evolution:
Web service evolution encompasses all modifications to the web service during its lifetime ranging from changes to the source code, adding or removing middleware- and protocol-types, changes to the deployment structures, and changes to its web service interface.

All three stakeholders play an essential role when discussing web service evolution. Web service evolution is induced by several forces, such as changes in the problem domain, changes in requirements and constraints in multi-level feedback systems offering user feedback, and automated collection of insights using monitoring techniques [173]. Monitoring system behavior is an established and decades-old technique that enables developers to extract actionable insights gathered from information collected about the monitored software system [264]. Collecting logs, metrics, and traces provides indicators to continuously observe distributed systems and are referred to as the three pillars of observability [268]. These tools help to provide feedback to the web service developer and enable evolving web services based on actionable insights generated from the collected indicators.

To track evolution, changes in the source code should be traceable to changes in functional or non-functional requirements or constraints. Requirements traceability is a well-established research field in software engineering, establishing a clear link between development artifacts and changes in requirements [116]. Tracing non-functional requirements in web service evolution is challenging as they are not contained in a single artifact but are distributed across different development and deployment-related artifacts as the web services are evolving [246].

Changes and development in web technologies and middleware- and protocol-types lead to web service evolution. Table 1.1 shows a small subset of modern and historic web API types. The tabular overview highlights the rapidly moving developments of web API, middleware, and protocol types used in web development. The advancements in web technologies and modern API types drive the adoption of newer middleware- and protocol-types as demonstrated in Section 2.2.1 and Chapter 3. This evolution force is amplified by the drive to continuously address new requirements and constraints defined by the problem domains, implementation tools,

Name	Year	Transport	Serialization	Type
ONC RPC [4, 276]	1988	TCP, UDP	XDR [2]	RPC-based
CORBA	1991	TCP, IIOP	GIOP	RPC-based
SOAP	1998	HTTP	XML	RPC-based
REST [96]	2000	HTTP	JSON, XML	Resource-based
Apache Thrift [263]	2007	TCP, HTTP	Binary, JSON	RPC-based
gRPC	2010	HTTP/2	Protocol Buffers	RPC-based
WebSocket [189]	2011	HTTP & TCP	Binary Data	Message-based
GraphQL	2015	HTTP	JSON	Message-based
JSON API	2015	HTTP	JSON	Message-based

Table 1.1: Selective overview of web API types, the year of their first release, and a subset of transport protocols as well as serialization techniques that are typically used when implementing a web service using the web API type. The last column details the web service interface type the web API type can be classified into as detailed in Section 2.2.1.

and programming languages. As developing web services is often strongly coupled with the middleware- and protocol-types of web APIs, refactoring the source code requires time and effort. We define this subset of web service evolution as **web service interface evolution**.

> **Definition 6 – Web Service Interface Evolution:**
> Web service interface evolution encompasses the additions, removals, or evolution of web service API, middleware, or protocol types.

Chapter 3 further investigates web service evolution by developing a metamodel to reason about web service evolution. The chapter also describes related work, including existing tools to support binding to other web API implementations or providing adapters mapping one web API type to another.

Web service API evolution is defined as a subset of web service evolution, focusing on web service APIs as contracts between web service developers and web service clients.

> **Definition 7 – Web Service API Evolution:**
> Web service API evolution encompasses all additions, removals, or modifications to a web service API that can be categorized into breaking and non-breaking changes specific for each web API type.

Web service API evolution only affects a single web service API. Most web service API evolution use cases are manifested in additions, removals, or modifications to the web service API structure and can be divided into several change patterns [177, 266]. Some changes such as removals and a subset of modifications to a

web service API result in changes breaking the API contract between the web service client and web service, resulting in **breaking changes** [313, 280]. Chapter 4 further investigates related work, including web service API change patterns and categorizations, to showcase the challenges of web service API evolution for web service clients and web service developers.

One additional aspect of web service evolution is the change to the deployment structure of web services. While web services can be developed as monolithic applications, redesigning or partitioning them into smaller services enables a clearer decoupling of functionality and responsibility [198, 194]. Emerging deployment platforms such as Functions as a Service (FaaS) execution environments further push towards decomposing and partitioning web services into single-purpose stateless functions [198, 145]. Platforms as a Service (PaaS), Software as a Service (SaaS), or FaaS require web services to be packaged in deployable containers that can be distributed to a wide variety of web service hosting providers. In addition, software architectures based on edge and fog computing require dynamic execution-environment/software/protocol mappings that adhere to non-functional requirements and execution-environment-specific constraints [139]. Evolving web services to support new deployment structures, interoperability between hosting providers, and dynamically changing execution environments challenges existing web service development workflows. Chapter 5 further investigates tools and related work that address **web service deployment evolution** in the context of cloud-based deployments, web service observability, and the Web of Things (WoT).

Definition 8 – Web Service Deployment Evolution:
Web service deployment evolution encompasses dynamic and static changes to the deployment structure due to new requirements, constraints, and execution environment changes.

To summarize, web service evolution poses several challenges that affect three stakeholder groups: web service developers, web service hosting providers, and web service clients. Advancements in web technologies drive the adoption of new middleware- and protocol-types. Changes in the architectural styles and execution environments affect web service developers and hosting providers. Changes to web service interfaces and APIs impact clients consuming web services, relying on stable web service APIs. These stakeholder goals and their partially conflicting nature demonstrate the challenges related to web service evolution. These challenges motivate the research project to investigate web service evolution further and develop suitable treatments. Section 1.1 structures the design science project using research goals, design problems, and research questions. Section 1.2 details the research approach to address these research goals and answer the research questions.

1.1 Research Process and Research Goals

The dissertation structure is derived by combining the software engineering approach defined in ISO/IEC/IEEE 12207 [15] and design science methodologies defined by Wieringa [309]. Wieringa's design-cycle-based approach described in *Design Science Methodology for Information Systems and Software Engineering* decomposes research projects into three tasks: *Problem investigation* (Part II), *treatment design* (Part III), and *treatment validation* (Part IV) [309]. Software engineering workflows, artifacts, and methodologies defined by ISO/IEC/IEEE 12207 [15] are refined in software engineering textbooks by Bernd Bruegge and Allen H. Dutoit [62], Ian Sommerville [267], and Hans van Vliet [288] structure the treatment design, development, testing, and validation of the developed treatment artifacts. Figure 1.1 details the typical structure of a design cycle, including the questions answered in each of the parts of this dissertation.

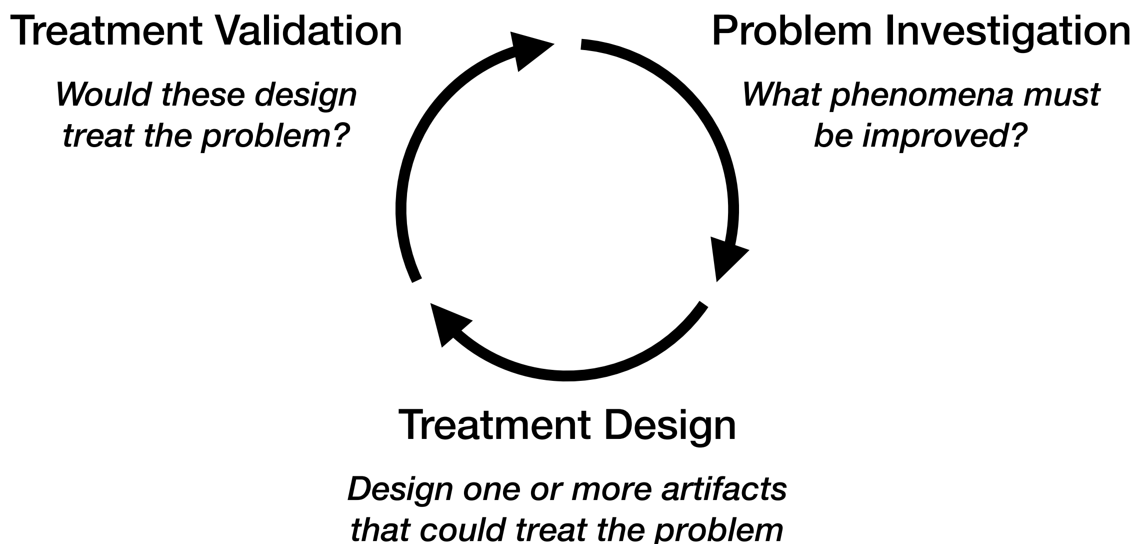


Figure 1.1: Design cycles are a subset of the engineering cycles presented in *Design Science Methodology for Information Systems and Software Engineering* by Wieringa [309]. Design cycles in design science research projects start with the problem investigation, leading up to the treatment design that is validated in the treatment validation before the design cycles potentially start again [309].

This section introduces the research goals of the dissertation based on the goal structure of design science research projects proposed by Wieringa shown in Figure 1.2. The following subsections present the hierarchical structure of stakeholder goals, technical research goals, knowledge goals, and instrument design goals.

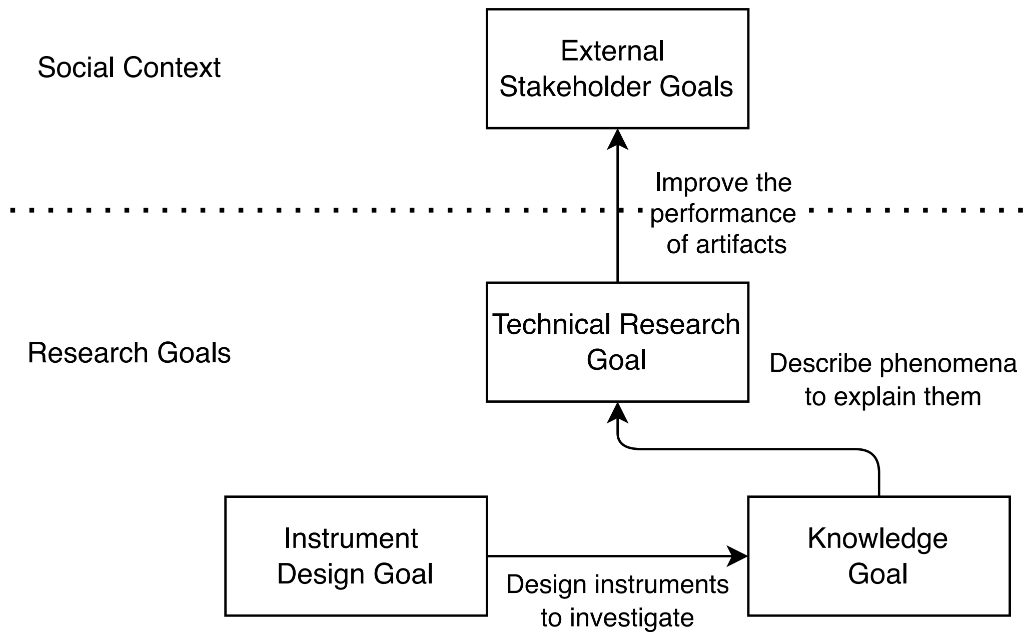


Figure 1.2: The goal structure and hierarchy according in design science projects according to Wieringa [309]: The external stakeholder goals are addressed by high-level technical research goals improving an artifact in a context. A knowledge goal further investigates the interaction of the artifact with its context, describing its internal mechanisms, while an instrument design goal describe goals to develop research instruments to provide insights to knowledge goal.

Stakeholder Goals

Wieringa notes that *"No goal exists in a normative vacuum, and the problem improvement goal in turn often supports some higher-level stakeholder goals"* [309].

> **Web Service Developer Stakeholder Goal:**
> Allow evolvability when developing and deploying web services.

Each web service stakeholder group is associated with a web service evolution-related stakeholder goal. A web service developer wants to evolve a web service, including its deployment structure, to deal with any change forces affecting the software system as defined in the web service developer stakeholder goal.

> **Web Service Hosting Provider Stakeholder Goal:**
> Enable an insightful and dynamic deployment of web service in a variety of deployment structures.

A web service hosting provider enables web service developers to deploy their web services to the hosting provider's platform. The hosting providers support different models and architectures to host a variety of deployment structures. A change in the web service deployment structure due to change forces impacting the

web service should be supported by the web service hosting provider to retain customers. It is essential to a web service developer to be able to evolve a web service and a deployment structure as detailed in the web service developer stakeholder goal. New additions to a web service are welcome as they increase the web service's functionality consumed by the web service client.

> **Web Service Client Stakeholder Goal:**
> Change and evolution stability when using web services.

While evolution is desirable for a web service developer to improve a web service, change affecting the web API often results in significant refactoring for the web service client. Therefore the goal of the web service client API stability while evolving the web service based on changes introduced by the web service developer.

Technical Research Goals

The overarching goal of design science projects is to improve the problem context by addressing the stakeholder goals with artifacts interacting with the problem context [309]. The problem context contains existing technologies to develop, evolve, deploy, and consume web services, the stakeholders interacting with web services, and existing business processes and techniques to enable web service evolution. Therefore, this dissertation aims to improve how stakeholders develop, deploy, and consume web services in the problem context of web service evolution. We define these goals as technical research goals to improve artifacts by enhancing their context-specific problem-solving performance [309].

> **Technical Research Goal 1:**
> Design artifacts supporting web service API type agnostic development to enable web service interface evolution.

> **Technical Research Goal 2:**
> Design artifacts enabling web service client compatibility while supporting web service API evolution.

> **Technical Research Goal 3:**
> Design artifacts supporting web service deployment evolution by supplying relevant context for generating static and dynamic deployment structures.

Technical Research Goal 1 addresses web service interface evolution by developing and improving artifacts that enable developing web services independent of web API types, including middleware- and protocol-types. Technical Research

Goal 2 addresses web service API evolution incorporating the stakeholder goals of stability for web service clients while also addressing the stakeholder goal of web service developers to enable developing evolvable web services. Technical Research Goal 3 addresses web service deployment evolution. Artifacts addressing web service deployment evolution need to have insights into deployment-related forces ranging from changes in requirements to changes in the deployment context as further investigated in Chapter 5.

Knowledge Goals

Based on the technical research goals, we establish knowledge goals with the aim *"to describe phenomena and to explain them"* [309]. Knowledge goals help validate the technical research goals and artifacts developed to fulfill these goals.

> **Knowledge Goal 1:**
> Identify generalizations and specializations of different web service interface, web service API, middleware, and protocol types.

Knowledge Goal 1 describes the goal to investigate different middleware- and protocol-types to benefit the Technical Research Goal 1. Identifying generalizations and specializations enables the design of artifacts to address web service evolution by developing web services independent of these aspects.

> **Knowledge Goal 2:**
> Identify web service interface type-independent change patterns to enable web service evolvability and change while providing stability to web service clients.

Knowledge Goal 2 addresses the conflicting stakeholder goals of web service developers and web service clients regarding evolvability and stability. Designing artifacts to tackle this discrepancy of goals requires an in-depth understanding of possible change patterns. Knowledge Goal 2 addresses the need to evaluate the artifacts developed as part of the Technical Research Goal 2.

Instrument Design Goals

Instrument design goals describe the intention to design research instruments to answer knowledge questions [309].

> **Instrument Design Goal 1:**
> Develop a metamodel to inspect generalizations and specializations of middleware, protocol, and deployment structures and processes.

Modeling is an expressive software engineering technique to inspect generalizations and specializations such as the different middleware- and protocol-types

described in Knowledge Goal 1. We use modeling to develop a research instrument to answer Knowledge Goal 1 using a metamodel-based approach. This metamodel allows us to investigate Knowledge Goal 1 and therefore shapes the artifacts envisioned in Technical Research Goal 1.

1.1.1 Design Problems

This section presents design problems, describing problems *"to (re)design an artifact so that it better contributes to the achievement of some goal"* [309]. Our design problems are presented based on a modified template by Wieringa, highlighting the problem context, artifact, requirements, and stakeholder goals [309]. The problem context is the design of web evolvable services. The artifacts in the design problems relate to tools, instruments, and workflows used to develop, deploy, and consume web services. As a result, the design problems are reduced to a combination of requirements and stakeholder goals, describing functional and non-functional requirements.

> **Design Problem 1:**
> Develop all aspects of web services in a web service interface type, web API, middleware, and protocol-independent description so that web service developers can support different web service interface types and web API types without rearchitecting web services.

The first design problem addresses the web service developer stakeholder goal to enable evolvability when developing web services. Design Problem 1 addresses web service interface evolution based on the Technical Research Goal 1. The design problem builds the foundation for addressing the challenges of web service evolution by describing the requirement of a web service description and development tool independent of the web service interface, web API, and other API-related implementation details. The web service description is one of the main challenges addressed by the artifacts designed in Part III. Answering Knowledge Goal 1 as well as the Instrument Design Goal 1 will provide a theoretical basis assisting in solving Design Problem 1.

> **Design Problem 2:**
> Automatically detect and migrate backward-incompatible changes of web service interfaces to enable web service client stability after modifications to the web service interface.

The second design problem addresses the need for backward-compatibility when web APIs evolve. This benefits the stability goal of web service clients while enabling evolvability for web service developers. The design problem describes the

need to automatically detect and migrate backward-incompatible changes to guarantee the web service client-perceived stability. Addressing web service deployment evolution requires providing and interpreting deployment- and application domain-related context and constraints.

> **Design Problem 3:**
> Develop a constraint-based service description so that web service developers can dynamically deploy web services incorporating different deployment structures, processes, and runtime constraints.

Design Problem 3 describes the corresponding design problem to Technical Research Goal 3. Investigating a constraint-based approach to generating a deployment structure enables the semi-automatic partitioning and deployment of web services according to a wide variety of deployment structures ranging from monolithic deployments to FaaS-based deployments. Requirements traceability and other annotations can be used as constraints to clarify possible deployment structures and partitionings of web services as detailed in Chapter 5.

1.1.2 Knowledge Questions

Knowledge questions are actionable refinements of knowledge goals, similarly to how design problems are actionable descriptions further defining and refining existing goals in design science projects [309]. The empirical knowledge questions build the foundation of the problem investigation in Part II and the validation found in Part IV. The knowledge questions are divided into three groups: Knowledge Questions 1 and 2 address web service interface evolution, Knowledge Questions 3 and 4 address web service API evolution, and Knowledge Questions 5 and 6 address web service deployment evolution.

Knowledge Goal 1 defines the goal to find generalizations and specializations of different middleware, protocol, and deployment structures and processes. The knowledge goal translates into Knowledge Question 1.

> **Knowledge Question 1:**
> What are the similarities, patterns, and differences of web service interface-, web service API-, middleware-, and protocol-types?

Chapter 3 picks up Knowledge Question 1 using the metamodel research instrument set out in Instrument Design Goal 1. Part III presents the Apodini ecosystem including the Apodini domain-specific language (DSL). Therefore Knowledge Question 2 investigates the applicability and extensibility as validated in Part IV.

> **Knowledge Question 2:**
> Does the Apodini DSL empower web service interface- and web API type-independent web service development?

Knowledge Question 3 refines Knowledge Goal 2 concerning web API evolution. As defined by Technical Research Goal 2 and Design Problem 2, automatically detecting and migrating backward-incompatible changes is a problem that should be addressed by artifacts designed as part of this research project. To design these artifacts, it is essential to answer Knowledge Question 3.

> **Knowledge Question 3:**
> What migration strategies can be used for different web API evolution change types?

Chapter 4 provides an overview of web service API evolution patterns including migration strategies addressing Knowledge Question 3. The different web service API evolution patterns form the requirements for artifacts designed in Part III to automatically or semi-automatically migrate web API changes. These artifacts include the creation of type-independent migration guides that are investigated in Part IV using Knowledge Question 4.

> **Knowledge Question 4:**
> How do web API type-independent migration guides translate to client-side web API-specific migration mechanisms?

Technical Research Goal 3 and Design Problem 3 concern web service deployment evolution. As noted in Technical Research Goal 3, tools to support web service deployment evolution need to retrieve context to create deployment structures. Therefore Knowledge Question 5 addresses the context collection further investigated in Chapter 5. The Apodini Deployer artifact designed in Part III including the Deployment Provider artifacts are validated in Part IV using Knowledge Question 6.

> **Knowledge Question 5:**
> How can artifacts collect requirements, constraints, and application domain and deployment environment-specific information to address web service deployment evolution?

> **Knowledge Question 6:**
> How do Deployment Providers enable deployment evolution in different deployment environments?

1.2 Dissertation Organization

Prelude

The prelude serves as an introduction to the topics discussed in this dissertation. Chapter 1 introduces the stakeholders, research process, research goals, design problems, and knowledge questions. Chapter 2 provides an overview of software engineering methodologies applied in this dissertation and foundational knowledge about distributed systems including web service interface types, web API types, and service description languages used to develop web services.

Problem Investigation

The problem investigation investigates the three aspects of web service evolution. Chapter 3 concerns web service interface evolution. The chapter provides an overview of different techniques ranging from service definition languages, adapter-based approaches, and mode-based approaches of providing abstractions spanning different web service interfaces and web API types. The web service interface metamodel (Figure 3.1, page 48) contributes to structuring the concepts shared across different web service interface and web API types and provides the foundation of the Apodini DSL. Section 3.2.2 presents examples for each web service interface type that conform to the metamodel. These conformances validate the applicability of the metamodel and answer Knowledge Goal 1 and Knowledge Question 1.

Chapter 4 investigates web service API evolution. The chapter provides a literature review of web API evolution challenges and resolution strategies, investigating Knowledge Goal 2. The chapter presents several related tools and techniques to identify changes in web APIs and classify these changes. Techniques to address API evolution are investigated by inspecting migration and protocol mechanisms that mitigate breaking API changes. The presented related work provides input to address Design Problem 2 when designing artifacts to address web API evolution. Section 4.2 introduces a web API change classification based on the web service interface metamodel introduced in Chapter 3. As demonstrated by the instantiated artifacts, the classification contributes to identifying, classifying, and migrating web API changes independent of the web service interface or web API type, addressing Knowledge Question 3.

Chapter 5 investigates web service deployment evolution-related challenges. Section 5.1 demonstrates several deployment-related challenges that web service developers and Deployment Providers need to consider when developing evolvable web services. The problem investigation investigates challenges in modern cloud-based FaaS-based deployments challenging traditional web service development. The increasing usage of Web of Things and the overall importance of developing

observable web services are investigated. The subsections in Section 5.1 all contribute to further clarify application areas for Design Problem 3. Section 5.2 describes annotation-based approaches to provide metadata to web services, providing additional context when developing and deploying web services, investigating Knowledge Question 5. These metadata annotation techniques build the foundation of the artifacts designed to address Design Problem 3.

Treatment Design

According to Wieringa *"treatment is the interaction between the artifact and the problem context"* [309]. The treatment design of the dissertation uses the knowledge context described in Chapter 2 and the Problem Investigation in Part II to design the Apodini ecosystem, addressing the technical research goals and design problems. As described in the Chapter 2, designing and specifying artifacts in the software engineering life cycle is done during system design, and object design. Section 1.1 already provides tangible research goals reflecting requirements set for the software systems developed in this dissertation. Therefore the treatment design part refines these requirements by containing a system design and object chapter, documenting the design decisions of the developed Apodini ecosystem using models and other software engineering artifacts.

Chapter 6 addresses the system structure and interaction of the individual artifacts. System design incorporates the definition of subsystems based on design goals and describes the control flow defining the interactions between the desired subsystems [62]. We define four subsystems that are discussed in detail: the Apodini DSL, the Interface Exporter subsystem, the Apodini Migrator subsystem, and the Apodini Deployer subsystem. These subsystems contain individual artifacts that address a part of the desired treatment described in this part of this dissertation.

Chapter 7 describes how the envisioned artifacts will be realized by specifying the structure and interaction of source code entities such as classes, structures, and algorithms [62]. The chapter presents the concrete instantiations of the artifacts, how they work internally, the challenges of developing these artifacts, and why certain development tradeoffs have been made. The object design introduces software libraries, internal and external domain-specific languages, and tools in the Apodini ecosystem, designed to specify and develop web services while addressing the design problems and research goals presented in Chapter 1.

Treatment Validation

The fourth part of the dissertation validates the artifacts presented in the previous sections: treatment validation. We present several single-case experiment mechanisms highlighting the artifacts in several environments and problem contexts.

Chapter 8 validates the web service interface and web API type-independent development capabilities of the Apodini ecosystem. Five Interface Exporter instantiations demonstrate the web service interface evolvability characteristics of the Apodini DSL and the Interface Exporter subsystem. The gRPC Interface Exporter demonstrates the applicability of the Apodini DSL and Interface Exporter mechanisms to RPC-based APIs and all communication patterns in an Interface Exporter. The WebSocket-, GraphQL-, and HTTP-based Interface Exporters detail the applicability of the Apodini DSL to message-based web service API types. The RESTful Interface Exporter demonstrates the applicability and extensibility of the Apodini ecosystem to resource-based APIs.

Chapter 9 presents five application domains in which we have developed web services demonstrating the different functionalities of the Apodini ecosystem and the applicability to these domains. Section 9.1 demonstrates Apodini's applicability to the sport and health science domain using the basketball player health monitoring system. The project is one of six projects in a project-based capstone course that used Apodini during semester-long projects and provided feedback about the Apodini DSL and related tools. Section 9.2 describes the event management platform that validates the automatic generation of a migration guide by the Apodini Migrator subsystem. Section 9.3 demonstrates a software system consisting of a mobile application and a web service to document financial transactions. The system demonstrates the functionality of the Apodini Deployer subsystem by automatically partitioning the web service into subprocesses using the Localhost Deployment Provider and FaaS cloud functions using the AWS Lambda Deployment Provider. The chapter also demonstrates two Internet of Things-based (IoT) projects using WoT web services to control and manage the IoT devices in fog-based architectures: Section 9.4 describes a smart city IoT system and Section 9.5 a water quality measurement system. The IoT Deployment Provider demonstrates the metadata annotation mechanisms enabling static configuration at deployment time and runtime reconfigurations based on changes in the deployment environment. The water quality measurement system also demonstrates the usage of the observability extensions of the Swift-based Apodini DSL.

Epilog

The epilog concludes this dissertation. Chapter 10 summarizes the research contributions and details possibilities for future work.

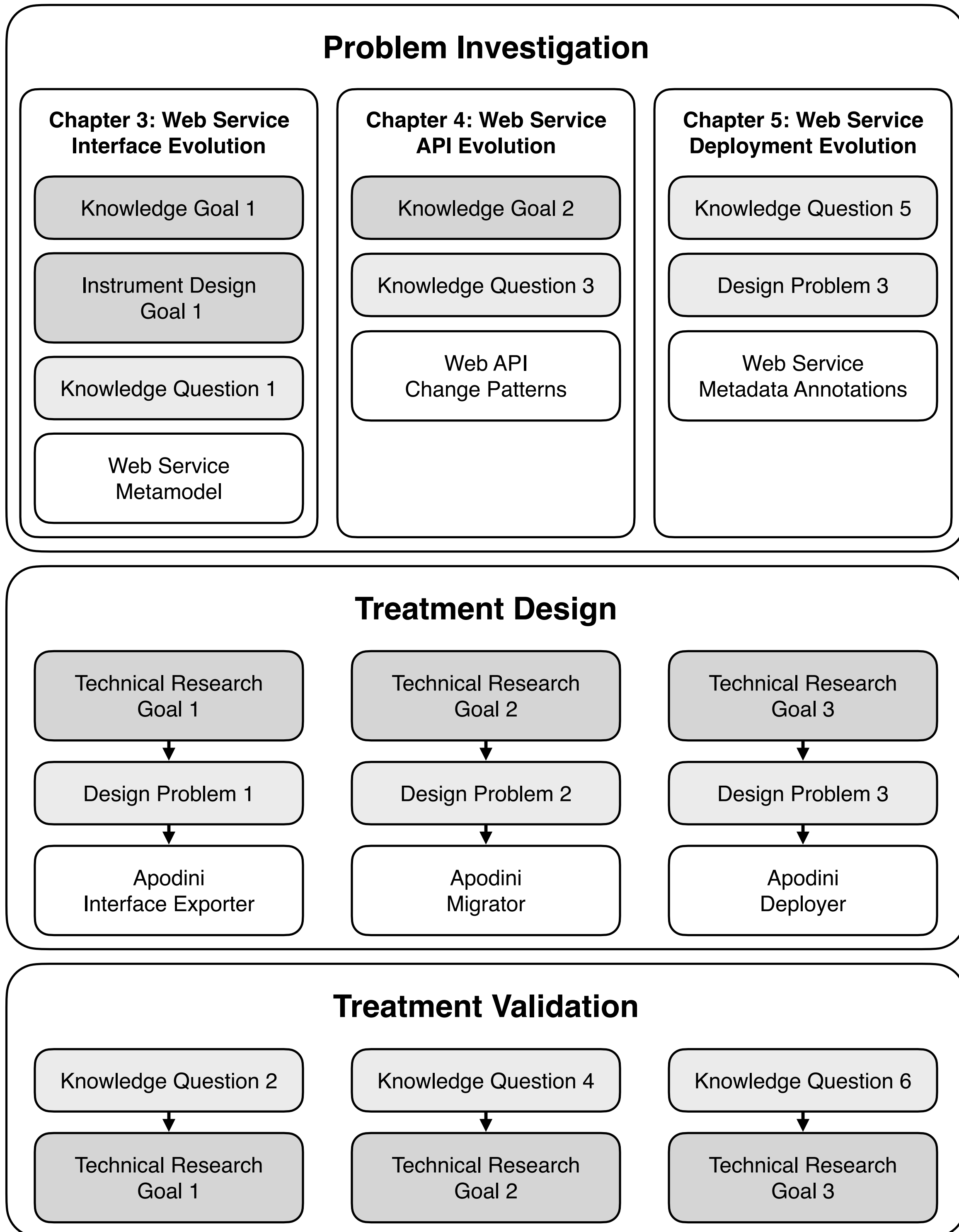


Figure 1.3: Visual representation of the dissertation structure. The **Problem Investigation** addresses the knowledge goals, instrument design goals, and knowledge questions by designing instruments and conceptual frameworks. The **Treatment Design** demonstrates the flow of technical research goals to design problems that are addressed by artifacts in the Apodini ecosystem. The **Treatment Validation** demonstrates the system and answers several knowledge questions investigating artifacts designed to address the technical research goals.

Chapter 2

Knowledge Context

In addition to the problem context and the social context, design science research is also embedded in a knowledge context [309]. Wieringa defines the knowledge context based on Vincenti's book *"What Engineers Know and How They Know It"* [291] as a collection of knowledge, ranging from common sense to scientific theories that are useful to the design science research [309]. This chapter summarizes the foundational knowledge from related research areas such as software engineering and software evolution, distributed systems, and domain-specific languages.

2.1 Software Engineering

Margaret Heafield Hamilton first coined the term *software engineering* as a necessity to create *"methods, standards, rules and tools for developing"* [265] software and embed into in the overall systems engineering process [68, 195]. The term grew in popularity and was first used as a conference title by the NATO (North Atlantic Treaty Organization) Software Engineering Conference chaired by Friedrich L. Bauer in 1968 [196]. The need for the discipline arose from the software crisis, resulting from computers becoming more powerful machines and incorporating complex hardware architectures that required abstractions, programming concepts, and software patterns [85]. Concepts like information hiding [213] and the benefits of object-oriented programming and other advancements in programming languages enabled reusable software architecture [256] and design patterns [109].

The ISO/IEC/IEEE 12207:2017 standard defines software engineering as the *"application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software"* [15]. Building complex software systems requires several activities that build up the processes of designing, implementing, and evolving software systems. The software life cycle processes and activities are grouped into four process groups (agreement processes, organizational project-enabling processes, technical management processes,

and technical processes), forming a *"set of interrelated or interacting activities that transforms inputs into outputs"* [15]. Software engineering mainly relates to the technical processes applied to transform the stakeholder goals into a software system, product, or service [15].

The *ISO/IEC/IEEE 12207 - Systems and software engineering – Software life cycle processes* [15] and *IEEE 1074 - IEEE Standard for Developing a Software Project Life Cycle Process* [7] standards form the foundation of several software engineering textbooks that elaborate on these activities, focusing on the technical processes, regrouping, and restructuring activities and processes. This dissertation refers to textbooks such as *Object Oriented Software Engineering Using UML, Patterns, and Java* by Bruegge and Dutoit [62], *Software Engineering: Principles and Practice* by van Vliet [288], and *Software Engineering* by Sommerville [267]. For instance, Bruegge and Dutoit define software engineering as a modeling, problem-solving, knowledge acquisition, and rationale-driven activity composed of several software engineering development activities [62]. Based on the ISO/IEC/IEEE 12207 technical processes, they define six software engineering development activities to deal with the complexity of developing a software system: requirements elicitation, analysis, system design, object design, implementation, and testing [62]. Similar to the ISO/IEC/IEEE 12207 technical processes, Sommerville describes software engineering as *"an engineering discipline that is concerned with all aspects of software production from initial conception to operation and maintenance"* [267]. Sommerville identifies common activities for a systematic approach to software engineering, the software process: software specification, software development, software validation, and software evolution [267]. It is important to note that these activities are independent of any life cycle model or sequence of processes in a life cycle model [15]. These activities are performed and organized based on the applied software life cycle model, project management activities, methodologies, and software process models used in software projects [62, 267].

Even though software maintenance and evolution were historically seen as separate software engineering activities, the *"distinction between development and maintenance is increasingly irrelevant"* [267]. According to Lehman, software evolution itself is not limited to a program statement level or a sequence of versions [172]. Software evolution considers the evolution of the processes and domains the software is embedded in, the software evolution process itself, and the models used to describe software evolution [172]. Agile software development practices embrace the importance of evolution, change, and feedback by defining empirical processes that determine short development cycles and feedback loops [156, 173, 254, 310]. The *"Stairway to Heaven"*[54, 203] model shows how continuous software engineering extends agile software development with the concepts of continuous integration and continuous deployment, leading up to the constant usage of feedback to experiment and test what the stakeholders need [54, 203]. Fitzgerald and Stol de-

fined *"Continuous *"* [97] as a set of activities in continuous software engineering that builds on top of the foundations of continuous improvement and innovation. This strive for constant evolution increases the importance of tools, infrastructure, and workflows that support these continuous activities [98].

In line with the focus on providing tools, infrastructure, and workflows to enable evolvable web services, we do not consider software maintenance or software evolution a separate software lifecycle process or activity, but consider evolvability a necessity for modern software systems. The following subsections provide an overview of different software engineering activities and foundational knowledge about software architectures.

Software Specification

According to Sommerville, requirements specification or requirements engineering includes requirements elicitation and analysis, requirements specification, and requirements validation [267]. Hull et al. define requirements engineering as *"the subset of systems engineering concerned with discovering, developing, tracing, analyzing, qualifying, communicating and managing requirements that define the system at successive levels of abstraction"* [134]. Requirements can be divided into two categories: functional requirements formalize the implementation-independent interactions of the system with its environment, non-functional requirements further specify how these interactions are performed [62]. Requirements can be part of several categories such as functionality, usability, reliability, performance, and supportability defined by the FURPS acronym originating from Hewlett-Packard (HP) [117, 118]. Software quality models as defined in ISO/IEC 25010 [14] and the Quamoco quality model presented by Wagner et al. [300] specify concepts and characteristics mainly focusing on non-functional requirements [299].

As software engineers need to understand the environment their system has to operate in, techniques such as modeling help software engineers to represent and understand systems [62]. During the analysis activity, developers transform the requirements and context from the application domain to form *"a model of the system that aims to be correct, complete, consistent, and unambiguous"* [62]. The software specifications created during requirements elicitation and analysis are subsequently used in further activities to develop a system to satisfy the stakeholder goals.

Software Development

Defining the design goals of a project, describing the boundary use cases, and constructing the software architecture is done during the system design software development activity [62]. System architects use requirements and the hardware architecture to formulate the software architecture, a design plan serving as a blueprint

during implementation [133]. A software architecture describes the system structure, including the composition of subsystems and software components and the interaction among these components, such as protocols for communication [256]. A software architectural style is a generalization, that defines element types and their interactions with no specific application domain [133].

Developers define subsystem interfaces, the services they provide, as well as the high-level behavior of a subsystem that are part of the software architecture during the system design phase [62]. System design includes the design of persistent data management, the mapping of subsystems to hardware, the choice of protocols used to communicate between the hardware nodes, and the design of the global control flow that impacts the interfaces of the subsystems [62]. During the object design phase, developers refine and extend the interfaces to include parameters and return types to construct the concrete Application Programmer Interface (API) and ultimately map the models and specifications created during all previous activities to code, forming the software system [62].

Software Verification & Validation

ISO/IEC/IEEE 12207 describes the verification process as a set of activities that *"provide objective evidence that a system or system element fulfils its specified requirements and characteristics."* [15]. The standard defines the validation process as a way *"to provide objective evidence that the system, when in use, fulfils its business or mission objectives and stakeholder requirements"* [15]. Software testing is a standard verification and validation activity to test software to reduce the risk of mistakes and verify and validate specified requirements and goals [12]. ISO/IEC/IEEE 29119-1 describes test processes and test sub-processes with different test levels or phases (e.g., component testing, integration testing, system testing, acceptance testing) and test types (e.g., performance, security, functional, and usability testing) [12]. Sommerville divides software testing into a three-stage testing process comprised of testing individual components using unit tests, a multistage system testing process that increasingly integrates components, and the final stage of testing performed by the customer with real data [267]. Bruegge and Dutoit define several testing activities, including test planning, usability testing, unit testing, integration testing, and system testing, which is sub-divided into functional testing, performance testing, and acceptance testing [62]. The instantiated artifacts presented in this dissertation use software testing techniques and activities to verify the stakeholder and research goals.

2.2 Distributed Systems

In 1960 Joseph Carl Robnett Licklider described the vision of a man-computer symbiosis, including a network of thinking centers, storing information similar to physical libraries [178]. Paul Baran pointed out the need for a standardized message block format and proposed to explore *"the possibilities of building a 'realtime' data transmission system using store-and-forward techniques"* [29] in the 1964 paper *On Distributed Communication Networks* [29].

The Advanced Research Projects Agency Network (ARPANET) and the National Physical Laboratory (NPL) network, which incorporated the concept of message switching, manifested their vision [79, 238]. Licklider and Taylor envisioned a geographically distributed computer network including *"message processors [...] [that are] interconnected to form a fast store-and-forward network"* [179]. ARPANET initially used the Network Control Program (NCP) to establish connections between hosts [70]. ARPANET showcased that such packet switching networks worked and could connect several distributed computer networks [238].

On January 1, 1983 the usage of NCP in ARPANET was eventually replaced by the Transmission Control Protocol (TCP) [226] and Internet Protocol (IP) [223] (TCP/IP) protocol stack [224]. This network stack builds the foundation of the Internet Protocol Suite that loosely incorporates five layers: the physical, data link, network, transport, and application layer [115]. The Internet Protocol Suite relies on a wide variety of mechanisms and protocols in the physical and data link layers to transmit bits and frames between devices [115]. The network layer provides a consistent network service that abstracts away characteristics of different links and physical layers that can exist on a network [142]. The network address is the crucial element that enables this functionality in the network layer [115]. The Internet Protocol, IPv4 [223] and nowadays increasingly IPv6 [82], provide IP addresses that are used to route traffic from a source to a destination [223]. The transport layer of the internet protocol suite is responsible for multiplexing multiple processes at a single network-layer address, such as using port addresses [115]. This layer handles connections, as well as flow and error control along the end-to-end connections [115]. The transport layer either uses the connection-oriented Transmission Control Protocol [71, 226] developed by Robert E. Kahn and colleagues to reliably deliver packets in an ordered manner or the User Datagram Protocol (UDP) [222] for a connectionless service that does not keep track of the order and delivery of packets [115]. The application layer is responsible for providing sessions to the application and converting network representations to representations in the application domain [115]. Services like Telnet [221], file transfer using the File Transfer Protocol (FTP) [220], and email using the Simple Mail Transfer Protocol (SMTP) [225] are implemented on the application layer, using TCP on the transport layer.

The switch from ARPANET to the TCP/IP protocol stack and the connection of multiple TCP/IP based networks, enabled autonomous systems connected by the Border Gateway Protocol (BGP) [181], lay the foundation of the modern internet. Based on these developments, Tim Berners-Lee proposed the idea of using a distributed hypertext system to manage information at the European Council for Nuclear Research (CERN) in 1989 [41]. Berners-Lee's concept eventually led to the development of the Hypertext Transfer Protocol (HTTP) and paved the way to the World-Wide Web (WWW), offering web services to clients [42]. While few creators mainly dominated the first years of the WWW, the Web 2.0 that emerged in the early 2000s, encourages user-generated content and builds on the idea of public APIs to extend, enrich, and consume existing web services [77]. These web services and their corresponding web APIs build the foundation of modern distributed systems. The Internet Protocol Suite and the Open Systems Interconnection (OSI) reference model (ISO/IEC-7498-1 standard [142]) both describe layered architectural styles that enable web service interconnection and interfaces [320]. In both models, the application layer is the topmost layer that defines the mapping of network communication mechanisms to application-specific representations [115].

As noted in Chapter 1, web services have traditionally been deployed on-premise, in the past decade, service models such as Software as a Service (SaaS) , Platform as a Service (PaaS), and Function as a Service (FaaS) do no longer require managing the underlying hardware infrastructure, by taking advantage of the benefits of cloud computing technologies like resource pooling [188, 209]. Virtual Machines (VMs), container technologies, and orchestration tools form the backbone of modern deployment strategies [209, 210]. Nowadays, cluster management architectures and new service models enable developers to abstract away hardware nodes when deploying a system [209]. Virtualization clusters include service discovery and orchestration mechanisms that allow developers to monitor, scale, and balance the load for different components of the systems at runtime [163].

With the growing role of IoT devices at the edge of the network, larger amounts of data produced by IoT systems, as well as a shift from data consumers to data producers, the bandwidth, computation power, and latency of cloud based systems are becoming a bottleneck of traditional client-service architectures [257]. Edge computing addresses these challenges by enabling computations in proximity to the data sources by performing these on smartphones, gateways, micro data centers, or cloudlets located at the edge of the network [257]. Going beyond the characteristics of edge computing, fog computing enables ubiquitous, scalable, dynamically reconfigurable, and context-aware access to smart and interconnected devices [139]. Heterogeneity of data acquired through multiple types of network communication capabilities has to be processed while requiring the interoperability of different services [139].

2.2.1 Web Service Interface Types

Web service interfaces are instantiated using web service APIs and provide a structured way to interact with web services. These web APIs behave like facades to facilitate the communication on the application layer between distributed components [78]. All web API types rely on passing packets between distributed instances but can be broadly divided into three main web service interface types: remote procedure call-based APIs, message-based APIs, and resource-based APIs [78]. The following section provides a historical overview of different web service interface types, detailing the evolution of web service interface types and web API technologies over time.

Remote Procedure Call APIs

Remote procedure call (RPC) APIs enable the transfer of parameters across address space boundaries, such as the execution on a remote machine, and passing results back to the original address space [45]. RPC APIs differentiate themselves from other web service interface types by reasoning about web services interfaces using procedures encapsulated in packets containing procedure arguments and procedure results [78]. Table 2.1 provides a historical overview of remote procedure call API types and their origins as detailed in this section.

Year	Development
1974	Procedure Call Protocol (PCP) [227]
1975	*Commentary on procedure calling as a network protocol* [244]
1976	*A High-Level Framework for Network-Based Resource Sharing* [307]
1981	Bruce Jay Nelson: *Remote Procedure Call* [197]
1984	*Implementing Remote Procedure Calls* [45]
1988	Sun RPC, later Open Network Computing (ONC) RPC [3, 4, 269, 276]
1989	Network File System (NFS) Protocol [5]
1991	Common Object Request Broker Architecture (CORBA)[1]
1996	Distributed Component Object Model (DCOM) [2]
2007	Apache Thrift [263]
2015	gRPC[3]

Table 2.1: Historic overview of RPC-based API types and their origins.

In 1974 the Stanford Research Institute (SRI), represented by Jon Postel and Jim White, published Request for Comments (RFC) 674 announcing the Procedure Call Protocol (PCP) documents. RFC 674 details first approaches to remote procedure

[1]CORBA version 1.0 can be found at https://www.omg.org/spec/CORBA/1.0/.

[2]Microsoft released a beta version of DCOM for Windows 95 in 1996: https://news.microsoft.com/1996/09/18/microsoft-releases-beta-version-of-dcom-for-windows-95/#Microsoft.

[3]Google open-sourced gRPC in 2015: https://developers.googleblog.com/2015/02/introducing-grpc-new-open-source-http2.html.

calls by referencing documentation on how to share resources and establish procedure calling interfaces instantiated for ARPANET [227]. The PCP documents sparked discussions about designing distributed systems and remote procedure calling as manifested in RFC 684, detailing several weaknesses of PCP and the underlying mechanisms [244]. Richard Schantz suggests that remote procedure calling is more related to inter-process communication (IPC) rather than local procedure calls, detailing weaknesses such as the lack of long-term concurrent executions and recovery in case of a component malfunction [244]. With the need to standardize remote procedure calls in ARPANET that included 75 host installations in 1976, RFC 707 detailes a *"network-wide protocol for invoking arbitrary named functions in a remote process"* [307] based on an inter-process communication facility [305]. The *High-Level Framework for Network-Based Resource Sharing* [305] provides an application-independent protocol providing programmers access to remote resources using a request-response protocol [305, 307]. James E. White of the Augmentation Research Center at the SRI further extends the presented Distributed Programming System (DPS) *"and standardizes other common forms of process interaction to provide a more powerful and comprehensive distributed programming system"* [306], laying the foundation for further research about remote procedure calls.

In 1981 Bruce Nelson's dissertation defined essential properties of remote procedure call mechanisms, building one of the first formal definitions of remote procedure calls: *"uniform call semantics, binding and configuration, strong typechecking, parameter functionality, and concurrency and exception control"* [197]. Birrell and Nelson further explain the structure of RPC mechanisms and implementing RPC services in their paper *Implementing Remote Procedure Calls* [45]. Their paper describes a typical implementation for modern RPC middleware- and protocol-type containing a client stub transmitting call packets. The process includes a way to identify the remote procedure and arguments that invokes the procedure implemented based on a service stub on the remote instance and returns a packet with the results back to the client stub that is mapped to a format the client can interpret [45].

As shown in Table 2.1, many different RPC middleware- and protocol-types have been developed and used to communicate in distributed systems and create web services. Sun's Remote Procedure Call messaging protocol documented in RFC 1050 and 1057 in 1988 is designed independent of the transport protocol and in its first version was implemented to be able to use both the TCP/IP and the UDP/IP protocol stacks [3, 4]. Sun RPC and the Sun Network Filesystem (NFS) also use RPC mechanisms to communicate with remote instances; both use the External Data Representation (XDR) to encode data and describe data formats across distributed and heterogeneous instances [2, 5]. The second version of Sun RPC, now named Open Network Computing (ONC) RPC, was released in 1995 and further developed until its current version in 2009 [269, 276]. Remote function calls, an object-oriented

approach of RPC, was instantiated with Java Remote Method Invocation (RMI) as *"a robust and effective way to build distributed applications in which all the participating programs are written in Java"* [120]. The Common Object Request Broker Architecture (CORBA)[4], and the Distributed Component Object Model (DCOM)[5] are further examples of RPC-based middleware- and protocol-types. CORBA offers discovery mechanisms and an Interface Definition Language (IDL), named CORBA IDL, used for defining the contractual interfaces of a CORBA component independent of a programming language [205].

After message-based and resource-based API types gained traction in the last 20 years, various modern RPC web API types such as Apache Thrift and gRPC revived RPC frameworks for usage in web services offered by large tech companies developing the RPC frameworks: gRPC and Apache Thrift [138, 263]. Apache Thrift was developed at Facebook, first described in a white paper in 2007, and a successor to the internal Pillar RPC framework developed at Caltech and Facebook [263]. Apache Thrift is a framework and code generation tool enabling developers to build RPC connections between distributed components [263]. Apache Thrift allows developers to express interfaces in an RPC-specific interface definition language, enabling flexibility in the choice of the underlying transport mechanism (e.g., HTTP, files on disk, or sockets) and transport representation (e.g., XML, ASCII text, or binary representations) [263]. Similar to Apache Thrift, Google open-sourced gRPC[6] in 2015 as a successor to the internally used Stubby RPC framework reaching version 1.0 in 2016[7]. gRPC is a cross-platform RPC framework that is part of the Cloud Native Computing Foundation (CNCF) using HTTP/2 as the wire transport protocol and Protocol Buffers as the interface definition language and high-performance, binary serialization format [138].

Message-Based APIs

Deriving a web service API from signatures of remote procedures introduces web API evolution-related challenges due to the resulting changes in the client code, requiring careful planning and additional collaboration between web service clients and web service developers [78]. Daigneau defines message-based APIs as interfaces that do not tie messages to specific procedures, but use self-descriptive messages sent to Uniform Resource Identifiers (URIs) where *"a web service deserializes and inspects the message, then selects an appropriate procedure (i.e., handler) to process the request"* [78]. The term message-based APIs is *"derived from the emphasis that is*

[4]The CORBA specification can be found at https://www.omg.org/spec/CORBA.
[5]The DCOM specification can be found at https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-dcom.
[6]The gRPC open-source project can be found at https://github.com/grpc/grpc.
[7]Blog post announcing gRPC version 1.0: https://cloud.google.com/blog/products/gcp/grpc-a-true-internet-scale-rpc-framework-is-now-1-and-ready-for-production-deployments.

placed on message design" [78] that the involved stakeholders agree on and are often expressed in various interface definition languages [78].

As noted in RFC 684-*Commentary on procedure calling as a network protocol*, RPC mechanisms have limitations that can be addressed with message-based APIs such as longer-running tasks or queued operations that can be better modeled with message-based APIs [244]. Message passing as a concept for inter-process communication on a local machine is a decades-old technique that message passing extends by routing messages in a network between distributed nodes [25]. A message in message-based APIs, as well as when using message passing, is a *"logical unit of information exchange between processes [and] queued as necessary in the sending node, in transit, and in the receiving node"* [25]. An example of a web service technology enabling message-based web service APIs is the WebSocket protocol, which web browsers provide as a built-in functionality offering a TCP message-based bidirectional communication with web services [189].

Another example is the Simple Object Access Protocol (SOAP). Even though SOAP defines an XML-based RPC mechanism and protocol (XML-RPC), its underlying mechanism conforms to the criteria defining message-based APIs according to Daigneau [78, 311]. Version 1.2 of the specification refers to SOAP as an *"an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols"* [191]. The SOAP design goals are simplicity and extensibility, focusing on the core message exchange using the Extensible Markup Language (XML) for serialization and supporting a wide variety of transport protocols including HTTP messages [191, 311]. Similar to RPC-based APIs, message-based APIs such as SOAP often use service descriptors instantiated using interface definition languages such as the Web Services Description Language (WSDL) and Web Application Description Language (WADL) to describe the web API and allow clients to generate client stubs to communicate with the web service [78]. The following subsection about domain-specific languages for web services provides a more detailed overview of mechanisms used for service descriptors and different instantiations using IDLs.

Created in 2012 and open-sourced in 2015, GraphQL is a message-based API middleware allowing clients to query and modify data using an execution engine hosted on a web service [93]. Message-based web APIs such as GraphQL rely on HTTP to express the functionality of the web API and transfer data over HTTP between web services and clients [219]. In accordance with the definition of a message-based API mentioned above, GraphQL APIs typically only offer a single HTTP endpoint, which dispatches requests to Handlers (most often GraphQL resolvers) that query or modify data to produce a response that is then aggregated and returned to the client [219].

Resource APIs

Most web services require application-domain-specific APIs that intreact with data using create, read, update, or delete (CRUD) operations [78]. Resource APIs map these concepts of the application domain to resources, standardized resource interactions mapped to URIs, resource representations, and HTTP methods expressing operations performed on these resources [78, 131]. Typical resource representations include XML and JSON as well as HTTP status codes to represent resources and the success or failure of an operation [78].

Defined by Roy Fielding in 2000, the Representational State Transfer (REST) architectural style constrains the exchange of resources, mostly implemented using HTTP messages, by offering no IDL and using Hypermedia As The Engine Of Application State (HATEOAS) to link resources [96]. The REST architectural style leverages application layer protocols such as HTTP as a foundation to transfer data and to define the semantic of a web API [78, 96]. Resource-based APIs as defined by Daigneau can adhere to the more strict REST constraints defined by Fielding, which are described in more detail in Part II: client-server, stateless, cache, uniform interface, layered system, and optionally code-on-demand [96].

Less strict HTTP- and JSON-based specifications, such as JSON-API[8], provide a further example of resource based web service interface types. Interface definition languages like OpenAPI[9] and the RESTful Service Description Language (RSDL) [239] allow developers to provide service descriptions of resource-based APIs and are further described in the following section.

2.2.2 Domain-Specific Languages for Web Services

General-purpose programming languages are Turing-complete computer languages with different abstraction levels used to implement procedures that can, e.g., be compiled or interpreted [293]. Program comprehension is a research field that investigates the understanding of source code by seeing code as an artifact that machines and programmers consume [60]. As von Mayrhauser and Vans state that *"Program understanding is a major factor in providing effective software maintenance and enabling successful evolution of computer systems"* [297]. In particular, adapting, perfecting, and correcting a software system requires a developer to have a mental model of the system, including the source code, the software architecture, requirements of the system, and suiting context in the problem domain [297]. Concepts such as declarative programming and domain-specific languages aim to increase program comprehension by providing a limited expressiveness by focusing on a domain that can be fluently and expressively described [107].

[8]The JSON-API specification is available at https://jsonapi.org/format/1.0.
[9]The OpenAPI specification is available at https://swagger.io/specification.

In contrast to general-purpose programming languages, Domain-Specific Languages (DSLs) target a specific problem area, focusing their syntax and semantics to that problem domain and hiding the complexity of the solution domain [111]. Martin Fowler defines a domain-specific language as *"a computer programming language of limited expressiveness focused on a particular domain"* [107]. In addition to programming, domain-specific languages can also be used to describe software from other viewpoints, such as Domain-Specific Modeling Languages (DSMLs) used in Model-Driven Development (MDD) [157]. External or standalone DSLs express an aspect of a software system using a custom syntax or language such as XML or JSON that is different from the primary programming language used to implement the software system [107]. Definition 9 defines externals DSLs for the scope of this dissertation based on definitions provided by Fowler [107] and Verna [290]:

Definition 9 – External Domain-Specific Language:
External domain-specific languages are autonomous languages that use a custom syntax to express domain-specific information.

In contrast to external DSLs, internal or embedded DSL are represented in general-purpose programming languages used to implement the software system and benefit from features found in integrated development environments such as code completion and compiler-level features such as type checking for strongly-typed languages [107, 290]. Definition 10 defines internal DSLs for the scope of this dissertation based on definitions provided by Fowler [107] and Verna [290]:

Definition 10 – Internal Domain-Specific Language:
Internal domain-specific languages stylize features of general-purpose programming languages to offer fluent domain-specific interfaces.

The increase in software complexity, in particular in distributed systems, has led to an increasing number of domain-specific languages [157]. While general-purpose programming languages are used mostly to express the functionality of web services, domain-specific languages are used in more specialized categories to build, describe, orchestrate, choreograph, and model web services [157]. The HyperText Markup Language (HTML) and Cascading Style Sheets (CSS) are examples of domain-specific languages to express the layout and style of websites, while the Structured Query Language (SQL) is a domain-specific language used to write database queries [107, 111]. The Web Services Description Language (WSDL) and other interface definition languages described in the following subsection are domain-specific languages used to specify, describe, and document web service APIs [192]. DSLs like the Web Services Business Process Execution Language (WS-BPEL), originating from the Web Services Flow Language (WSFL) [175], can build

on top of interface definition languages to describe processes involving multiple web services [19, 147]. Similarly, the Web Services Conversation Language (WSCL) builds on top of the WSDL to define the choreography of interactions between services as transitions and conversations [28]. As noted in Chapter 1, web service clients interact with web services using their interfaces. Therefore, interface definition languages are especially interesting when further discussing web service interface evolution in Chapter 3 as well as web service API evolution in Chapter 4.

Interface Defintion Languages

Daigneau notes that *"A Service Contract can be thought of as an agreement that specifies how clients and services may interact"* [78]. These contracts can be made implicitly by using predefined specifications, such as the HTTP specification providing a uniform interface for RESTful APIs, or explicitly by using specifications defined using interface definition languages [78]. Interface definition languages allow to specify the interface syntax of an API, excluding semantics, ordering, and other non-functional constraints [19, 272]. Using IDLs to describe web services also implicitly includes constraints resulting from the middleware- and protocol-types defined or expressible with the IDL that must be known to the web service client or tools to generate client stubs [19]. Properly specified interfaces that are complete and neutral enable interoperability and portability of web services but lack implementation-specific details about the inner workings of the web service that can differ between different implementations of the same web service interface [272].

RPC APIs use IDLs to express procedures, arguments, and return types that can be used to generate client and service stubs, e.g., Apache Thrift uses the Thrift IDL to express interfaces [263] and gRPC uses the Protocol Buffer IDL [138]. Similarly, middleware types like CORBA use the Object Management Group (OMG) IDL to specify the CORBA interface of services and generate client and code stubs that can be used as a starting point of an implementation, and abstract away the network and serialization logic [205].

The Web Services Description Language (WSDL), standardized by the World Wide Web Consortium (W3C), is a commonly used XML-based interface definition language with version 2.0 being released in 2007 [192]. WSDL uses operations to expose the functionality of a web service while *"an XML Schema defines the structure, the containers, and the types that are represented in the messages used for interacting with that service"* [36]. In contrast to most IDLs, WSDL does not rely on an implicit mapping to a middleware or protocol type but features an abstract definition of interfaces (port types in WSDL 1.1) consisting of different operations that exchange input, output, and faults using types (messages using types in WSDL 1.1) defined in the XML Schema [19, 75, 192]. These abstract interfaces and operations are instantiated using multiple transmission protocol bindings, while types are mapped to message format

bindings that specify the concrete service aspects of the abstract definitions [19, 192]. In the WSDL, IDL services describe the interface of a web service using endpoints (ports in WSDL 1.1) that map bindings associated with abstract interfaces to a network address to fully describe the web service [192]. Moving from WSDL 1.1 to WSDL 2.0 added support for all HTTP methods and reorganized and renamed several aspects of WSDL 1.1 to simplify the specification of web services while still being criticized for being overly complex because *"it is not meant to be directly created or read by humans"* [78] but to be used with specialized tools and IDEs [78]. In summary, the WSDL can be used as a traditional service description language and input to generate service and client stubs while keeping the non-functional concerns, service semantics, and implementation details out of the WSDL specifications [19].

The Web Application Description Language (WADL) [125], OpenAPI, and RESTful Service Description Language (RSDL) [239] focus on providing an IDL for RESTful and HTTP message based web services even though REST specifies no need for an IDL due to its uniform interface, e.g., using the HTTP protocol and links [78, 96]. The Web Application Description Language features XML-based textual documentations and specifications of HTTP-based web services, including the definition of resources, requests, and responses [125]. An OpenAPI document[10] features similar descriptions for RESTful APIs to document requests, responses, and exchanged data types to document, generate client or web service stubs, or test the web service API. The RESTful Service Description Language (RSDL) provides an XML schema for documenting RESTful APIs by taking *"a purist hypermedia-driven approach to REST design, requiring that a service have a single entry point, and focusing the design on resources, links, and media types"* [239].

[10]The OpenAPI specification can be found at https://swagger.io/specification/.

Part II
Problem Investigation

THE problem context of a design science project contains existing software, processes, methods, and other related artifacts [309]. Wieringa states that *"It is the interaction between the artifact and a problem context that contributes to solving a problem"* [309]. Therefore, it is essential to investigate and understand the problem context that contains existing software, methods, processes, and techniques used when dealing with the evolution of web services.

The problem investigation part is divided into three chapters.
Chapter 3 investigates web service interface evolution and the impacts on web service development, including the design of a web service UML metamodel.
Chapter 4 addresses web service API evolution by providing insights into the challenge, classifications, and current treatments of API evolution in general and web API evolution in specific.
Chapter 5 describes current challenges and approaches to deal with web service deployment evolution including metadata-based annotation models.

Chapter 3

Web Service Interface Evolution

Web service interface evolution is a subset of web service evolution. Web service interface evolution is concerned with the instantiation of the web service interface using web API types and the evolution of web service technologies during the lifetime of a web service:

> **Web Service Interface Evolution (Definition 6)**
> Web service interface evolution encompasses the additions, removals, or evolution of web service API, middleware, or protocol types.

Web service interface evolution is distinct from web service API evolution (Definition 7, page 7). In contrast to web service interface evolution, web service API evolution is concerned with how the evolution of the web service interface itself is manifested in different web API types and how this evolution affects web service clients consuming the service using a specific API type. These web API types can be grouped into three web service interface types: remote procedure call-based APIs, message-based APIs, and resource-based APIs [78]. Section 2.2.1 provides a detailed historical overview of web API, middleware, and protocol types from all three web service interface types.

Incorporating the wide range of web API types challenges system designers. Modern computing architectures such as cloud, edge, and fog computing often include a broad set of heterogeneous components with different interface requirements [188, 139]. Chen and Kazman introduce the classification of edge-dominant systems as *"one that depends crucially on the inputs and resources of its users for its success"* [72]. Edge-dominant systems, such as Wikipedia, Facebook, Twitter, or Youtube, can be described by the Metropolis structure and held together by a core software that provides services using different web APIs that are offered to developers, consumers, and producers [31]. The Metropolis structure has several implications on the software architecture, such as high modularity of the core and the requirement that core APIs are well-documented, and the description technology,

as well as the descriptions, must be kept up-to-date [31]. Supporting web API types for changing web technologies, in particular web protocol and middleware types, is essential to retain and gain developers writing software that creates an ecosystem around the core components.

Modifiability is the *"quality of a system describing how easily existing models can be modified"* [62] to incorporate changes in a software system, including changes to technologies, protocols, and standards [31]. Modifiability is enabled by dividing a software system into services that offer interfaces and strive for low coupling and high cohesion to replace and evolve individual subsystems easily [62]. Therefore, it is desirable to have low coupling between the functionality and the web service API implementation. Designing modifiable web services is therefore essential and a challenge as using a specific middleware or protocol on the application layer heavily influences the structure of components. In addition, correctly implementing middleware- and protocol-specific details of web API types requires expert knowledge. A lack of knowledge poses the challenge that interfaces might not conform to web API type specifications. Interfaces that do not conform to specifications require producers and consumers to carefully examine each interface individually to integrate it into their system.

A typical approach to creating software is model-driven development. Model-driven development allows developers to validate models in combination with other artifacts and generate data models, as well as web service- and client-stubs based on the models [260]. Model-driven development of web services can be performed using textural and graphical modeling notations or interface definition languages that capture web API-specific contextual information [272]. This approach is also referred to as design by contract, contract-first, or design-first development when the modeling of the web service API is performed using service description languages as detailed in Section 2.2.2 [78, 89, 132]. Interface definition languages and models mainly capture the syntax of web services while the semantics of the interfaces are manifested in the implementation of the web service [272]. Changes to the web technologies require the regeneration of source code stubs and result in a distribution of functional and non-functional concerns across the models, IDL definitions, and the application domain logic.

Alternatives to the contract-first approach of developing web services is the code-first approach. In the code-first approach, web service developers *"take the internal code implementing the given business logic and generate service descriptions from that business logic"* [260] that can then be distributed to web service clients. Some web API types do not require service descriptions, that rely on a uniform interface implemented in a code-first approach instead, allowing the discovery of functionality using a concept like hypermedia as the engine of application state (HATEOAS) in RESTful web services [96]. While contract-first approaches enable extensive control

over the structure and formatting of the exchanged messages, they require expert knowledge of the specific definition languages [78]. Contract-first approaches also introduce a high coupling between the service description artifacts and the source code, as they require the regeneration of source code stubs on the web service and client [78]. Code-first approaches benefit from fast prototyping times, not requiring maintaining separate IDL artifacts and generating source code stubs while forfeiting control over the concrete message structure [78].

Code-first or model-based approaches both introduce tradeoffs resulting in reduced maintainability in return for faster protocol, technology, or middleware-specific results. Using the metaphor of minimally invasive procedures[11], our research aims to introduce minimally invasive extension points addressing evolvability by combining the benefits of both approaches resulting in evolution-focused web service engineering artifacts. Designing this minimally invasive approach requires the investigation of challenges and solutions to address web service interface evolution in existing model and code-first methods. These approaches are manifested in different web service API-, middleware-, and protocol-types. Knowledge Question 1 formalizes this problem investigation based on Knowledge Goal 1.

> **Knowledge Goal 1:**
> Identify generalizations and specializations of different web service interface, web service API, middleware, and protocol types.

> **Knowledge Question 1:**
> What are the similarities, patterns, and differences of web service interface-, web service API-, middleware-, and protocol-types?

Section 3.1 highlights aspects of model-first and code-first approaches enabling web service interface evolution. Based on these insights from the problem context, conceptual frameworks *"can be used to frame a research problem, describe phenomena, and analyze their structure"* [309]. Section 3.2 describes such a conceptual framework around web service development by presenting a web service metamodel, building the foundation of the creation of new artifacts to develop and evolve web services as described in Part III addressing Instrument Design Goal 1.

> **Instrument Design Goal 1:**
> Develop a metamodel to inspect generalizations and specializations of middleware, protocol, and deployment structures and processes.

[11] *Minimally invasive medical procedures* describing surgeries that reduce the size of incisions and therefore reduce their risk.

3.1 Related Work

The related work section provides an overview of relevant web service interface evolution research and approaches. Section 3.1.1 describes different service description languages that can be used to express multiple web API types, enabling reuse across different web API types. Section 3.1.2 showcases adapter-based approaches to enable web service interface evolution by exposing different web API types using adapters to convert the interface of a legacy web API type. Section 3.1.3 collects several modeling-based approaches for web service interface evolution, such as metamodels for web service interfaces as well as model-based web service development approaches.

3.1.1 Service Definition Languages

Chapter 2, and especially Section 2.2.2 about web service description languages, provides an overview of different domain-specific languages that can be used to model web service interfaces. While most interface definition languages are specific to a single API type, some modeling and definition languages allow modeling web services for different web API types.

A prominent example already shortly described in Section 2.2.2 is the Web Service Description Language (WSDL). The latest version of the specification from 2007, WSDL 2.0, defines that *"WSDL 2.0 describes a Web service in two fundamental stages: one abstract and one concrete"* [192] while focusing on message-based APIs: *"At an abstract level, WSDL 2.0 describes a Web service in terms of the messages it sends and receives; messages are described independently of a specific wire format using a type system, typically XML Schema."* [192]. WSDL interfaces contain operations that are defined by a combination of message exchange patterns and messages [192]. This abstraction layer provides the WSDL the possibility to describe different message-based web service interfaces while bindings describe the *"concrete message format and transmission protocol which may be used to define an endpoint"* [192]. Part 2 of the specification *Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts* [204] defines concrete bindings to SOAP 1.2 and HTTP.

In addition to bindings, WDSL offers predefined message exchange patterns and operation styles. Operation styles define additional information and constraints about operations, especially on the messages and faults that can be exchanged with the web service [192, 204]. The *Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts* defines several operation styles such as a remote procedure call (RPC) style, Internationalized Resource Identifier (IRI) style, and a multipart style [204]. Operation styles such as the RPC style provide constraints for the messages that are associated with an operation as well as message exchange patterns, e.g., the *in-only* and *in-out* message exchange patterns for the RPC style [204].

While WSDL 2.0 is primarily focused on message-based web API types, operation styles such as the RPC style and possible extension points using bindings also allow the creation of RPC-based APIs. While a WSDL document can provide an abstract overview of the interfaces offered, its proprietary XML-based syntax differs from IDL definitions from modern RPC standards described in Section 2.2.1. Supporting these interface types would require an automatic or manual translation of WSDL documents to, e.g., Protocol Buffer definitions for gRPC based interfaces.

The HTTP binding builds the foundation to model resource-based web APIs using WSDL 2.0. While API types such as the RESTful API style and its uniform interface requirement build on the expectation of dynamic discovery of resources using HATEOAS, WSDL documents can be used to generate API interfaces similar but not out of the box conformant to RESTful APIs. In addition to the HTTP binding, they require additional logic in the service-side stubs or annotations in the XML description to more closely fulfill the uniform interface requirements of RESTful APIs such as HATEOAS most often achieved using a code-first approach.

In their paper *A Metamodel for the Web Services Standards*, Simon et al. describe a metamodel for service-oriented architectures (SOAs) (Section 3.1.3) as well as the domain-specific SOA language (SOAL) for describing web services [259]. SOAL offers a top-down development approach based on a compact domain-specific language to describe platform-independent web services that can be used to *"generate WSDL, program code and configuration files for the various SOA products"* [259]. As the SOAL language includes a limited subset of WSDL options constrained by their metamodel, that enables a compact representation of web service interfaces that are translated into more verbose service definition specifications [259].

Rademacher et al. present an external DSL similar to the WSDL for describing web services in a technology-independent description supporting the SOAP protocol and RESTful web APIs [231]. The DSL introduces several generalizations that are expressed in the service specification language, including request, response, and basic types to formulate the content of a request and response [231]. A code generation component parses the DSL and generates boilerplate code that is subsequently filled with application domain logic provided by the web service developer [231].

Based on the investigation of the service definition languages for web service interface evolution, we conclude that the overall approach of abstracting web API types into a common interface is a promising approach. At the same time, the complexity of these languages and the disconnect between source code and specifications needs to be addressed. The most prominent example, the WSDL, enables a contract-first development workflow enabling web service interface evolution by defining different bindings and operation styles and therefore offers essential insights and learnings for the research conducted in this dissertation. The complexity of the WSDL's XML-based syntax eventually led to a decrease in usage and

ultimately to no further development of the WSDL standard beyond version 2.0. Advancements in web technologies resulted in the usage of more specialized and less verbose service specifications such as OpenAPI for HTTP message-based APIs. WSDL documents and code skeletons created from languages like SOAL can subsequently be used to generate client and web service stubs filled with application domain-specific logic not expressible in the external domain-specific language. The DSL developed by Rademacher et al. provides a promising approach for further abstraction using an external DSL separate from the WSDL.

3.1.2 Adapters

The adapter pattern for object-oriented software as defined by Gamma et al. is used to *"Convert the interface of a class into another interface clients expect"* [109]. While the adapter pattern, according to Gamma et al. is defined for object-oriented programming, according to Buschmann et al., it *"is too broad to be considered a single pattern at this level: it captures the general essence of a family of patterns concerned with adaptation"* [67]. This section demonstrates the decades-old research field that applies adapter patterns to address web service interface evolution.

Aversano et al. demonstrate how evolving services with no web API, such as moving legacy systems written in COBOL to web-enabled services, involves creating adapters wrapping the existing functionality into components to expose the functionality as a web service, including web pages [26]. Bridging technologies, as presented by Futoonhi et al., addressed the interoperability between different services by bridging the communication between different middleware- and protocol-types such as Distributed Computing Environment (DCE), CORBA, and DCOM described in Section 2.2.1 [95].

As described in Section 3.1.1, WSDL and SOAP-based web services have been evolving to support more or less RESTful APIs by taking advantage of the uniform interfaces defined by RESTful APIs relying on HTTP as the transport protocol. Protocol adapters such as the SOAP to RESTful HTTP mapping (StoRHm) developed by Kennedy et al. enable a client-side mapping between existing SOAP-based web services and already defined RESTful web APIs while transmitting the same information as the SOAP-based instance [154]. Using protocol adapters such as StoRHm allows gradual client-side transitions from one web API type to the other by hiding the mapping behind the adapter while taking advantage of the benefits of the newly adopted or in-parallel used web API type [154]. StoRHm relies on a semantic layer, similar to how domain-specific languages can use semantic models that SOAP descriptions are transformed into, and RESTful schemas are generated out of [153]. Due to the SOAP-specific information provided in WSDL schemas, this approach offers several limitations, including that HTTP PUT and POST requests rely on unaltered XML-based SOAP messages that have to be understood

by a RESTful web service [153]. Even though service-side adapters enabling web service interface evolution are mentioned in future work, all published versions of StoRHm solely apply to the client-side, requiring the existence of a RESTful web service [153]. In contrast to StoRHm, Upadhyaya et al. present a sophisticated web service-based adapter and migration approach to transforming SOAP-based web services to offer RESTful web APIs [285]. The paper *Migration of SOAP-based Services to RESTful Services* presents a multi-step process of identifying resources and HTTP methods based on WSDL definitions by clustering operations and generating wrappers and configurations to SOAP messages to RESTful HTTP messages [285]. Research by Strauch and Schreier showcases RESTify, a procedure model to transform RPC-inspired web APIs using WSDL documents to HTTP-based RESTful web APIs [271]. RESTify is based on a three-step process: first, split the service into district resources, apply a uniform interface, and therefore bridging the gap between operations and resources and in a third step, refining hypermedia as the engine of application state (HATEOAS) [271]. The focus on HATEOAS is crucial to fulfilling the constraints defined by the RESTful architectural style and especially important as it is most often missed in other adapter-based tools such as StoRHm [154] and the adapter presented by Upadhyaya et al. [285]. When transforming abstract service interfaces to RESTful interfaces, the artifacts designed as part of this dissertation need to assure that information required to support HATEOAS is inferred from the service interface and can be refined by the web service developer.

As described in Section 2.2.1 GraphQL is a modern message-based API type starting to gain traction in replacing RESTful web APIs by reducing response sizes compared to RESTful APIs [58]. Brito et al. demonstrate the reduction in response size in a practical assessment that creates a GraphQL-based wrapper adapter around a RESTful API [58]. Similarly, Wittern et al. present the reusable OASGraph tool that generates GraphQL-based adapters for existing RESTful APIs described by OpenAPI documents [312]. OASGraph contains a multi-step process that translates types in an OpenAPI specification to GraphQL types, removes duplicates, provides a translation between OpenAPI types and GraphQL types, creates resolver functions, and gathers nested data using the HATEOAS links information stored in OpenAPI specifications [312]. Similar to OASGraph, the Swagger-to-GraphQL[12] and GraphQL Mesh[13] open-source projects allow to create a GraphQL wrapper based on other web service description artifacts of legacy web services. Hernandez-Mendez et al. present a model-based approach to generate RESTful web services that query and compose several web services, including GraphQL-based web services, blurring the line between an adapter-based approach and model-based approaches as discussed in Section 3.1.3 [130].

[12]Swagger-to-GraphQL can be found at https://github.com/yarax/swagger-to-graphql.
[13]GraphQL Mesh can be found at https://github.com/Urigo/graphql-mesh.

3.1.3 Model-Based Approaches

As described in more detail in Section 3.2, UML can be extended using profiles and stereotypes that enable *"the use of specific terminology or notation"* [13]. *Toward UML Profiles for Web Services and their Extra-Functional Properties* by Ortiz and Hernandez presents a metamodel to express non-functional concerns in middleware and protocol-independent ways [208]. Integrated into the WS-* ecosystem (described in more detail in Section 5.2.2), the introduced UML profile and UML stereotypes can be used to generate WS-Policy standard-based documents [208]. Similar to Ortiz and Hernandez, Wada et al. also present a model-driven approach to express non-functional concerns when developing web services, focusing on a model-driven development (MDD) framework [298]. The presented UML profile contains several stereotypes *"to specify service-oriented applications: service, message exchange, message, connector and filter"* [298]. The UML profile can be used to specify and evolve services that are transformed into source code using a presented MDD tool, improving the reusability and maintainability of web services by abstracting implementation-related details in the UML models [298].

Kchaou et al. define *WS-UML: a UML profile for web service applications*, addressing web service specific aspects currently not specified in UML: security, composition, quality of service, the community of service and functionality, as well as execution traces [151]. In addition, Kchaou et al. present a UML metamodel, describing web services based on the abstract level of the WSDL and therefore independent of the web API type, enhanced with the five perspectives derived from the UML profiles which are described in the publication [151]. The UML profiles, as well as the UML metamodel, can be used to increase the expressiveness of UML models, allowing developers to map concepts from the application domain and solution domain to concepts to WSDL based web services. According to Kchaou et al. this increases the comprehension of WS-* based concepts while explicitly providing modeling tools for the mentioned five perspectives [151].

Elyacoubi et al. present a UML metamodel combining aspects of the WSDL and an extension defined by the Semantic Annotations for WSDL and XML Schema (SAWSDL) standard [88]. SAWSDL enables semantic annotations to several parts of WSDL specifications, enabling the mapping of XML schema types to semantic models such as modeling established ontologies [94]. The metamodel allows web service developers to add organization-adopted semantic model annotations to UML-based web service models, enabling web service interface evolution across different encoding representations [88].

Standardized by the Object Management Group (OMG), *"The Service oriented architecture Modeling Language (SoaML) specification provides a metamodel and a UML profile for the specification and design of services"* [11]. SoaML enables developers to specify web services, including their functional and non-functional concerns, using

extensions of the UML [11]. This enables SoaML users to focus on service modeling aspects. The metamodel contains the core concept of services that are offered using UML ports using the service stereotype [11]. Stakeholders can use services offered using ports using the request stereotype [11]. SoaML embraces a model-driven architecture approach that enables web service interface evolution using different technology profiles. SoaML enables simpler interfaces that embrace a one-way interaction similar to PRC-based interfaces, as well as service interfaces for bidirectional services that include callbacks or sophisticated choreographies [11].

Simon et al. present a metamodel incorporating aspects of the WS-* standards including security, reliability, and transactions, allowing web service developers to model platform-independent web services [259]. The SOA metamodel (SoaMM) by Simon et al. provides UML extensions in the form of stereotypes enabling a top-down development approach for web services using the WS-* protocols [259]. The metamodel builds the foundation for the SOA Language (SOAL) described in Section 3.1.1 and can be used to model web services in a platform-independent way by describing interface and middleware related aspects [259]. The expressiveness of the metamodel is limited to a subset of possibilities in the WSDL and WS-* extensions and, therefore, similar to the other WSDL-based metamodels.

Fokaefs and Stroulia present *WSMeta: A Meta-Model for Web Services to Compare Service Interfaces*, identifying a common ground between the WSDL which mainly focuses on SOAP based web API types and the XML-based Web Application Description Language (WADL) which focuses on describing HTTP based web APIs such as RESTful APIs [103]. The metamodel offers a generalization of the WSDL- and WADL-specification that allows the transformation of one specification to the other, enabling web service interface evolution for a subset of web service specification languages that can be abstracted using WSMeta [103].

Garriga and Flores present a metamodel for heterogeneous web services driven by standards such as SoaML, WSDL 2.0, and WADL and verify its extensibility by applying it to HTTP-based IDLs such as OpenAPI and RAML [110]. The goal of the metamodel is to express meaningful information independent of the underlying technology to enable the comparability of web services as well as building adapters as described in Section 3.1.2 [110]. The publication provides a mapping between the metamodel and SoaML, WSDL 2.0, and WADL by introducing abstractions applicable to all standards as well as specialized converters to translate the metamodel to OpenAPI and RAML specifications [110]. The metamodel presented by Garriga and Flores provides valuable insights into abstractions needed to support multiple different web service interface types which are applied to the metamodel presented in Section 3.2.

3.2 Web Service Interface Metamodel

This section presents a conceptual framework for web service interface evolution using a web service interface metamodel. Following the design science methodology by Wieringa, the conceptual framework enables us to describe the phenomena of web service interface evolution and provides generalizations that help express the requirements for artifacts [309].

The presented metamodel (Figure 3.1) is a UML version 2.5-based metamodel that uses a UML profile. The UML specification describes the UML syntax by defining the UML metamodel using metaclasses that are described using a subset of UML [13, 16]. The UML extension syntax is defined in the Meta Object Facility (MOF) Core Specification, showcasing the four-layered metamodel architecture consisting of MOF, UML, the user model, and the user object [16].

UML and its metamodel can be extended using profiles [13]. The UML specification states that *"the intention of Profiles is to give a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular domain, platform, or method"* [13]. A common way to extend UML using profiles is stereotypes. As described in the UML specification, *"A Stereotype defines an extension for one or more metaclasses, and enables the use of specific terminology or notation in place of, or in addition to, the ones used for the extended metaclasses."* [13]. A stereotype can have properties and can participate in binary associations [13].

Metamodels are not exclusive to the UML but can also be expressed in several other formalisms. Kleppe and Rensink present a graph-based semantic of UML diagrams, presenting type graphs as an equivalent to UML class diagrams and instance graphs as an equivalent to UML object diagrams [236]. Using these semantics, Kleppe defines that *"A metamodel is a model used to specify a language"* [157], allowing us to define domain-specific languages based on metamodels, using abstract syntax models (ASMs), concrete syntax models (CSMs), and semantic domain models (SDMs) [157].

The metamodel shown in Section 3.2.1 enables web service interface evolution by providing a generalization of several web service interface types and web API types. The metamodel differs from the related work listed in Section 3.1.3 by encompassing all web service interface types described in section Section 2.2.1 and being independent of any existing web service description language. The conformance of different web service interface types and their instantiations in concrete web API types to the metamodel for a resource-based (REST, Figure 3.2), a message-based (GraphQL, Figure 3.3), and a RPC-based (gRPC, Figure 3.4) is described in Section 3.2.2.

3.2.1 Web Service Interface Metamodel

The domain presented in the metamodel is web service interface types. The metamodel presented in Figure 3.1 is a UML based metamodel using a UML profile. The stereotypes extend the class metatype with several web service interface concepts that can be applied instances of web service interfaces: web API types. To enable web service interface evolution, we define several generalizations to compare different web API types. These generalizations enable web service interface evolution by designing artifacts to transform web service API types into each other or design artifacts that allow web service development at an abstract level independent of the web service API types. The metamodel addresses Instrument Design Goal 1.

> **Instrument Design Goal 1:**
> Develop a metamodel to inspect generalizations and specializations of middleware, protocol, and deployment structures and processes.

A web service consists of several **Handlers**, handling requests arriving at a web service. The concept of a Handler is derived from the web service API concepts described by Daigneau, describing web services as programs that deserialize messages, inspect them and then select the appropriate Handler (e.g., a remote procedure, message handler, or resource) to handle the request and possibly produce a response [78]. Handlers are identified by an identifier that is used to multiplex incoming requests to the corresponding Handler. Requests as well as corresponding responses contain serializations of application domain entities that are defined by the concrete web API type.

Communication Pattern	Association Request – Response
Request-response	1...1
Client-side stream	*...1
Service-side stream	1...*
Bidirectional stream	*

Table 3.1: Overview of possible communication patterns of a Handler in web service interface metamodel presented in Figure 3.1. A Handler can have four different communication patterns: request-response, client-side stream, service-side stream, and bidirectional stream.

The communication pattern defines the multiplicity of the association between Handlers and requests, as well as requests and a responses. The multiplicities of these associations are not explicitly modeled in the web service interface metamodel

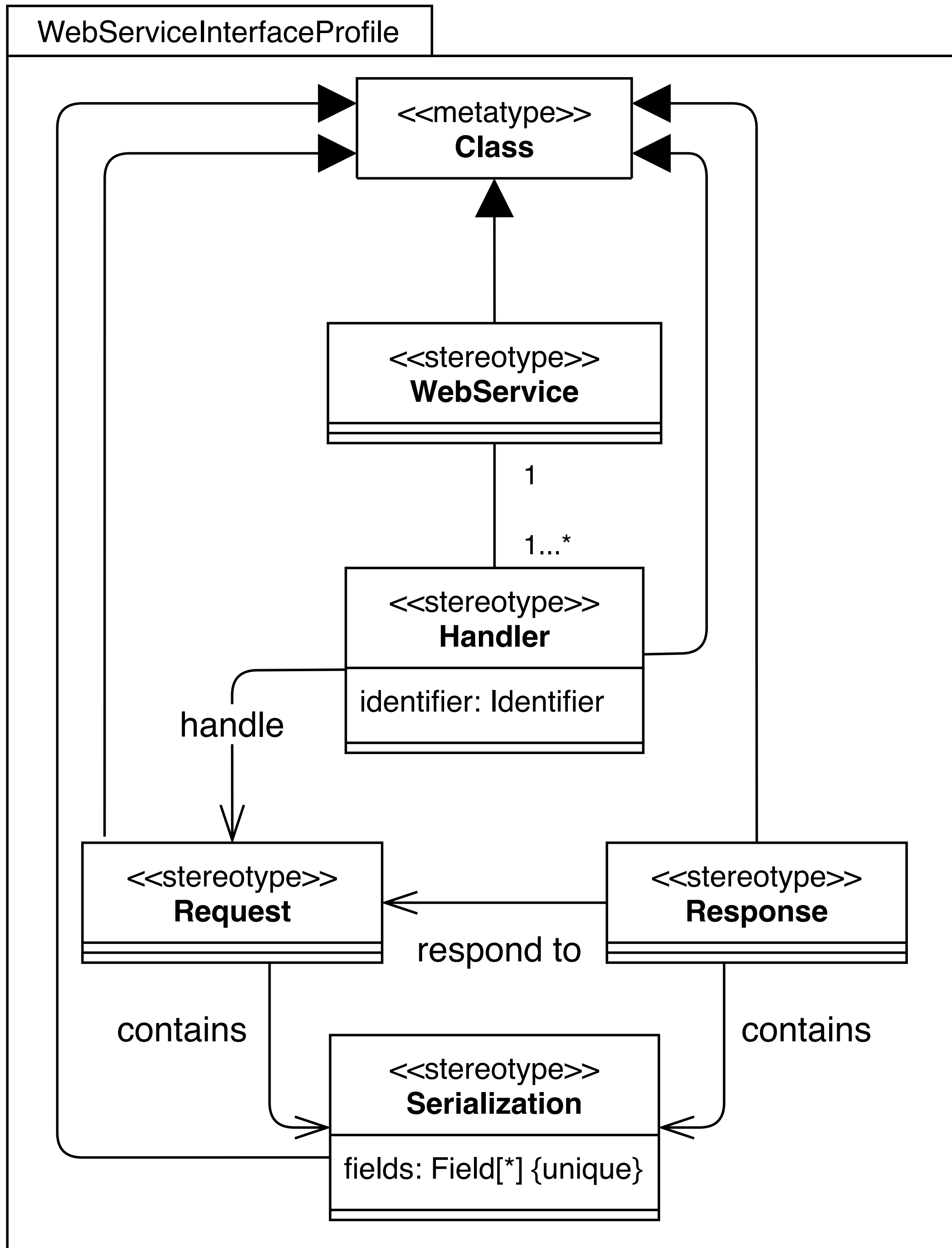


Figure 3.1: The web service interface type metamodel defines several UML stereotypes that can be used to extend UML class diagrams representing web API types. These stereotypes include the web service stereotype that is associated with several Handlers, handling requests and returning responses. A serialization contains several unique fields carrying information in a serialized format. (UML Metamodel Profile)

as they are constrained on a web API level and defined on a Handler level in the concrete web API. The constraint of possible communication patterns is demonstrated by the multiplicities introduced between a Handler, request and response in Figure 3.2, Figure 3.3, and Figure 3.4. We distinguish between four different communication patterns as described in Table 3.1.

A request-response pattern accepts a single request and executes the functionality of the Handler, producing a single response returned to the client application. The client-side stream pattern assumes that Handlers receive several requests originating from the client application triggering functionality in the Handlers. After the client indicates the end of the stream or the Handler terminates the stream, it eventually returns a single response to the client application. A service-side stream allows a client application to send a single request to a Handler that returns a stream of responses to a client application. The bidirectional stream pattern enables an open connection between a client application and a web service. It allows an arbitrary number of requests and responses to be exchanged between the distributed instances before one instance terminates the bidirectional stream.

3.2.2 Metamodel Conformant Web API Types

The metamodel defined in Figure 3.1 provides several stereotypes describing web service interfaces that can be applied to classes modeling concrete web API types. This section demonstrates the applicability of the metamodel to different web service interfaces and web API types to validate the metamodel before we design artifacts in Part III based on the insights gathered from the metamodel. The metamodel and the web API types that conform to the metamodel provide answers to Knowledge Question 1.

> **Knowledge Question 1:**
> What are the similarities, patterns, and differences of web service interface-, web service API-, middleware-, and protocol-types?

RESTful Web API Model

The first web API type we investigate is a resource-based web service interface: the RESTful API type. Several different REST-specific UML metamodels and profiles exist to allow modeling RESTful APIs using the UML [206, 243, 252, 253]. In contrast to the web service interface evolution-focused models presented in Section 3.1.3, the research presented in this section focuses exclusively on the RESTful web API type.

Schreier presents a metamodel divided into a structural and behavioral model enabling web service developers to model RESTful web services based on the Eclipse Modeling Framework (EMF) and the Essential MOF (EMOF) specification [253].

The metamodel contains core REST concepts such as resource types and identifiers, links connecting resources according to the HATEOAS requirement, method types that can be mapped to HTTP methods, parameters, and media types to describe serializations [253]. Similarly, Ormeño et al. present a small UML profile defining several stereotypes to extend application domain models with elements from the metamodel such as rest-controller and rest-service [206].

Schreibmann and Braun introduce a UML-based metamodel as well as a domain-specific language, the REST Domain Specific Language (RDSL), as ways to model RESTful web services [252]. The metamodel contains REST-specific concepts such as resources, media types, concepts to query resources, as well as the possibility to express caching and pagination mechanisms [252].

Rossi presents a UML profile-based approach to model-driven development of RESTful APIs, addressing the wide variety of service description languages that can be used to model RESTful web services such as JSON Schema, WADL, OpenAPI, and RAML [243]. The metamodel provides a language-agnostic solution describing the structure of a RESTful API and enables the generation of code skeletons and API documentation without the need to introduce a new service description language [243]. The transformation to source code and API documentation is realized using a UML-to-RAML transformation achieved by a UML-based REST-specific profile that allows web service developers to model RESTful APIs using resource, resource path, and HTTP method stereotypes [243].

Figure 3.2 presents a model for RESTful web APIs conforming to the metamodel presented in Figure 3.1 using the terminology defined by Fielding [96]. A RESTful web service marked with the web service stereotype is composed with several resources handling incoming requests. The resource type is therefore marked with the Handler stereotype. Resources are identified by unique resource identifier as well as metadata associated with it, enabling the creation of HATEOAS links and other elements needed to from a uniform interface of a RESTful API. As noted by Fielding, RESTful APIs are not restricted to a specific communication protocol and serialization even though most RESTful APIs use HTTP and JSON as the serialization mechanism [96]. The extensibility of the RESTful architectural style in regards to communication protocol as well as serialization format is modeled using the HTTP request, HTTP response, and JSON resource representation types. The JSON resource representation is a subclass of the resource representation type annotated with the serialization stereotype. Each resource representation contains specific metadata such as resource specific HATEOAS links that are encoded in the response returned to the web service client. As noted in several metamodels described above, the request as well as the response contain a media type that specifies the requested serialization format, e.g. the JSON resource representation.

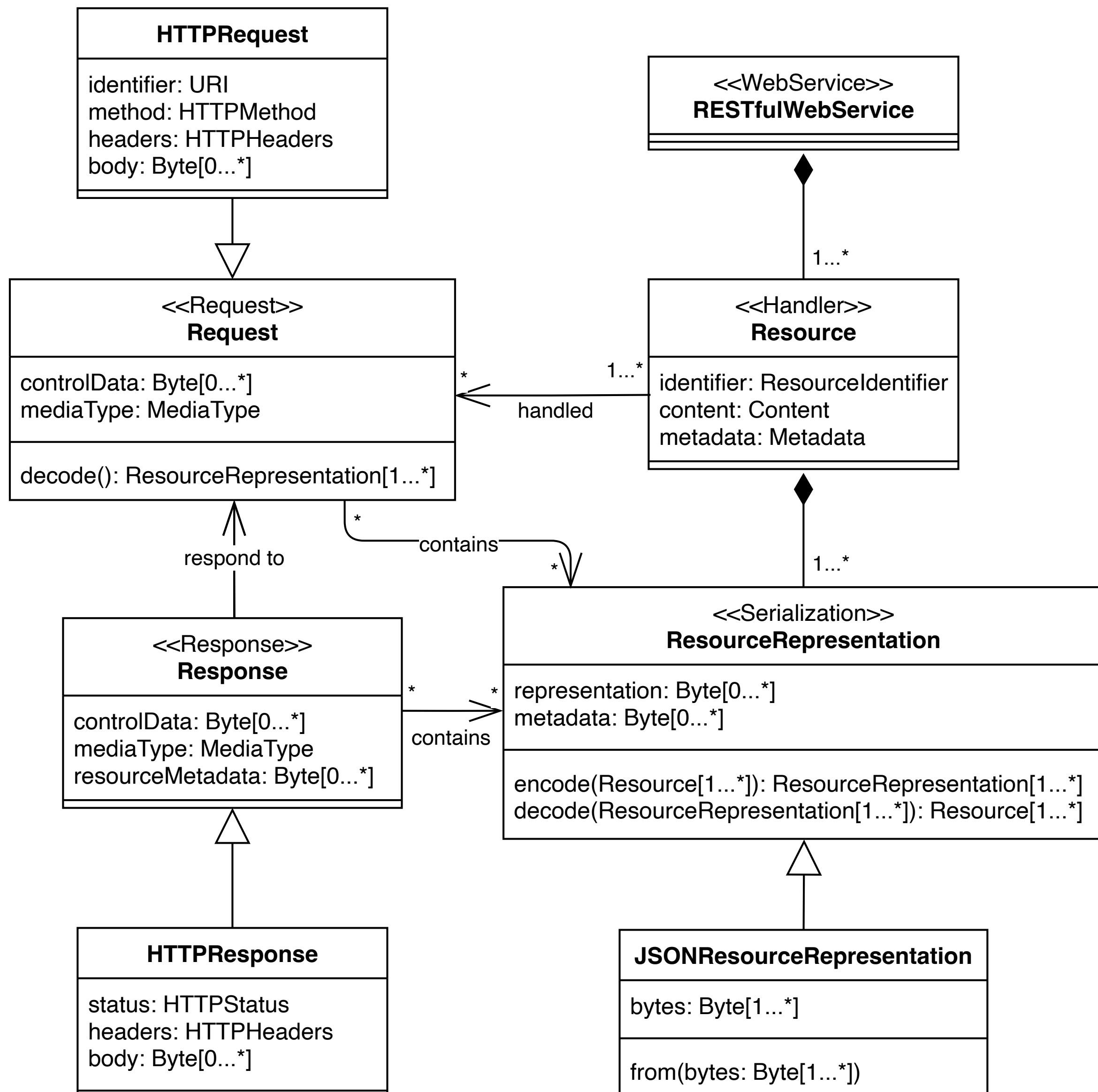


Figure 3.2: A model of RESTful web APIs conforming to the metamodel presented in Figure 3.1. The core concepts of RESTful APIs are enhanced by the stereotypes defined in the web service interface metamodel, detailing how resource-based APIs can conform to the metamodel. (UML Class Diagram)

GraphQL-Based Web API Model

Figure 3.3 presents a UML class diagram describing the main entities that are part of a GraphQL web API according to the GraphQL specification [93]. The GraphQL service annotated with the web service stereotype consists of several operations handling requests of a GraphQL web service. An operation is either a query providing idempotent and safe operations to query data of a GraphQL web service, or mutation, mutating data. A subscription is a subclasses of a query, providing a way for web service clients to subscribe to changes of data returned by a query. Similar to

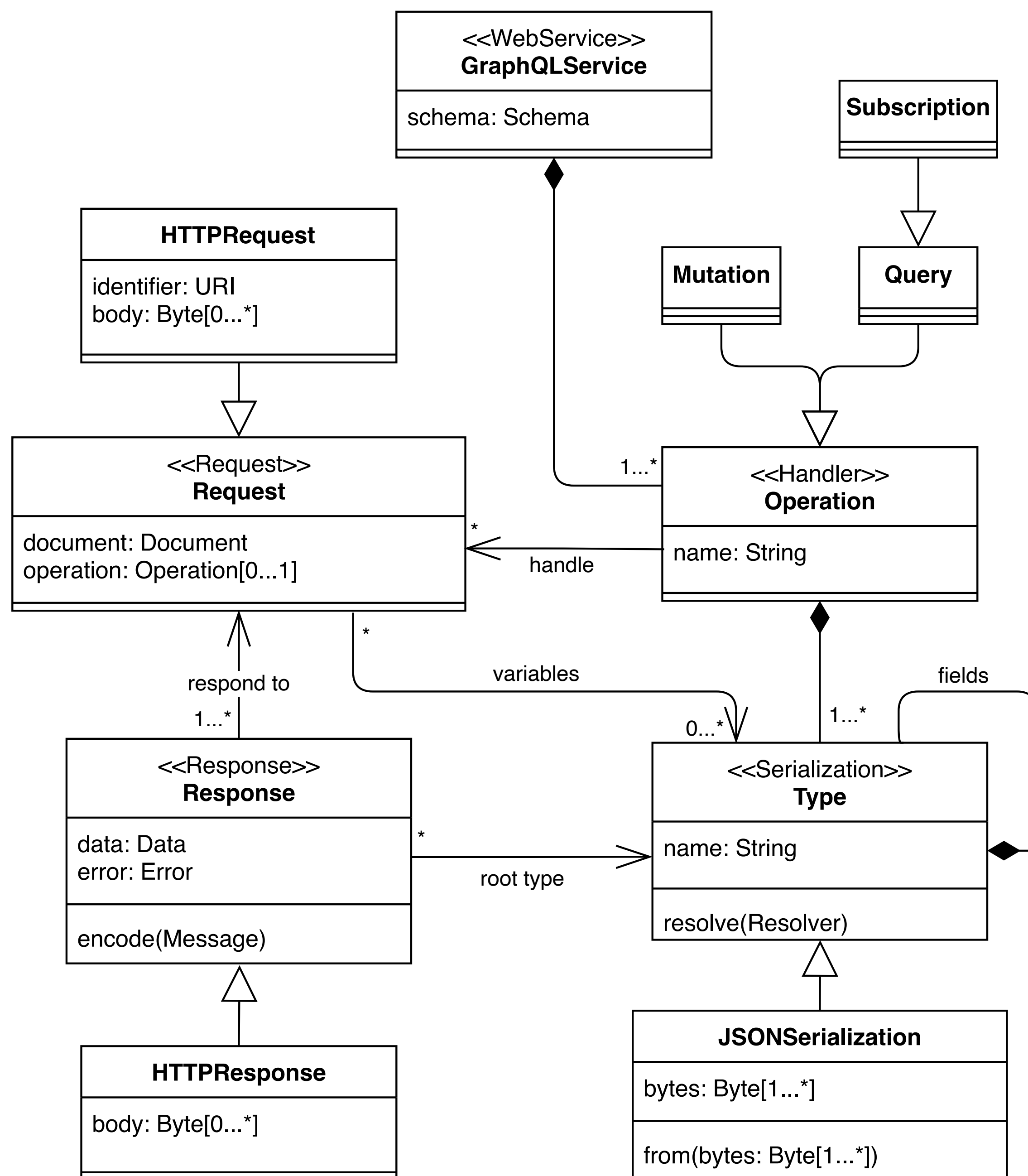


Figure 3.3: A model for GraphQL-based web APIs conforming to the web service interface metamodel presented in Figure 3.1. The GraphQL API type presents how an instance of a message-based web service interface can conform to the web service interface metamodel using stereotypes defined in the metamodel. (UML Class Diagram)

RESTful APIs, GraphQL also does not explicitly define HTTP as the only possible communication protocol. Still, HTTP is the common choice when serving GraphQL APIs[14]. Therefore, the request as well as the response types have specific HTTP-

[14]The official GraphQL website states that *"HTTP is the most common choice for client-server protocol when using GraphQL"*: https://graphql.org/learn/serving-over-http/.

based subclasses. A GraphQL request can contain several variables expressed using the GraphQL type schema. Similarly, a GraphQL response contains a GraphQL type as the root type of the response. GraphQL type schema is used to resolve different types based on application specific logic possibly containing a nested resolving process based on different fields contained in a GraphQL type. GraphQL types represent the serialization of the web service interface metamodel, most often being instantiated by a JSON serialization.

gRPC-Based Web API Model

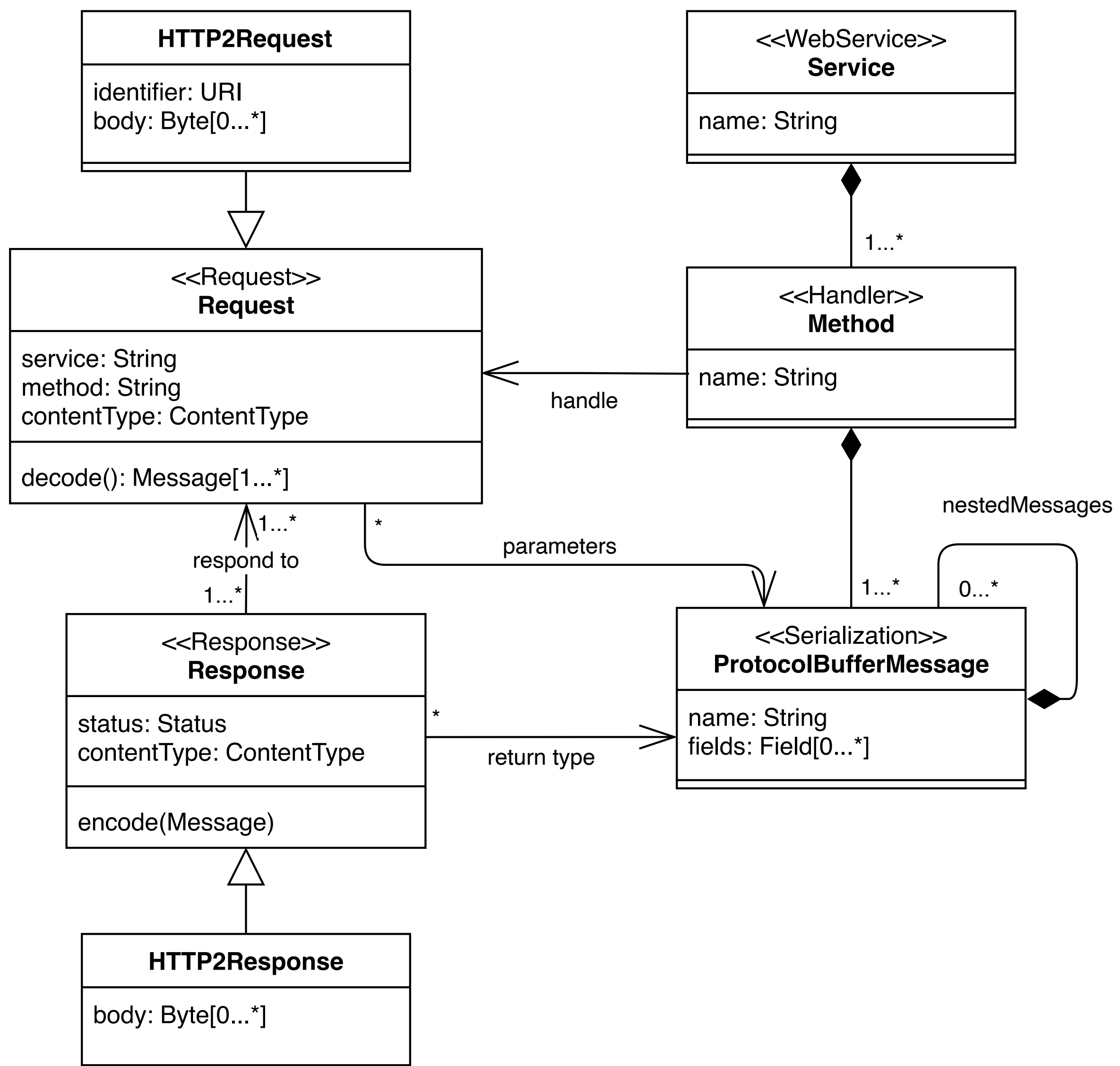


Figure 3.4: A model for gRPC web APIs conforming to the web service interface metamodel presented in Figure 3.1. gRPC is an instance of the RPC-based web service interface types using RPC mechanisms and patterns to offer functionality to web service clients. (UML Class Diagram)

Figure 3.4 presents a UML class diagram describing the main entities that are part of a gRPC web API. A gRPC service contains several methods that can be invoked using RPC requests. A method is a Handler uniquely identified by its name that expects a request that can contain several parameters expressed by nesting several gRPC Protocol Buffer messages into a single message type. Each method has a single request and response type associated with it. Requests as well as responses can be reused across different methods. The serialization is provided using the Protocol Buffer message types containing several fields storing primitive types or nested messages. The messages are only serialized and expressed in the Protocol Buffer format defining the encoding and decoding of the data exchanged with the web service [114]. HTTP/2 is the most common transport protocol for gRPC[15] requests and responses as indicated by the HTTP/2 Request and HTTP/2 response subclasses.

Figure 3.2, Figure 3.3, and Figure 3.4 showcase three web API types conforming to the web service interface metamodel presented in Figure 3.1. They validate the conformance of the generic web service interface model to different web service interface types and web API type instantiations: REST, GraphQL, and gRPC. The metamodel, as well as the models describing the web API types, build the foundation of the design of artifacts addressing web service interface evolution in Part III. The conformance of the web API types guarantees that web service descriptions designed in conformance to the metamodel can be transformed into several web interface types and concrete web API types, enabling web service interface evolution.

[15]gRPC over HTTP/2 is described as the native gRPC protocol: https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-HTTP2.md.

Chapter 4

Web Service API Evolution

Web service API evolution is a subset of web service evolution concerning the evolution of concrete web APIs during the lifetime of a web service. As described in Chapter 3, web service API evolution is distinct from web service interface evolution (Definition 6, page 7). In contrast to the addition, removal, or evolution of complete web API types composing web service interfaces, web service API evolution is solely concerned with fine-grained changes in web service API.

Web Service API Evolution (Definition 7)
Web service API evolution encompasses all additions, removals, or modifications to a web service API that can be categorized into breaking and non-breaking changes specific for each web API type.

Bruegge and Dutoit note that *"Change pervades the development process"* [62]. Requirements, technologies, and design goals evolve, resulting in system, component, and interface design changes that affect every artifact that uses or consumes the evolving software system [62]. It is essential to keep track of requirements and changes, in general, to assess their impact on other software systems. According to Arnold and Bohner, *"Impact analysis (IA) is the activity of identifying what to modify to accomplish a change, or of identifying the potential consequences of a change"* [23]. Arnold and Bohner define a framework classifying impact analysis approaches by their application, parts, and effectiveness [23]. When performing an impact analysis, a change is defined by a change specification that is used to describe the impact based on the approach and how its parts are applied to perform the analysis [23].

As noted by Treiber et al., *"Changes of requirements are the main driver for all evolutionary Web service changes"* [280]. If the changed requirements affect the web service functionality, web APIs offered by a web service have to evolve to reflect these changes [280]. With the increase of edge-dominant systems, these requirements constantly change, emerge from different consumers, producers, and developers, conflict with each other, or might never be fully defined [31]. Different forces in-

fluence the publishing of new versions of a web service. Innovation and business perspectives prefer releasing changes as soon as possible, while IT governance suggests releasing with care, resulting in compromises gradually publishing releases to a growing set of consumers [44]. These influences lead to a constant evolution of web service APIs, making impact analysis on stakeholders such as web service clients an essential aspect of web service evolution.

In the case of web service API evolution, we investigate the impact of web service API changes on clients using a version-centered approach. Compared to history-centered approaches, a version-centered analysis focuses on the comparison between two versions and how elements of the software system evolved in this time-frame [113]. Versioning is a technique to define an order or releases that correspond with features and API definitions that can be used and consumed by web service client applications [78].

The semantic versioning strategy is a strategy that defines rules and requirements, dictating how versions are assigned and incremented to provide stability guarantees for specific version increments to API consumers [228]. Semantic versioning defines versions as a combination of *major.minor.patch* increments where the major version is incremented when an API introduces backward-incompatible changes, the minor version is incremented when purely backward-compatible functionality is added, and the patch version is incremented when the API only introduces backward-compatible bug fixes [228]. Backward-compatible changes are also referred to as non-breaking changes, while backward-incompatible changes are referred to as breaking changes, highlighting the crashing or non-compiling impact of backward-incompatible changes on API consumers and web API clients. In addition to the regular *major.minor.patch* format, the semantic versioning specification also allows for hyphen-appended labels for pre-release versions as well as plus-sign-appended labels for build-related metadata that can be used to provide further flexibility and information in the versioning scheme [228].

Thomas Erl presents more general versioning techniques for specification languages such as WSDL-based web services and web service contracts by defining three versioning strategies: strict, flexible, and loose [89]. Erl defines backward-compatibility as continuing to fully guarantee the non-breaking support of web service clients designed with an over version of the API, while forward-compatibility defines designing a web service, so it anticipates future versions of web service client applications [90]. A strict versioning strategy requires the generation of a new contract and new namespace in case of WSDL contracts with any kind of change to the web API, while a flexible strategy only requires forcing a new contract version when backward-incompatible changes are introduced [90]. A loose versioning strategy also requires that incompatible web API changes result in new contract versions while generally enabling forward- and backward-compatibility [90]. The forward-

and backward-compatibility is achieved by deliberately using vague service contracts involving elements such as wildcards usable in WSDL or XML Schema specifications [90]. While the loose versioning strategy can lead to greater flexibility for web service developers, the use of wildcards results in less expressive web API descriptions and a potential misinterpretation of the API contract on the web API client-side.

Even though properly-used versioning strategies can improve the experience using APIs, they alone can not guarantee change and evolution stability when using web services as defined by the web service client stakeholder goal. Wittern et al. detail four challenges that are commonly related to consuming web APIs [313]. The four challenges manifest themselves as follows: in contrast to local libraries that can be embedded into software, (1) web APIs clients might have no control over the web service, which can potentially break the API contract at any time [313]. Web service clients (2) cannot rely on compile-time checks for API calls as they can with local APIs, and even if clients consume the web service using client stubs generated from an interface definition language, (3) there is no guarantee that the generated stubs do not get out of sync with the web service API [313]. The last challenge (4) is related to the inherently distributed nature of web services, including asynchronous calls, latency, and general quality of service issues that require deliberately designed client code, software patterns, and architectures [313]. Wittern et al. describe a vision of web API research that includes the creation of a web API *"refactoring support, for example, to help developers to migrate to new web API versions"* [313] to truly enable change and evolution stability when using web services APIs.

Amundsen defines three *"principles of safe and effective API changes: First, do no harm. Fork your API. Know when to say no"* [20]. These principles indicate that (1) an API should not unexpectedly introduce breaking changes for API consumers, that (2) multiple consumer-breaking API versions should be provided in parallel to allow migrations from one version to the next, and that (3) API developers should be careful about changing APIs and sometimes have to decline client requests for API changes [20]. Similar to the refactoring support described by Wittern et al. [313], Amundsen mentions the usage of *"'migration kits'-utilities and guides"* [20] to support manual or semi-automatic migration from one major API version to the next. In addition to the principles listed above, Amundsen also provides rules and patterns to safely evolve and test web APIs changes to reason about the introduction of web service client-breaking changes [20]. Nevertheless, the manual creation and application of utilities and textual migration guides requires an in-depth understanding of the web API changes.

Artifacts designed as part of this research project should easily enable consumer applications to co-evolve with web services when changes are made to the web service API. Similar to the co-evolution of services as discussed by De Sanctis et al., the

co-evolution of web service client applications may require unforeseen maintenance to the consumer code when web services introduce breaking changes and, ideally, a way to express these changes to automatically reason about them [81]. Research by Espinha et al. describes how unexpected breaking changes and a lack of clear policies impact client developers when breaking changes occur [92, 91]. Therefore, web service consumers need to anticipate change and, due to the loosely-coupled nature of web services, have to rapidly adapt to changes of web APIs as *"in general, all web API providers will sooner or later impose changes on their clients"* [92, 91].

> **Design Problem 2:**
> Automatically detect and migrate backward-incompatible changes of web service interfaces to enable web service client stability after modifications to the web service interface.

The following chapter provides an overview of the challenges associated with web service API evolution. We aim to investigate the problem context surrounding Design Problem 2. Section 4.1 details several artifacts and approaches that have been developed to address web service API evolution. The presented solutions provide the groundwork for artifacts designed in Part III addressing web service API evolution using a semi-automatic migration approach. In contrast to the presented related work, our approach combines the challenges of web API evolution with the other web service evolution-related challenges described in Chapter 1. The design of these artifacts requires a web service interface and web service API-independent classification of web API changes that is presented as a collection of web service API evolution patterns in Section 4.2. These web service API evolution patterns are subsequently compared to existing web service API change types.

4.1 Related Work

This section provides an overview of related work concerning solutions tackling web service API evolution. While web API evolution comes with several unique challenges, the decades-old research on local API evolution of libraries and frameworks also provides valuable insights into the challenges of web API evolution. The section aims to investigate several web API evolution mitigations to address Knowledge Question 3.

> **Knowledge Question 3:**
> What migration strategies can be used for different web API evolution change types?

4.1.1 Local API Evolution

Dig and Johnson investigate framework API evolution with four years old and matured frameworks that are used by a substantial number of users that might be impacted by breaking changes [84]. The study analyzes API changes and classifies them into several groups while noting that more than 80% of the breaking changes result from refactoring, such as moving of methods and fields, as well as renaming or changing method signatures [84]. Dig and Johnson suggest *"that component producers should document the changes in each product release in terms of refactorings"* [84]. This can also apply to web service APIs as refactoring and migration descriptions might also provide valuable insights for web service clients. In their publication *How do APIs evolve? A story of refactoring*, they suggest the creation of a migration tool that can automatically detect and record refactoring in a log when the framework developer performs it and replays it at the client application's side [84]. The creation of refactoring logs can also be applied to the evolution of web APIs. Tools detailed below create similar documents to detail changes between versions of frameworks or web APIs and build the inspiration of the web service interface independent semi-automatically generated migration guide detailed in Part III.

Dig and Johnson mention *CatchUp!* by Henkel and Diwan [127] as a research prototype meeting the requirements of such a migration tool with the support for a limited number of refactorings [84]. *"CatchUp! [...] captures and replays refactoring actions within an integrated development environment semi-automatically"* [127] without the need for a centralized infrastructure and instantiated for in the Eclipse integrated development environment (IDE) [127]. The created migration specification details how classes in an old version of the library are mapped to newer versions, demonstrating the usage of the plugin for Java applications with a success rate of 90% for the observed and tested refactorings [27]. The tool demonstrates the power of automatic migrations between versions. Nevertheless, a more generalized migration description approach as documented for *CatchUp!* will be needed because web services and web clients often are not developed in the same programming language.

Instead of recording refactorings between different versions, Brito et al. introduce APIDIFF as a tool to *"identify API breaking and non-breaking changes between two versions of a Java library"* [57]. APIDIFF compares two versions of Java libraries stored in git repositories while categorizing them into breaking and non-breaking changes, as well as several refactoring types excluding API parts marked as deprecated [57]. Section 4.2 further details the API change types described by Brito et al. and compares them with generalized evolution patterns for web APIs. Tools like APIDIFF and other web API-specific tools to detect changes in API surfaces provide valuable insights into the generation of automatic detection of changes in web APIs as presented in Part III.

4.1.2 Web API Evolution Strategies

Before we present possible ways to automatically identify web API evolution, that can be classified in evolution patterns in Section 4.2, we present strategies to reason about introducing web service API evolution in the first place. The strategies presented in this section primarily focus on reasoning about web service client breaking changes. These types of changes highlight the conflicting stakeholder goals to provide evolution stability and at the same time allow evolvability when developing and deploying web services.

Lübke et al. present eight development strategy patterns concerning versioning strategies as well as the lifetime of individual versions that can be applied to the design of a web service API evolution strategy [182]. The first three patterns defined by Lübke et al. suggest (1) the creation of an *API Description* to share knowledge about the API with the web service client, (2) the usage of *Version Identifier*s in exchanged messages, and (3) the usage of the *Semantic Versioning* scheme detailed in Chapter 3 [182]. The next three patterns detail different lifetime strategies for individual versions: (4) the *Two in Production* pattern suggest the overlapping deployment of at least two API versions to simplify client migrations, while (5) the *Limited Lifetime Guarantee* suggests explicitly providing fixed time frames until an API version will be removed, and (6) the *Eternal Lifetime Guarantee* suggests providing an unlimited lifetime guarantee of clients are unable to update to new versions [182]. Lastly, (7) the *Aggressive Obsolescence* pattern suggests aggressively communicating short limited lifetime guarantees to push clients to update to newer API versions while (8) the *Experimental Preview* pattern suggests providing previews of new API versions to collect feedback from web service clients [182]. Lübke et al. also highlight the relevance of correct and updated machine-readable API descriptions using interface definition languages to simplify the evolution of web APIs for client developers [182]. The patterns defined by Lübke et al. provide useful guidance for web API developers and highlight the challenge of web API users to migrate to later versions and the resulting complications for web API developers to support multiple versions or create and maintain sophisticated API evolution strategies. Our research aims to reduce the overhead for web API consumers created by web API evolution. We provide tools and infrastructure to easily migrate from one version to the next version of the web API, therefore, reducing the pressure put forward by web API developers when deprecating old API versions as suggested by Lübke et al.

Newman provides several suggestions to avoid and mitigate breaking web API changes in the book *Building Microservices: Designing Fine-Grained Systems* [198]. Newman suggests avoiding breaking changes by automatically detecting accidental breaking changes as described in Section 4.1.3 and using the tolerant reader pattern and technologies that can more easily manage backward-incompatible changes as described in Section 4.1.5 [198]. Once breaking changes occur in a microservice

setup, Newman suggests using mechanisms such as a lockstep deployment, coexisting incompatible microservice versions, as well as the emulation of old web APIs to continue supporting non-migratable web service clients [198]. Details about the migration and emulation of old web API versions are further described in Section 4.1.4 detailing several tools to automatically migrate web API users to newer web API versions while providing a consistent interface for the API consumer.

De Sanctis et al. describe the challenges of co-evolution of web services and detail an approach using evolution management agents that coordinate evolution across web services, which might resolve changes automatically or notify developers, if manual adaptions are needed [81]. De Sanctis et al. describe several structural and behavioral evolutionary changes (described in detail in Section 4.2.2) that can partially enable an automatic migration while some patterns require manual migrations sending an evolution request to the development team [81]. The combination of a process model with added tooling support presented by De Sanctis et al. provides web service developers as well as consumers with the ability to better manage web service API evolution.

The change-centric web service model by Zuo proposes solutions for web API evolution by addressing unambiguity in web API changes, incorporating web API evolution in web service development, enabling graceful versioning detailing a delta to previous versions, and allowing consumers to monitor changes and adapt client applications accordingly [323]. The WSDL-focused delta between versions is provided by the web service developer and helps all stakeholders to understand web API evolution, an important impact for the generalized artifacts designed in this dissertation [322, 323].

4.1.3 Web API Change Identification

Fokaefs et al. present VTracker, a tree alignment algorithm based on an algorithm to compute the editing distance between trees presented by Zhang and Shasha [319] and Mikhaiel et al. [190], that can be used to compare WSDL specifications. VTracker only focuses on web API evolution by using a simplified version of the WSDL specification concerning operations and the input and output of these operations represented with data types [100]. VTracker generates insights about changes between two web API versions and identifies typical changes in WSDL-based web APIs used in production [100]. Building on top of VTracker, Tsantalis et al. present WebDiff, a web-based frontend application that allows developers to inspect and observe the process of comparing two interface specifications [282].

To expand the usage of VTracker to more web API types and more easily compare web services described in the WSDL and WADL service description languages, Fokaefs and Stroulia introduce *WSMeta: A Meta-Model for Web Services to Compare Service Interfaces* [103]. In comparison to the artifacts designed in Part III that are in-

tended to describe web services independent of the web service interface type and web API type, WSMeta only aims to provide a metamodel to compare web service interfaces [103].

A toolset addressing web API evolution is WSDarwin, developed over multiple years as part of the dissertation of Marios-Eleftherios Fokaefs named *WSDarwin: A Comprehensive Framework for Supporting Service-Oriented Systems Evolution* [106], as well as several publications by Fokaefs and colleagues [99, 100, 101, 102, 103, 104, 105]. Building on top of the insights gathered from VTracker and WSMeta, the web service interface comparator in WSDarwin supports web service API evolution based on evaluating the information contained in web service description artifacts such as WSDL and WADL documents [102]. Identifying changes between two versions of a web service specification allows web service clients to analyze the impact of the changes and effectively use the gathered insights in an adaptation or migration process as described in Section 4.1.4 [102]. In contrast to VTracker which uses a generic comparison algorithm for XML documents, WSDarwin is developed explicitly for comparing web service interfaces and can take advantage of additional domain-specific context available to the algorithm and provide better results when comparing web service specifications [104]. The domain-specific knowledge is also beneficial for comparing RESTful web APIs using WADL specifications by taking advantage of identifiers as well as data structures enabling WSDarwin to group changes into five change types: Additions, deletions, moves, changes, as well as a combined category of moves-and-changes [101]. Based on validated advantages of WSDarwin, we conclude that the comparison algorithm building the basis for web service API evolution detection and migration presented in Part III should also take advantage of a domain-specific comparison to identify web service API changes.

In their publication *Managing the Evolution of Service Specifications*, Andrikopoulos et al. introduce the concept of service evolution management by developing a formal model for web service evolution to identify and track changes between different web service versions [21]. Service evolution management, according to Andrikopoulos et al., encompasses the identification and classification of changes, propagation analysis mechanisms to analyze the impact of web API evolution, the validation and conformance checks to validate service updates in accordance with existing contracts, and migration mechanisms to update entities using the updated services [21]. Andrikopoulos et al. provide theoretical foundations for service evolution management by designing a technology-agnostic model to *"identify and study the changes happening to a service during its lifetime"* [21]. The publication reasons about impacts and changes in web services using an Abstract Service Description (ASD) model that encompasses different elements of web services that might be impacted by web service evolution, including a structural layer relevant for web service API evolution [21]. Building on top of the work done by Andrikopoulos et

al., Vara et al. reason about web service evolution using several domain-specific languages and model transformations enabling the transformation of WSDL files into the Abstract Service Description (ASD) [289].

Romano and Pinzger present WSDLDiff based on the UMLDiff algorithm [315] as a tool to compare WSDL interface specifications of subsequent web service versions by comparing operations, messages, and types; grouping them as additions, moves, modifications, and removals [241]. The tool parses the WSDL interfaces, transforms it to an internal XML Schema Definition (XSD), and applies a matching algorithm using the Eclipse Modeling Framework (EMF) to detect matching nodes and subsequently identify differences reported in a tree of structural changes [241]. The output of WSDLDiff can be used to analyze typical changes in a WSDL-based web service interface and can benefit web service consumers to classify web services based on their historical web service API evolution stability [241]. WSDLDiff and UMLDiff provide valuable insights into the identification of changes in a web service interface and offer algorithmic foundations to identifying changes in the artifacts introduced in Part III.

4.1.4 Web API Evolution Migration

As discussed in Section 4.1.3, WSDarwin offers tools to support web API evolution, including a service-interface specification generator, a service-interface comparator, a client-proxy generator and client-application adapter, as well as a cross-vendor service mapper [106]. The service interface specification generator enables clients to automatically generate WADL specifications for RESTful web services without the need to access the web service's source code [101, 106]. The WADL generator expects endpoint URLs and suiting HTTP requests and automatically sends requests to the web service to inspect requests and responses, combining all gathered information about the web API into a single WADL specification [101].

The service-interface comparator and client-proxy generator in WSDarwin address a manual process: When a web API user detects a breaking change in an API, the web API user needs to compare the web service specifications, identify the nature of the breaking change, perform adaptations in the client code, and finally verify the changes using tests [104]. WSDarwin offers the tools to automatically compare XML-based specifications such as WADL and WSDL and provides a client adaptation algorithm to support the migration of client applications based on the detected changes [102]. WSDarwin allows the migration of client stubs from one version to the next by creating a new internal client stub: While the client application still interacts with the old stub it internally maps to calls to the new stub using a description produced during the web service API comparison [102].

Developing migration adapters can also originate from the heterogeneity of web service API types as well as a high number of heterogeneous clients consuming dif-

ferent web APIs concerning the same application domain [39]. Work by Benatallah et al. focuses on business protocol adapters using mismatch patterns to identify differences between services and guiding developers to create adapters in interface as well as business-level protocols [39].

Zuo presents a multi-step API evolution migration process based on the change-centric model discussed in Section 4.1.2. The process focuses on WSDL-based web services monitoring changes in service specifications and analyzing the impact of changes to web service consumers [321]. The toolchain generates adapting strategies based on the web API changes and subsequently creates a web service proxy adapting the web API changes transparent to the web service consumer [323].

Kaminski et al. present the chain of adapters as a design technique to address web API evolution and migrate web API changes on the web service-side [149]. The approach builds on the adapter pattern defined by Gamma et al. that allows deevelopers to *"Convert the interface of a class into another interface clients expect"* [109]. Kaminski et al. propose to duplicate the current web API in a new namespace, adapt the new interface according to the new requirements, and delegate all calls from the old interface to corresponding functionality in the new interface [149]. The implementation of the old interface uses adapters to maintain the contract of the new interface, e.g., additions require default parameters, changes require type translations, and removals must be reimplemented in the old interface [149].

The migration approaches presented in WSDarwin, as well as the general concept of a chain of adapters, offer an essential related work for the artifacts designed in Part III. The chain of adapters provides important insights into the required information needed to implement suiting adapters for different types of web API changes. In contrast to WSDarwin and other tools, our research aims to connect the identification and migration challenges of web API evolution with the challenges of web service interface evolution. While WSDarwin, the underlying comparison algorithms, and other tools presented in this subsection focus mainly on the XML-based WSDL and WADL specifications, our research aims to generalize the insights for multiple web service interface and web API types.

4.1.5 Protocol-Enabled API Evolution

Existing research to identify web API changes as demonstrated in Section 4.1.3 primarily focuses on XML-based interface definition languages to identify changes. These XML-based interface definition languages, such as WSDL and WADL, result in classifications of changes influenced by protocol-specific constraints as demonstrated in Section 4.2. In contrast to these IDLs, other middleware- and protocol-types have different mechanisms to be more resilient to changes in the web API, therefore, reducing the number of changes manifesting themselves as breaking changes for a web service client.

The tolerant reader pattern is a mechanism to improve stability when sending and parsing protocol-conformant messages exchanged between different components [78]. The tolerant reader pattern originates from Postel's law, defined as part of RFC 760 presenting a first version of the internet protocol: *"In general, an implementation must be conservative in its sending behavior, and liberal in its receiving behavior"* [1, 223]. The tolerant reader pattern, therefore, suggests to *"extract only what is needed from a message and ignore the rest"* [78]. Messages received by a tolerant reader should be continued to be parsed in a best-effort approach even if the reader detects a resolvable schema violation or ignore missing values that are not required to fulfill the desired functionality on the reader side [78]. The tolerant reader pattern can also be applied and used when reasoning about web service API evolution to mitigate breaking changes on the serialization parsing or protocol level. Even though removals of a web API functionality result in a breaking change, using mechanisms like the tolerant reader approach might mitigate the impact of the breaking change for web service clients not relying on specific removed functionality.

Different communication middleware types and protocols provide built-in functionality to mitigate potential breaking changes in web APIs. Apache Avro is a data serialization framework that can be used to implement remote procedure call-based systems using a binary encoding with built-in mechanisms to evolve schemes using a Schema Resolution mechanism [22]. A reader of local data or RPC messages must have the "same" schema, as defined by the Parsing Canonical Form of a schema used for the Schema Resolution mechanism [22]. The Schema Resolution mechanism includes several rules that allow identifying compatible schemes, providing a mechanism to classify changes as compatible or breaking changes [22].

Similar to Apache Avro, Apache Thrift also includes mechanisms to reduce the impact of traditional breaking changes for web API consumers on an API and the protocol abstractions [263]. Apache Thrift uses field identifiers in types, allowing field names to change without resulting in a breaking change for web service consumers [263]. In accordance with the tolerant reader pattern, Apache Thrift defines that unexpected fields should be ignored and discarded, and missing fields should be expressed using explicit null values in compatible programming languages [263]. Mark et al. describe several cases where the mechanisms introduced in Apache Thrift result in breaking changes, e.g., when fields for a request are added, fields needed by a client are removed [263]. The Protocol Buffer serialization format used for the gRPC middleware also uses field identifiers providing similar advantages and mechanisms as Apache Thrift [114].

In summary, the API evolution mechanisms in different middleware- and protocol-types have to be considered when developing the artifacts in Part III. Protocol-enabled API evolution can automatically mitigate breaking changes into non-breaking changes, simplifying the adapter generation for some web API types.

4.2 Web Service API Change Classification

To address the impact of web API evolution requires identifying and classifying changes that occur in a web API. This section introduces a generic web API type-independent evolution patterns showcased in Table 4.1. The proposed classification allows us to combine different aspects of web service evolution, including web service interface evolution and web service API evolution, in artifacts designed in Part III. Subsequently, we compare the introduced web API change patterns with different existing web API-specific classifications.

4.2.1 Web Service API Evolution Patterns

In this section, we introduce web service API type-independent evolution patterns to classify web API evolution-related changes. The evolution patterns address Knowledge Goal 2 and build on Section 4.1 to answer Knowledge Question 3.

> **Knowledge Goal 2:**
> Identify web service interface type-independent change patterns to enable web service evolvability and change while providing stability to web service clients.

> **Knowledge Question 3:**
> What migration strategies can be used for different web API evolution change types?

We specify and categorize the patterns based on the compact pattern template similar to the (anti-)pattern templates presented by Brown et al. [61]. We use a variation of the inductive mini-pattern template using a name, context, forces, and a solution [61]. A web API evolution pattern is defined by a four-tuple consisting of a context, change type, classification, and migration strategy. The name of the pattern is constructed using a combination of the context and change type, e.g., Handler Addition. The context of the mini-pattern template corresponds to the context of the evolution pattern. The force is specified by the change type of the web API evolution pattern. The solution is defined by the combination of the classification and the migration strategy.

Table 4.1 details an overview of the web API change patterns based on the concepts defined in the web service interface metamodel (Figure 3.1, page 48). The entities from the metamodel are used as the contexts of the web API evolution patterns. The evolution patterns are based on the related work collected in Section 4.1 and are subsequently compared to other change classifications in Section 4.2.2. Table 4.1 details a compact overview of the web API evolution patterns. The following section details additional context and information about the patterns.

Context	Change Type	Classification	Migration Strategy
Handler	Addition	Non-breaking	-
	Removal	Breaking	Fallback response
	Handler identifier	Breaking	Identifier mapping
	Communication pattern	Breaking	Multiplicity conversion
Request Serialization	Add field	Breaking	Default value
	Remove field	Non-breaking	-
	Field identifier	Breaking	Identifier conversion
	Root or field type	Breaking	Type conversion
Response Serialization	Add field	Non-breaking	-
	Remove field	Breaking	Fallback value
	Field identifier	Breaking	Identifier conversion
	Root or field type	Breaking	Type conversion

Table 4.1: Overview of the proposed web service API type-independent web service API evolution patterns. The patterns refer to the entities presented in the web service interface metamodel (Figure 3.1, page 48). Following the web API evolution pattern template, the patterns identify possible change types to a context, a classification into breaking and non-breaking changes, and migration strategies.

A Handler can be added or removed from a web service. Additions result in no breaking changes for web service consumers, while a removal results in a breaking change eliminating functionality from the web API. The removal can be migrated if the web service developer can provide a placeholder response, or a placeholder response can be constructed based on the combination of responses from one or more Handlers. Besides additions and removals, Handler identifiers and communication patterns can also change, resulting in breaking changes. The identifier change can be migrated by providing an identifier mapping, allowing the web service consumers to identify the changed Handler based on the old identifier in the new version of the web API interface. Changing a Handler's communication pattern (as introduced in Section 3.2.1), is also a breaking change. The communication pattern change can be migrated if a multiplicity mapping can be provided, e.g., describing how multiple responses can be combined into a single response.

Serialization of requests can experience several types of changes. A field can be added, leading to a breaking change as the web service client needs to provide a value for the field to successfully construct a request to the web service. Providing a default value for the added field allows the web service client to migrate to the new

version of the web service interface without any changes in the client application. Removing a field from a request serialization is a non-breaking change assuming the web service adheres to the tolerant reader pattern and accepts requests with additional fields. While removing a field is syntactically non-breaking, it has significant implications on the application structure and meaning of requests which should be considered when consuming subsequent versions of the web service API. To alter the field identifier and to modify the root or field type of a request serialization result in breaking changes. Both evolution patterns require mapping the old identifier or instances of the old type to the identifier or instances of the new type.

Response serialization features similar change types but differ in their classifications. The evolution pattern of adding a field to the serialization of responses is no breaking change. Similar to removals from request serializations, this classification assumes that best practices such as the tolerant reader pattern are applied. This means that responses with additional fields are accepted, and the client application ignores unknown fields. The removal of a field for serialization of responses is a breaking change as web service clients might expect the functionality in the client applications. The breaking change can be migrated if a placeholder value can be provided or the web service client is provided with mechanisms to compute the removed value from other web API responses. The change field identifier and change root or field type changes for response serializations result in the same evolution patterns as for request serializations. They are breaking changes and require mappings for identifiers and types to migrate a web service client application.

4.2.2 API Change Classifications Comparisons

Classifying interface evolution is not only relevant for web service APIs, but is a general challenge for interfaces between subsystems and software components, including local APIs offered by frameworks and libraries. Brito et al. provide an overview of change types detected by the APIDiff tool, allowing developers to identify breaking and non-breaking changes in Java libraries as described in Section 4.1.1 [57]. The findings by Brito et al. show that breaking changes in types, methods, and fields are mostly related to refactoring actions such as renames, moves, removals, changes in default values, and visibility reductions [57]. Non-breaking changes are mostly related to additions, visibility modifier changes to gain visibility [57]. These change classifications can also be applied to similarly typed remote procedure call and message-based middleware types.

Biehl introduces several classifications for API evolution, focusing on RESTful and SOAP-based APIs and classifying the changes into backward-compatible, forward-compatible, and breaking changes [44]. Biehl lists several backward-compatible changes primarily related to additive changes to a web API, such as adding parameters, adding fields in JSON or XML encodings, new REST endpoints, new

SOAP operations, new optional fields, and making mandatory fields optional [44]. Added SOAP operations and REST endpoints can be grouped under the Handler addition evolution pattern. The addition of an optional field or parameter can be grouped as an addition of a field in a request serializations that already provides a default null migration value. The forward-compatibility mentioned by Biehl, that guarantees that a new client can also communicate with an old version of an API, is not a concern for the backward-compatibility-focused nature of the artifacts presented in Part III [44]. The backward-incompatible changes introduced by Biehl can also be mapped to the existing web API evolution patterns described in Table 4.1. The presented classifications such as removing or changing data structures, requiring previously optional fields, introducing new fields to a request are all encompassed by the request and response serialization web API change patterns [44]. Changing the URI of an endpoint or the complete web service identified as a breaking change by Biehl is also covered by the Handler identifier evolution pattern [44].

Robert Daigneau describes several actions performed on a web API interface that may cause breaking changes for client applications, such as the removal of elements and attributes that can be matched to the more concrete removal of fields in request and response serializations [78]. While we do not list all individual breaking changes identified by Daigneau, all change types mentioned in *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services* can be mapped to the web API evolution patterns demonstrated in Table 4.1 [78]. Most changes such as the specified *"Changing the hierarchical relationships between complex data structures"* [78] change type can be mapped to more generalized web API evolution patterns, such as root and field type changes in request and response serializations. Changes to the authentication policies, as mentioned by Daigneau, are not an explicit web API evolution pattern but can be interpreted as a change to a request serialization root or field type, changing how authentication information is serialized and expressed in requests from the client application [78].

The common incompatible changes listed in *Service-Oriented Architecture: Analysis and Design for Services and Microservices* by Erl are focused on WSDL and XML-specific changes happening to web APIs [90]. The changes to the WSDL interface such as *"Adding a new required XML Schema element or attribute declaration to a message definition"* [90] can be mapped to the web API evolution patterns presented in Table 4.1 such as adding fields to a request or response serialization. Similar to other authentication changes, changes to WS-Policy assertions or expressions can be mapped to web API evolution patterns concerning request and response serializations or are not taken into consideration if they are purely semantic changes [90].

Leitner et al. introduce a WEDL-driven classification of web API changes focusing on operations and parameters with a particular focus on mandatory and optional parameters [174]. Based on the work of Leitner et al., Sanctis et al. introduce

an evolutionary change taxonomy providing a classification of structural changes that can also be mapped to the web API change patterns introduced in Table 4.1 [81]. The structural changes Leitner et al. and Sanctis et al. present, such as the deletion and addition of operations, can be mapped to the corresponding additions and deletions of Handlers [81, 174]. The change of operation type and operation name corresponds to the change Handler identifier web API evolution pattern [81]. Changes to input and output parameters as described by Sanctis et al. can be mapped to the evolution patterns for request and response serializations [81]. Leitner et al. focus on optional and mandatory parameters that each have a different effect on the possibility to automatically migrate web API evolution for client applications but are not present in every web API type and, therefore, omitted in the generalized table of web API evolution patterns introduced in this section [174]. These web API type-specific possibilities will be incorporated in the artifacts designed in Part III, showcasing the instantiation of the evolution patterns and the extensibility of the taxonomy presented in Table 4.1. The classification by Sanctis et al. in compatible, not compatible but resolvable, and not resolvable can be mapped to the more generalized classification of breaking and non-breaking changes, and migration strategies presented in Table 4.1.

Li et al. analyzed the change types of five popular web APIs, categorizing them into 16 change patterns and comparing the challenges of web API evolution to the evolution of local APIs [177]. The change patterns are divided up into breaking changes that cause compile-time errors when using a client stub generator and runtime errors, including changes such as changing the default value of a parameter, modifying the accepted values of a parameter, and authorization changes [177]. Compile-time errors can be considered breaking changes. The patterns introduced by Li et al. include adding or removing parameters, deleting endpoints, changing request and response types, and further change patterns that are all patterns that are instantiations of the web API evolution patterns introduced in this section [177]. Sohan et al. extend the change patterns introduced by Li et al. with additional change types, such as the move of API elements, the renaming of API elements, and changes in the HTTP header and error handling [266]. Li et al. define several challenges for web API migrations that we categorize as web service interface evolution, such as switching from an XML-based encoding to a JSON encoding [177]. Challenges that are related to web API evolution include deleting endpoints, changes to the authorization protocol as well as the observation that Web APIs tend to change more often than local APIs and affect a more far-reaching API surface than the typical local API evolution described in the publication [177].

Koçi et al. present a *Classification of Changes in API Evolution*, focusing on the interaction of API producers and API consumers and artifacts such as release notes, issue trackers, and version control systems tracking the evolution of web APIs [158].

Koçi et al. focus on HTTP-based APIs, classifying changes into changes on API endpoint identifiers, parameters as part of the HTTP body or query, parameter value constraints, HTTP methods, and changes to the authority levels used to authorize access to HTTP endpoints [158]. The publication inspects two subsequent versions of web APIs and manually classifies the changes such as adding parameters, changing the authority level, and removing HTTP endpoints [158]. Wang et al. present a collection of API-level, method-level, and parameter-level changes for RESTful APIs by comparing subsequent versions of popular web APIs and categorizing them in different change types [302]. The change types range from modifications to the complete serialization of a web API, which we would group as web service interface evolution, to the method and parameter-level modifications, such as the addition and deletion of parameters and methods [302]. Yasmin et al. focus on detecting and classifying deprecations of RESTful web APIs by analyzing OpenAPI definitions and investigating the depreciation patterns used to communicate breaking changes in RESTful APIs to web API users [316]. Similar to the other change classifications introduced above, the change types presented by Koçi et al., Wang et al., and Yasmin et al. formulate subtypes of the web API evolution patterns introduced in this section, showcasing their applicability to HTTP-based and RESTful web APIs.

Most existing web API change classifications focus on specific middleware- and protocol-types used to develop and offer web APIs. The change classifications in this section have shown that these classifications, patterns, and groups can be compared to and seen as specializations of the introduced web API evolution patterns in Table 4.1. These patterns provide a web API independent classification of web API evolution based on the web service interface metamodel introduced in Figure 3.1.

Chapter 5

Web Service Deployment Evolution

Web service deployment evolution is a subset of web service evolution concerning the dynamic and static changes of the deployment structure of a web service:

> **Web Service Deployment Evolution (Definition 8)**
> Web service deployment evolution encompasses dynamic and static changes to the deployment structure due to new requirements, constraints, and execution environment changes.

The web API and web service interface are expressions of functional requirements of web services embedded in non-functional requirements and constraints. The deployment of a web service, in contrast, is dependent on non-functional requirements and constraints to enable the web service functionality. Non-functional requirements that affect the deployment structure, the deployment process, and dynamic deployment reconfigurations at runtime can be categorized under the FURPS non-functional requirement categories: usability, reliability, performance, and supportability [117, 118]. It is crucial to identify, measure, and incorporate non-functional requirements in the development process, including the deployment pipeline [135]. Incorporating and treating the added business value of non-functional requirements is essential, but requires significant preparation and results in various tradeoffs [135]. During system design, design goals document the results from trade-off decisions derived from non-functional requirements [62]. Web services introduce several deployment-related trade-offs that need to be analyzed, documented, and incorporated in development and deployment processes.

The first non-functional FURPS category is usability [118]. According to Nielsen, usability comprises five attributes: learnability, efficiency, memorability, errors, and satisfaction [200]. The resource availability of web service execution environments can heavily influence the efficiency and response time. Inefficient systems can result in systems that are not pleasant to use, resulting in poor user satisfaction and

usability [200]. In regards to efficiency, memorability, and error rate, usability is also involved in the perceived trade-off between usability and system security [24]. Development processes, continuous delivery, and system automation aim to ensure that usability from a developer's and consumer's perspective and security are not opposed concerns when developing and deploying web services [198].

Reliability focuses on the *"ability of a system or component to perform its required functions under stated conditions"* [62]. Reliability models and theories can be used to estimate and predict future failure behavior based on failure sample data and stochastic models [299]. The dependability of a system includes reliability as well as robustness and safety, defining how and under what circumstances a component can still deliver its service [62]. The dependability requirements of web services influence the choice of software execution environments, development, and deployment of the services. The deployment needs to be a repeatable and reliable process that focuses on automation and incorporates the non-functional concerns defined in the deployment, startup, and runtime of the system [135].

Performance non-functional requirements define the response time, throughput, and availability of a software system [62]. Web services must deal with numerous requests, requiring implementations and deployments that can scale with the dynamic requirements at runtime. Web service implementations and deployment structure instantiations have a capacity defining the maximum throughput and therefore the performance of a system [135, 201]. Demand can exceed the capacity, which can be addressed by scaling the system to provide more resources and instances if the system is designed and implemented for scalability [201]. To meet web service client demands, defining scaling mechanisms and a suitable deployment structure requires keeping track of performance non-functional requirements.

Supportability is concerned with enabling change, including software maintenance and evolution [62]. According to Robert Grady, supportability is compromised out of several subcategories including: *"Testability, Extensibility, Adaptability, Maintainability, Compatibility, Configurability, Serviceability, Installability, [and] Localizability"* [117]. The ISO/IEC 25010 standard differentiates between maintainability and portability, refining and extending the supportability FURPS categorization [14]. ISO/IEC 25010 defines portability as the *"degree of effectiveness and efficiency with which a system, product, or component can be transferred from one hardware, software, or other operational or usage environment to another"* [14]. Web service deployment enables this portability by containerization and orchestration software to provide a platform abstracting hardware and execution environments, but also constraining the deployment to the used abstraction technologies [198]. Vendor lock-in to a specific Deployment Provider or technology reduces the portability of web services while enabling easier reuse of functionality compared to a Deployment Provider-independent implementation [193].

Requirements traceability is a decades-old research field that builds on top of requirements engineering concerning *"the ability to describe and follow the life of a requirement, in both a forwards and backwards direction"* [116]. According to Gotel and Finkelstein, requirements traceability can be achieved using basic techniques, such as documenting and referencing requirements; or using automated tool supports such as general-purpose and special-purpose tools, workbenches, or requirement traceability environments combining several tools [116]. Gotel and Finkelstein differentiate between pre-requirements-specification traceability concerning requirement production before the requirement was specified and post-requirements-specification traceability concerning the life of the requirement [116]. Requirements traceability is seen as an important tool to document the system development, and improve the software quality and development process documentation [233].

Aizenbud-Reshef et al. extend the definition of requirements traceability beyond requirements *"as any relationship that exists between artifacts involved in the software-engineering life cycle"* [17]. This includes explicitly defined links or mappings, links inferred from existing information, and statistically inferred links computed based on the historical changes to the involved artifacts [17]. Mäder et al. have demonstrated different approaches to traceability: a UML-based traceability information model [186] and mechanisms that update tracing information based on changes in UML system design models [185]. Further research by Mäder et al. defines mechanisms to model and query traceability information [183] and demonstrates the usefulness of requirements traceability, showing that traceability improves the speed and correctness of software maintenance tasks [184]. Research by De Lucia et al. shows that integrating software engineering traceability information in the development of artifacts improves the quality of code identifiers and comments [80]. The research presents tools to show traceability information when writing source code and suggests related artifacts to increase the consistency between them [80].

Traceability research by Aizenbud-Reshef et al. [17], Mäder and Egyed [184], and De Lucia et al. [80] showcase the benefits of bringing requirements and other software engineering artifacts closer to the source code and using tools to track these artifacts in the ongoing evolution of software systems. We believe that integrating a subset of non-functional web service-related concerns into the development of web services can benefit the development, deployment, dynamic runtime behavior of web services. Web service deployment evolution can benefit from these annotations by incorporating the gained context to improve the deployment and fulfillment of non-functional requirements at runtime. Section 5.1 demonstrates use cases for these annotations, by identifying opportunities in FaaS-based deployments, observability, and Web of Things-based deployment. Section 5.2 introduces the notion of web service metadata annotations incorporating non-functional requirements, constraints, and other information in web service development.

5.1 Web Service Deployment Evolution Domains

Web service deployment evolution investigates changes in the deployment structure due to new requirements, constraints, and execution environment changes. Automating the development of distributed systems using continuous integration and continuous delivery enables web service developers to continuously release and provide value to the stakeholders [308]. Deployment pipelines instantiate the continuous integration process by defining a set of steps triggered by changes to application artifacts that test, integrate, and package the software, define configuration options and execution environments, and release it [135]. Using these techniques enables continuous software engineering [54, 203] as a development methodology that sees software development *"as a way of experimenting and testing what the customer needs"* [54]. This approach to meet the customers' needs requires great flexibility, automation, and a suiting infrastructure supported by tools and workflows [98].

Developer Operations (DevOps) is a set of cultural transformations of adopting and promoting continuous integration and agile development based on open sharing of information enabled by advancements from software development, operation tools, and agile methodologies [201]. DevOps methodologies should enable fast and continuous feedback, strengthening iteration speed and confidence across all software engineering activities [155]. A central enabler of the DevOps transformations are approaches such as infrastructure as code, applying software development mechanisms such as version control, tests, and continuous integration to infrastructure configurations in source code documents [193]. Infrastructure as code uses versioned definition files that can be automatically tested and, e.g., describe the desired usage and configuration of computing resources of dynamic infrastructure platforms [193]. These mechanisms enable consistent and repeatable routines for configuring, rebuilding, and scaling infrastructure resulting in changeable systems that can easily be created, replaced, resized, and destroyed [193].

Containerization technologies, such as Docker, further enable standard packaging and distribution of software components, using layered images with no virtualization overhead compared to virtual machines and being configured and built using infrastructure as code-like configuration files [24]. Kubernetes, a successor to the Google Borg and Omega container-management systems, leverages infrastructure as code and container-based deployment mechanisms to observe and scale clusters of containerized software components [66]. Kubernetes has become a standard for managing scalable, container-based clusters across a wide variety of Infrastructure as a Service (IaaS) providers [65].

Infrastructure as code in, containerization, and container orchestration provide powerful tools to promote DevOps and enables continuous integration for cloud deployments. Infrastructure configuration files can be generated based on exist-

ing architecture models as demonstrated by Simon et al. using a metamodel-based technique of modeling web services [259]. Annotations and metadata as described in Section 5.2 can provide additional context to the generation of infrastructure specifications and can move infrastructure as code even closer to the implementation of the web service. Resource constraints such as memory, computation power, and sustainability-related energy constraints and goals can be annotated on a web service or Handler level to be incorporated in web service deployment structures. In addition, emerging serverless and Function as a Service (FaaS) deployments require even more fine-grained information to deploy web service functions.

Section 5.1.1 details challenges in modern cloud-based deployments that motivate the usage of service metadata annotations in these cloud-based deployments. Annotation mechanisms can also enable use cases beyond orchestration and infrastructure provisioning by improving the observability of web services and, therefore, providing the required feedback for DevOps-based methodologies. Section 5.1.2 provides an overview of current observability techniques and motivates how service annotations and additional metadata-based context can improve web service deployment evolution. Section 5.1.3 subsequently motivates how web service metadata annotations can enrich the deployment beyond cloud-based applications with a focus on the Web of Things and fog computing applications.

5.1.1 Cloud-Based Deployments

The Cloud Native Computing Foundation (CNCF) defines cloud native technologies as means that *"empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds"* [273]. The CNCF, a Linux Foundation project, especially highlights *"Containers, service meshes, microservices, immutable infrastructure, and declarative APIs"* [273] as essential technologies and methodologies to achieve cloud native systems. As noted by the definitions, scalable systems, such as microservices, and smaller deployment units, such as functions in Function as a Service (FaaS) deployments, play a critical role in cloud native systems. Nadareishvili et al. define a microservice as *"an independently deployable component of bounded scope"* [194] that can be used to develop highly automated and evolvable systems [194]. According to Nadareishvili et al., having empowering methodologies such as DevOps and agile development as well as tools such as containerization and orchestration tools available is essential to producing good microservice system behavior [194]. There are different options available to host microservices, ranging from physical machines, virtual machines, containers, platform as a service (PaaS) provides, and serverless offerings [198].

Serverless computing, a term meant to express the absence of manually managing servers or virtual machines, is a deployment and automatic scaling concept using SaaS, FaaS, and Backend as a Service (BaaS) offerings [145]. Cloud functions exe-

cutable on FaaS platforms are complemented by BaaS offering such as object storage, databases, publisher-subscriber mechanisms, big-data queries, and more, requiring no manual provision of computing instances [145]. In contrast to serverful computing, serverless decouples the computation from storage while executing the code without the need to allocate resources as web service developers and only paying in proportionally to the resources used instead of resource allocations [145]. Cloud functions are stateless event-driven resources offering time-constrained executions of code triggered by ephemeral messages that are removed after execution and only leave side effects on any BaaS or external platforms the cloud function interacted with [112]. In the book *Software Architecture Patterns for Serverless Systems*, Gilbert argues that serverless computing is a giant leap in the direction of event-driven architectures (EDAs), mitigating the ever-growing trend of *microservice death stars*, representing distributed systems relying on over-connected microservices [112]. While FaaS offerings provide several advantages, they also introduce limitations such as limited control of resources provisioned to the cloud functions[16], how long cloud functions are allowed to run[17], and possible limitations that result from the stateless nature of cloud functions [198]. Additional challenges include the startup time of cloud functions, which is addressed by intelligently keeping functions warm, which starts the cloud function runtime before a request arrives [198]. Cloud functions also provide unique opportunities for FaaS providers, further isolating and mitigating the security-related impact of individual requests and additionally improving the cost-performance advantages using specialized hardware and execution environments for cloud functions [145].

Before serverless computing, the mapping of software components such as monolithic applications and microservices has been a mapping of a single software component to a hardware instance, virtual machine, or single container [198]. Newman notes that, while a microservice can be deployed to a single cloud function, this requires a dispatch from the single entry point to the different functionalities, and contradicts the single purpose stateless event-based and scalable nature of serverless computing [198]. Instead of deploying a microservice to a cloud function, the book

[16]AWS Lambda functions currently offer different memory configurations ranging from 128 MB to 10,240 MB: https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html#configuration-memory-console. The available computing power is measured virtual in CPUs (vCPUs) that correlate with the memory configuration, currently offering six vCPUs at the maximum memory configuration: https://aws.amazon.com/about-aws/whats-new/2020/12/aws-lambda-supports-10gb-memory-6-vcpu-cores-lambda-functions/. Similarly, Google Cloud functions currently offer different Cloud Function Configurations from 128MB and approximately a 200MHz CPU to 8192MB and approximately a 4.8 GHz CPU: https://cloud.google.com/functions/pricing.

[17]Google cloud functions currently have configurable timeouts of one to eight minutes: https://cloud.google.com/functions/docs/concepts/exec#timeout. AWS Lambda functions currently have a maximum timeout of 15 minutes: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html.

proposes the concept of further decomposing microservices, breaking the boundary of a single deployable unit and transforming it into logical concepts of similar functionality that can be deployed individually [198]. This evolution of the microservice architectural style to a logical collection of functionality, that can be individually deployed, requires special tools and mechanisms to develop and identify individually deployable functionality in web services.

5.1.2 Observability

While traditional methods of software engineering followed sequential and activity-centered life cycle models such as the waterfall model and the V-model, software projects shifted to use iterative and agile life cycle models to develop software systems [62, 310]. At the same time, modern open-source development projects such as the Linux kernel development described by Raymond in *The Cathedral & The Bazaar* [235] showcase that software does not inherently require careful planning and can be developed using *"the naively simple strategy of releasing every week and getting feedback from hundreds of users within days, creating a sort of rapid Darwinian selection on the mutations introduced by developers"* [235]. Frequent releases, many users, and developers observing the system behavior in its target environment facilitate that problems can be quickly characterized and fixed in subsequent releases [235]. Technical insights provided by source-aware users and actionable insights from bug reports enable developers to quickly fix possible bugs and avoid failures in future versions [235]. The more users, developers, and beta-testers run and observe the system, the higher the number of bugs found in the software system [59].

In addition to users and developers manually observing the system behavior, different mechanisms and techniques can be used to automatically and continuously gather insights and evolve software systems. Monitoring systems to extract the needed actionable insights based on information collected about the monitored software system is an established and decades-old technique [264]. The three pillars of observability in web services, namely logs, metrics, and distributed traces, are powerful tools and indicators to continuously observe distributed systems [268]. Logs are collections of records generated in response to an event that includes information such as date, time, and context to describe the event [9, 76]. Metrics represent measurable data that can be aggregated and interpreted using mathematical models [268]. Traces are collections of event logs and metadata identified by a trace context associated with a request as it propagates through the distributed system [212]. Distributed traces enable further insights by being request-centered, attaching metadata to requests in a distributed system, allowing instrumentation to record events, and associating events with the request [258]. Services can be instrumented to react to specific events and can aggregate these events to determine and debug the correctness and performance of software systems [32, 63].

Sridharan describes observability of software systems as a superset of monitoring that *"provides not only high-level overviews of the system's health but also highly granular insights into the implicit failure modes of the system"* [268]. The term observability originates from early control and information systems and describes the concept of determining an unmeasurable state by observing measurable system behavior [148]. Observability goes beyond monitoring by designing systems to expect failure and produce fast feedback loops by observing key performance indicators in production environments [268]. Karumuri et al. structure system observability into two activities: data management and analytics [150]. Data management includes the instruments that emit information as well as the storage and processing of this information; the analytics activity consists of the monitoring and analysis of the data that lead to decisions and responses to the observed behavior [150].

Niedermaier et al. provide a qualitative study based on semi-structured interviews of software professionals detailing challenges, requirements, and corresponding solutions to observe and monitor distributed systems [199]. The study identifies several challenges: The complexity and heterogeneity of modern distributed systems; the flood of data produced; the lack of expertise, time, and resources; unclear non-functional requirements; and the resulting reactive approach of only implementing monitoring after failures [199]. Niedermaier et al. highlight several requirements and solutions to address these challenges, including the requirements to provide a holistic approach for context propagation across components as offered by distributed tracing and adding metadata to metrics and logs [199]. A key requirement stated is the inclusion of observability functionality from the start and the use of automated and smart approaches to analyze the observed information and provide valuable insights about the system behavior [199]. Tools and all-in-one solutions play a crucial role in enabling this level of observability, mainly focusing on scalability, extensibility, portability, and security [199]. Based on these insights, artifacts enabling web service deployment evolution need to address the observability-related expectations of software developers. Observability functionality must be smart, context-aware, and easily usable to provide valuable insights into the evolution-related challenges of web services. Service annotations as highlighted in Section 5.2 can provide additional context to allow smart monitoring decisions and anomaly detection based on annotated non-functional requirements and other constraints. The artifacts designed in Part III embrace these observability-related requirements to support web service evolution.

5.1.3 Web of Things

The Internet of Things domain concerns applications of ubiquitous connections between devices enabled by innovations in hardware, software, and communication technology [18, 262]. Al-Fuqaha et al. and Sisinni et al. provide extensive surveys on

the enabling technologies and emerging challenges, highlighting the need for innovations in hardware, software, and architectures to enable IoT applications [18, 262]. As detailed in Section 2.2, fog computing is an architecture creating a *"layered model for enabling ubiquitous access to a shared continuum of scalable computing resources"* [139]. Fog computing extends the static nature of edge computing with dynamic and intelligent reconfiguration benefiting from decoupling hardware and software functions [139]. Fog nodes, managed by a service orchestration layer, communicating with each other, devices at the edge of the network, and cloud services build the essential building blocks of a fog-based architecture [52, 53]. This orchestration is performed based on distributed policies building on different, often conflicting non-functional concerns that need to be collected, stored, managed, and resolved [52]. Henze et al. provide additional formalizations focusing on the dynamic and scalable aspects of fog-based architectures, defining concepts such as fog visibility, fog horizons, and fog reachability using the set-theory-based approach to reason about possible interactions between fog nodes [128, 129]. Andreas Seitz formalizes fog computing approaches by presenting the Fogxy architectural style, enabling the application of fog computing to a wide variety of application domains [255]. Service annotations such as introduced in Section 5.2.1 can enable these orchestration mechanisms to identify and collect non-functional concerns of participating web services. Part IV introduces several case studies in the IoT and fog computing domain, showcasing the application of fog computing in these domains, and validates the designed artifacts in Part III.

The Web of Things leverages the idea of interconnecting IoT devices to offer web service-based access to resource-constrained IoT devices using web-based API types and protocols like HTTP [86]. More capable IoT devices enable the use usage of the entire OSI protocol stack in resource-constrained environments and lower the interoperability challenges of IoT-based communication protocols and technologies by using the same web API types used to communicate with web services [86, 277]. The dissertations by Guinard and Trifa and publications with their colleagues address the Web of Things architecture composing smart devices in composite web-based software systems using architectural styles such as RESTful web services [121, 281]. Guinard and Trifa present different approaches to turning connected devices into RESTful web services using embedded computing to offer the web API on the device or using smart gateways abstracting the actual communication protocol for resource-constrained devices [122]. Guinard et al. go beyond modeling web service protocols as transport mechanisms and elevate IoT devices by offering services adhering to the REST constraints such as the uniform interface using an HTTP-based RESTful implementation [124]. The smart gateway for resource-constrained devices acts as a proxy mapping the incoming requests to proprietary APIs of the smart device and establishes a connection to the device's lower-level communication pro-

tocol [124]. Guinard et al. propose semantic annotations as described in Section 5.2 to find and describe smart devices in the Web of Things [123]. These annotations can semantically annotate HTML files describing the RESTful APIs and can be indexed by search engines to localize smart devices [123].

Beyond the search and discovery optimizations for smart IoT devices, annotations and additional metadata added to smart devices and web services can facilitate automated and constraint-solving deployment mechanisms for WoT applications. Hur et al. use semantic descriptions to generate service descriptions and deployment based on these annotations across different IoT platforms [136, 137]. The research by Hur et al. applies IoT deployment mechanisms from IoT platforms to WoT enabled by a heterogeneous semantic service description addressing interoperability issues between different IoT platforms [137]. The semantic service description consists of properties defining static attributes of physical objects, capabilities describing dynamic data provided by physical objects, and service profiles specifying the configuration of a physical object interacting with specific IoT platforms [136, 137]. The process of generating a service deployment consists of extracting the metadata from physical devices and platforms, transforming it to a platform-specific service description, and deploying the software to the IoT platforms [137]. The process and approach by Hur et al. demonstrate a sophisticated approach of using annotations and metadata to create static deployment configurations for WoT applications. While the deployment evolution enabling annotations focus on the physical devices and IoT platforms, the insights and procedure provide valuable insights into applying annotation-based approaches to enhance the static and dynamic behavior of deployment structures of evolvable WoT web services.

Vögler et al. present DIANE and LEONORE as tools to enable dynamic IoT application deployment, triggered by changes to the applications business logic, customer request patterns, and changes in the physical infrastructure [295]. LEONORE is *"a service oriented infrastructure and toolset for provisioning application components on edge devices in large-scale IoT deployments"* [294]. LEONORE prepares installable packages catered to the target environment and allows a push and pull-based provisioning of devices [294]. Optimizing the elastic deployment of these IoT applications is based on dynamic information received from application rules, infrastructure rules, and policies to optimize the deployment based on changing internal and external factors [296]. Application rules, such as response time, and infrastructure rules, such as CPU usage, are monitored, and the application deployment is constantly optimized to scale the application within its hardware constraints [296]. Instead of using optimized installation packages, research by Islam et al. use Docker-based software components to IoT-based deployments, taking advantage of advancements in computation power and memory, allowing a containerized deployment of WoT services [140]. Islam et al. use containerized deployment approaches

to dynamically detect IoT gateways and suitable software components and perform resource matching and deployment to the IoT environments [141]. Research by Vögler et al. and Islam et al. demonstrates the possibilities of dynamically adjusting IoT-based applications to changing internal and external factors contributing to dynamic changes in the deployment structure of web services. All approaches rely on domain-specific and service-level information about the software components to distribute software interacting with connected devices. The artifacts designed in Part III contribute to the dynamic web service deployment evolution by providing extensible and domain-specific annotation, collection, and interpretation mechanisms that provide domain-specific information to static and dynamic IoT deployment mechanisms.

5.2 Web Service Metadata Annotations

An important aspect of specifying and incorporating non-functional requirements in a software system is to design a suiting software architecture and use tools, patterns, and data structures that benefit the fulfillment of these requirements [135]. Continuous software engineering mechanisms and tools such as tests, profiling tools, deployment mechanisms, and monitoring tools must incorporate and validate these requirements [135]. Non-functional concerns, other constraints, and application domain context connected with web service functionality also play a crucial role in better understanding, deploying, and consuming web services. Annotation mechanisms enable developers to express these non-functional requirements and other semantic information as part of the software engineering artifacts and make them available to tools. The annotation model defined in Section 5.2.1 describes annotated information to enable web service deployment evolution-related domains detailed in Section 5.1. Section 5.2.2 provides an overview of different web service metadata annotation domains, enriching web service-related software systems with information about non-functional concerns and semantic information.

5.2.1 Web Service Metadata Annotation Model

This section presents the web service metadata annotation model. The model details a high-level metadata annotation model concerning different levels of the web service interface metamodel (Figure 3.1, page 48) stereotypes.

> **Knowledge Question 5:**
> How can artifacts collect requirements, constraints, and application domain and deployment environment-specific information to address web service deployment evolution?

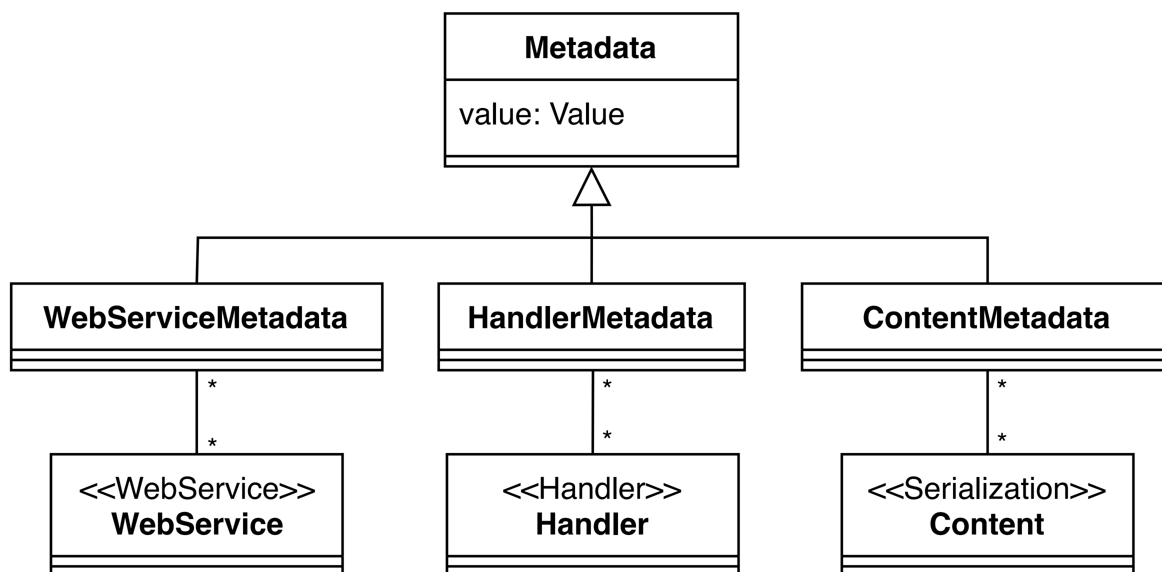


Figure 5.1: The web service metadata annotation model presents a hierarchy of metadata annotations that are associated with the web service interface type-independent stereotypes presented in the web service interface metamodel (Figure 3.1, page 48). The metadata type can be extended for different non-functional requirements, constraints, and application domains and can be specialized for different web service stereotypes. (UML Class Diagram)

The model address Knowledge Question 5. We choose the term metadata as a collective term to describe the possible annotations for the web service design mechanisms. Based on the metadata definition in the WS-Policy specification [55] presented in Section 5.2.2, we interpret metadata as a collection of machine- and human-readable aspects refining functional aspects of web service elements. The annotation-based approach is derived from the declarative component configuration pattern defined by Buschmann et al. as an approach that *"addresses how the resource and infrastructure needs of a component can be passed to its hosting infrastructure"* [67]. The pattern is instantiated and often referred to as *Annotations*, a feature present in different programming languages to specify metadata [67].

The presented model does not contain complex relationships or properties of the modeled stereotypes. It provides a simple foundation to model and express possible metadata-related annotations for the domains detailed in Section 5.1. While web service metadata annotation domains presented in Section 5.2.2 showcase annotation models specific for non-functional concerns and semantic information, the goal of the model and the artifacts designed in Part III is to provide an annotation mechanism to benefit web service evolution-related challenges. The metadata annotation model incorporated aims to provide extensible and web service interface type-independent annotation mechanisms. The concrete mechanisms defined in Part III are not constrained to semantic annotations or non-functional concerns,

but can be used to express and interpret these aspects to tackle web service evolution. The metadata annotation types are specialized for different application domains. Part IV explores how the presented extensible metadata annotation model can be used to address challenges in the areas of web service observability, service partitioning, resource usage, and the Web of Things.

As shown in Figure 5.1, the metadata type contains a value defining the content of the metadata annotation. We differentiate between three different metadata specializations: web service metadata, Handler metadata, and serialization metadata. The metadata annotations are associated with instantiations of the web service interface profile. Web service metadata is refining the functionality of a web service instance annotated with the web service stereotype. Similarly, Handler metadata is refining the functionality of a Handler instance annotated with the Handler stereotype. Content metadata annotates the content of requests and responses conforming to a serialization stereotype. Each metadata annotation is in a many-to-many relationship with the annotated entity. To reduce the amount of possible metadata and reduce complexity, we deliberately do not define metadata on the request or response instantiations of a web service as any aspects refining Handlers and serializations can also be used to refine request and response mechanisms.

Artifacts designed in Part III need to incorporate mechanisms to annotate elements of web service artifacts with metadata information that can be parsed and provided to deployment evolution-related tools. The collection mechanism needs to allow a structured export of the metadata information with a variable level of detail based on the deployment mechanism and application domain. The metadata annotation instantiations can include non-functional concerns such as response times or memory metadata of a Handler, hardware constraints of a web service, and additional context for content exchanged with web services.

5.2.2 Web Service Metadata Annotation Domains

This section presents five different web service metadata annotation-related domains that use metadata annotations to improve web service development and web service evolution. These domains range from model-based approaches to annotations in service specification languages and web pages in the semantic web.

Model-Based Web Service Annotations

Annotating non-functional requirements in model-driven architectures is achieved using UML profiles such as the *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification* (QoS) [8] and the *UML Profile for Schedulability, Performance, and Time Specification* (SPT) [6] adopted by the Object Management Group (OMG) [40].

The QoS UML profile offers an extensible mechanism to associate quality of service and fault-tolerance properties with model elements [8]. The annotation process consists of three steps: Defining the QoS characteristics, defining quality models by instantiating the QoS characteristics template classes, and annotating the application domain UML models with QoS constraints and QoS values [8, 40]. The SPT profile provides stereotypes and attributes that can be used to stereotype elements or annotate model elements, and assign values to these properties using a comment-based notation [6, 40]. The QoS and SPT profiles provide different tradeoffs between flexibility and convenience: The QoS profile provides more extension points, but is more complex to apply, while the SPT profile is less flexible, but involves fewer steps to apply non-functional concerns to UML models' elements [40].

In contrast to the OMG-adopted UML profiles, Ortiz and Hernandez present a metamodel using UML profiles and UML stereotypes, such as the *Extra-Functional Property* stereotype, to express web service-related non-functional concerns in UML models [208]. The stereotype annotations can also be exported to generate WS-Policy documents [55] incorporated in the web service specifications ecosystem [208]. Building on top of this metamodel, Ortiz and Hernandez present an aspect-oriented approach to specifying the enforcement that is subsequently used to generate aspect-oriented Java code skeletons [207].

While the artifacts designed in Part III do not follow a purely model-based approach, the model-based approaches highlight different levels of flexibility and extensibility to express non-functional concerns during model-driven development of software systems. As highlighted by Bernardi and Petriu, the balance between flexibility and convenience of expression is a challenge that requires extensible yet straightforward approaches to annotate web service implementations with non-functional requirements and other constraints [40].

Composition of Non-Functional Concerns in Web Services

As part of the dissertation project on *Composing Non-Functional Concerns in Web Services*, Schmeling et al. published several papers and articles proposing methodologies to specify, enforce, and compose non-functional concerns (NFCs) such as security and performance in web services [245, 246, 247, 248]. The metamodel based on definitions by Rosa et al. [242], defines non-functional attributes as the quality or characteristics of non-functional concerns that are affected by non-functional actions (NFAs), representing aspects that affect non-functional attributes [242, 245].

Schmeling defines three composition types: composition of functional concerns with NFAs, enriching functional code with non-functional concerns, the composition of superimposing NFAs, where multiple NFAs are enforced at once, and the composition of composite NFAs where NFAs are composed similar to functional composition in business processes [245]. Schmeling presents the *NFComp* ap-

proach enabling non-functional requirements composition consisting of a development process, a metamodel for non-functional concerns, a graphical notation composing non-functional concerns, modeling editors, code generators, and tools to validate models at design time [245]. It differentiates between the specification and realization of non-functional concerns in web services as well as the perspectives enabling different levels of insight into the internals of a service [33, 246, 247].

The article *A survey on non-functional concerns in web services* provides an extensive literature research clustering web service and non-functional concern composition, highlighting the high number of publications that investigate the specification of non-functional concerns while pointing out that there is less focus on web services-related non-functional concerns [246]. Schmeling et al. define several requirements to specify non-functional requirements, such as an action that can contribute to the realization of the requirement, that defines a subject of the non-functional requirement, and the ability to define the order of different non-functional actions for a use case [247]. The enforcement of non-functional concerns is performed using the interceptor pattern using message-oriented middleware or aspect-oriented programming using language features, such as aspects to crosscut behavior using pointcuts [246, 247].

Schmeling et al. also define several requirements for the enforcement of non-functional requirements: including the enforcement separate from functional concerns, the ability to associate one NFC with multiple functional concerns, and the ability to enforce NFCs even if the enforcement and functionality are implemented in different execution environments [247]. The research presented provides a model-based development process for specification and realization of NFRs in accordance with the defined requirements, starting from a specification phase with a requirements engineer to a code realization phase in which a service provider generates code to enhance a web service to enforce the defined non-functional concerns [248]. The enforcement of these composed non-functional requirements is performed using a proxy-based approach that is deployed in front of web services, executing the middleware services enforcing the NFAs, and finally delegating the handled request to the original web service [248].

The research by Schmeling et al. provides in-depth insights on the composition of non-functional requirements and the effect of enforcing and specifying them on composing web services. The proxy-based approach allows service provides to retroactively add non-functional actions to web services to enforce multiple composed non-functional concerns. While our proposed approach is mainly concerned with annotating and using non-functional concerns and semantic information to improve web service deployment evolution-related changes, the differentiation between non-functional requirement, non-functional concern, and non-functional action provides input on modeling metadata for web services defined in Section 5.2.1.

WS-* Ecosystem Metadata Annotations

The the web service specification (WS-*) ecosystem of standards incorporates several specifications and extensions around XML-based standards originating from the WSDL [75, 192], the SOAP [311] and UDDI [37].

The Web Services Policy (WS-Policy) specification defines service metadata as *"an expression of the visible aspects of a Web service, and consists of a mixture of machine- and human-readable languages"* [55]. The specification differentates between metadata describing the payload formats of web services expressed by the XML Schema Definition (XSD) language [301], metadata describing the web service interface expressed by the WSDL [75], and metadata describing the capabilities and requirements of web services such as the WS-Policy XML-based DSL [55].

The WS-* ecosystem includes many extensions that can be used to implement and extend non-functional concerns, e.g., WS-Security [167], as an extension to SOAP enables message content integrity and confidentiality. WS-Trust specifies methods to *"issuing, renewing, and validating security tokens"* [169] and establishing trust relationships between web services. WS-Trust also defines predefined WSDL endpoints, ports in WSDL 1.1, that can be instantiated for different WSDL bindings [169]. WS-SecureConversation builds on top of WS-Security and WS-Trust, further refining how security contexts and trust between web services are established [168]. Machine-readable metadata, such as the WS-Policy domain-specific language, can be used to enable tooling describing security and reliability non-functional concerns [55].

Providing metadata related to non-functional concerns at runtime to be monitored, collected, and interpreted can provide web service consumers and service compositions a holistic overview of the service offered [261]. Therefore, the Web Services Metadata Exchange (WS-MetadataExchange) standard defines expressing metadata as HTTP resources and how metadata can be retrieved from web services using request-response interactions [303].

The WS-Policy and related standards enable the expression and enforcement of non-functional concerns in web services developed using the WS-* ecosystem primarily focused on SOAP-based interfaces. The definition of service metadata provides essential insights for the web service interface type-independent metadata annotation model presented in Section 5.2.1.

Semantic Web-Based Metadata Annotations

The idea of the semantic web is, according to Berners-Lee et al., that it *"will enable machines to COMPREHEND semantic documents and data, not human speech and writings"* [43]. The semantic web concept primarily relates to web pages where information is annotated and connected using the standards like the Resource Description Framework (RDF) [166] and the Web Ontology Language (OWL) [10].

Annotating semantic web service information, going beyond non-functional characteristics in the WS-* ecosystem, is achieved using Semantic Annotations for WSDL and XML Schema (SAWSDL) [94]. The SAWSDL *"defines how to add semantic annotations to various parts of a WSDL document such as input and output message structures, interfaces and operations"* [94]. SAWSDL uses extension mechanisms provided in the WSDL [75] and XML Schema [301] specifications to annotate schema types and provide mappings to ontologies defined by semantic models [94]. Semantic models contain concepts that are identified or created using XML attributes as defined by the SAWSDL standard [94].

Before SAWSDL existed, Patil et al. presented the METEOR-S Web Service Annotation Framework (MWSAF) as a specification to enhance service discovery and composition by mapping WSDL documents with ontologies [214]. MWSAF proposes abstractions to create SchemaGraph representations from WSDL documents and ontologies, provide a matching algorithm suggesting mappings to developers, and write these mappings back in WSDL documents using XML based annotations [214]. Peng and Bai list several semantic annotation mechanisms for RESTful APIs and present the Semantic RESource Tagging (SemREST) method to annotate OpenAPI specifications with semantic tags [215].

The mechanisms used in SAWSDL, MWSAF, and SemREST demonstrate annotation-based additions of context to web service interfaces definition languages. While the focus on the web service interface provides additional semantic context for web service composition and discovery, mechanisms closer to the functionality of the source code are more suited when providing functionality-specific context annotations for the deployment-related domains listed in Section 5.1.

Energy-Efficient Computing

As discussed in this section, memory, computation power, and other performance and usability constraints such as timeouts can be inferred from non-functional requirement-based annotations. These insights can be applied beyond FaaS related deployments to IaaS, PaaS, or container-based deployment and orchestration mechanisms. While performance and responsiveness are often considered essential, computer science research, hosting providers, and other participating stakeholders must also consider sustainability-related aspects of the tools and methods. Consolidating computation power in energy-efficient ways, using energy-efficient equipment, and dynamically allocating resources and, therefore, reducing idle times are some of many techniques to reduce the power usage and carbon footprint in regards to cloud deployments [234].

One key challenge is analyzing, predicting, and monitoring software to identify high energy usage and designing tools to negotiate lower quality of service (QoS) modes and subsequently implement alternatives to reduce energy usage [234]. In

their research, Bartalos et al. highlight the challenges of predicting power consumption of black-box web services due to the short execution times of web service functionality, the dependence on the current state of the web service, as well as the correlation of power consumption and the input provided to the web service [30].

One source of information can be web service developers as they have concrete insights into the algorithmic complexity and correlations of input to output. Developers can make this information available to sustainability-aware resource allocation and configuration mechanisms. Based on this principle, Vitali proposes mechanisms to enable sustainable cloud-native applications and their composition by leveraging different requirements, constraints, and possible sustainability-related execution alternatives for web services [292]. Vitali presents a four-step sustainable application design process correlating to four levels of sustainability awareness, with level zero being the current state-of-the-art microservice-based cloud native application [292]. Step one encompasses adding sustainability-aware information as well as non-functional requirements to web services that providers can use to optimize the deployment structure [292]. Levels two and three address the composition of microservices and the enrichment of business processes encompassing the microservices with different execution modalities for normal execution, high-performance, and low-power modes impacting the energy consumption of the executed business process [292].

Research by Vitali highlights how metadata annotations can enrich solutions to address the cloud native web service deployment evolution-related challenges. The treatment design (Part III) and treatment validation (Part IV) demonstrates the applicability of the designed artifacts and annotation mechanisms to address web service deployment evolution.

Part III
Treatment Design

TREATMENT design describes the design and instantiation of artifacts so they can be applied to treat problems investigated in the Part II [309]. We present the Apodini ecosystem, which encompasses a collection of domain-specific languages, frameworks, and tools that address web service evolvability. We base the treatment design on the methodology described in Section 1.2, combining the software engineering approach defined in ISO/IEC/IEEE 12207 [15] and design science methodologies defined by Wieringa [309] to describe the artifact design process. The software engineering approach is refined using methodologies, best practices, and documentation approaches presented by Bruegge and Dutoit [62].

This part documents the treatment design of artifacts using system design and object design documentation techniques. Chapter 6 documents the system design of the Apodini system using an adapted version of the System Design Document (SDD) presented by Bruegge and Dutoit [62]. Chapter 7 realizes the system architecture from the previous chapter and defines class models, type signatures, and other implementation-related models describing the internal behavior of the system [62].

Chapter 6

System Design

Bruegge and Dutoit state that *"During system design, we identify design goals, decompose the system into subsystems, and refine the subsystem decomposition"* [62]. This activity is documented in the System Design Document (SDD), serving as a reference for design goals, architecture-level decisions, and defining interfaces between more extensive subsystems refined in later software engineering activities [62]. Defined by Bruegge and Dutoit as a model to help the initial understanding of the system architecture, *"The top-level design represents the initial decomposition of the system into subsystems"* [62]. Figure 6.1 details the top-level design of the Apodini ecosystem.

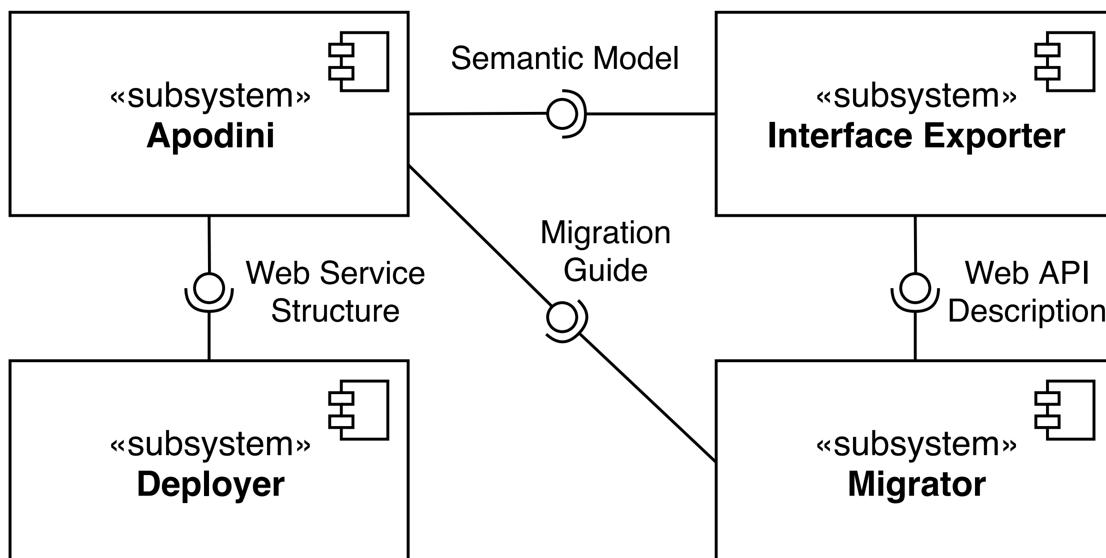


Figure 6.1: Top level design of the Apodini ecosystem. The simplified ecosystem consists of four main subsystems: Apodini itself, the Interface Exporter subsystem, the Deployer subsystem, and the Migrator subsystem. The model displays simplified high-level interfaces to provide an initial understanding of the system architecture. (UML Component Diagram)

Section 1.1 introduces several technical research goals, expressing goals to design or improve existing artifacts. These artifacts aim to improve the process of designing evolvable web services. Apodini addresses aspects of web service interface evolution, web service API evolution, and web service deployment evolution. The design problems introduced in Section 1.1.1 refine these technical research goals to highlight problems addressing stakeholder goals that contribute to their fulfillment [309]. The design problems can be mapped to the different subsystems introduced in the top level design of the Apodini ecosystem. Design Problem 1 is addressed by the Apodini and Interface Exporter subsystems:

> **Design Problem 1:**
> Develop all aspects of web services in a web service interface type, web API, middleware, and protocol-independent description so that web service developers can support different web service interface types and web API types without rearchitecting web services.

The **Apodini** subsystem enables web service interface type-agnostic web service development. As detailed in the Object Design Chapter, Apodini uses an internal domain-specific language to express the interface, functionality, and metadata annotations in a single representation that can be parsed and interpreted by the other subsystems. The concept of the Apodini internal DSL was first described in *Apodini: An Internal Domain Specific Language to Design Web Services* [249]. A semantic model is a domain model representing an in-memory model of the domain the DSL describes and is generated out of a syntax tree parsed from the DSL description [107].

The **Interface Exporter** subsystem uses the semantic model to address web service interface evolution. Providing the interface agnostic semantic model to the Interface Exporter subsystem enables an extensible ecosystem of Interface Exporters that interpret and instantiate the semantic model and its functionality for different web service interfaces and web API types. Part IV validates the extensibility and expressiveness of the Apodini DSL and the Interface Exporter subsystem by demonstrating different Interface Exporters such as HTTP, REST, WebSocket, GraphQL, and gRPC Interface Exporters.

> **Design Problem 2:**
> Automatically detect and migrate backward-incompatible changes of web service interfaces to enable web service client stability after modifications to the web service interface.

The **Migrator** subsystem addresses Design Problem 2 and therefore web service API evolution. The Migrator functionality extends Apodini to generate a **migration guide**, expressing the changes and migrations between two versions of an Apodini-based web service. The migration guide further described in Section 7.3 provides

a change classification based in the change patterns described in Section 4.2.1. The Migrator subsystem uses the migration guide and web API type-specific web API description generated by an Interface Exporter to create client libraries. These client libraries guarantee web service client stability by applying the abstract migration steps to the concrete web API description.

> **Design Problem 3:**
> Develop a constraint-based service description so that web service developers can dynamically deploy web services incorporating different deployment structures, processes, and runtime constraints.

The **Deployer** subsystem addresses Design Problem 3 and therefore web service deployment evolution. The Deployer extends Apodini to extract a web service structure from the semantic model, expressing an abstract overview of possible deployment-related metadata information and decomposition constraints. Apodini Deployer provides an extensible architecture to support different application domains such as WoT deployments and web service hosting providers such as FaaS providers to optimize the deployment automatically. By gaining a holistic overview of the web service structure, annotated non-functional requirements, and other constraints, the Deployer subsystem can address challenges in the dynamic and static evolution of **deployment structures** based on different forces affecting deployment-related evolution.

This chapter presents the system design using an adapted version of the System Design Document (SDD) [62]. Section 6.1 defines the design goals of the Apodini ecosystem and its subsystems based on the foundations and insights defined in the previous chapters. The control flow section (Section 6.2) refines the system design with dynamic models, depicting the interactions between the different subsystems. Section 6.3 defines a subsystem decomposition including software components for the different subsystems represented in Figure 6.1.

6.1 Design Goals

This section translates Wieringa's technical research goals and design problems into actionable software engineering criteria using design goals. Design goals or quality attribute considerations identify the qualities that a system should focus on and enable consistent criteria for design decisions involving requirement tradeoffs [31, 62]. Bruegge and Dutoit group the design goal criteria in five groups: *"performance, dependability, cost, maintenance, and end user criteria"* [62]. This section describes different design goals of the Apodini ecosystem of tools, libraries, and methods based on the previous chapters' knowledge context and problem investigation.

Performance Criteria

Performance criteria are related to the speed and space criteria of the system and are grouped into response time, throughput, and memory criteria [62]. The Apodini project is a research project following the design science methodology of Wieringa and has the main goal to design and validate artifacts addressing aspects of web service evolution. The abstract mechanisms needed to support different web service interface types and web API types while dynamically decomposing and redeploying the web services result in tradeoffs between demonstrating these capabilities and focusing on performance.

While performance, throughput, and memory usage should not be neglected, the project's main focus resides on demonstrating evolvability mechanisms and not building a high-performance low-level web service development framework. Therefore we do not explicitly define response time, throughput, and memory criteria, but want to make sure that the Apodini ecosystem provides comparable performance results to other web service development frameworks. We enable reasonable performance, throughput, and memory usage baselines by building on shared and well-proven programming language-specific networking and event management frameworks.

Dependability Criteria

Users depend on software, and software systems depend on execution environment factors which requires software developers to consider hardware failures, software failures, and operational failures caused by human users [267]. Dependability criteria relate to efforts in minimizing system failure grouped into robustness, reliability, availability, fault tolerance, security, and safety criteria [62]. The Apodini subsystem provides a mechanism to express the structure, functionality, and non-functional concerns of a web service in a single description to support web service evolution. As further explained in Chapter 7 and showcased by different related work in Part II, a domain-specific language offers flexible and extensible ways to define and use such a web service description.

Robustness is essential in parsing, interpreting, and further using DSL-based descriptions. The compiler-based support of internal domain-specific languages (Definition 10, page 32) enables robustness by benefiting from features found in integrated development environments such as code completion and compiler-level features such as type checking for strongly-typed languages [107]. Similarly, reliability is also an important criterion when analyzing the web service description. While subsystems like the Interface Exportrer subsystem need to make web API type-based assumptions about the concrete web API, the observed behavior should still reflect the boundaries and functionality expressed in the DSL.

Similar to the performance goals defined above, availability, fault tolerance, security, and safety should not be neglected when designing the system but are not the main focus of the demonstrated treatment design and validation. Nevertheless, the web service development tools must support relevant security mechanisms when validating the artifacts, such as encrypted communication using HTTPS. Faults created in developing a web service should be communicated to the developer at compile or startup time. Faults at runtime should be communicated to the web service client using meaningful error messages and possible hints on how to resolve the issue if it originated from faulty requests.

Cost Criteria

Estimating cost is a complex estimation conducted using cost calculation models to approximate the cost of software systems throughout their lifetime [288]. Cost criteria include design and managerial decisions cost decisions including development cost, deployment cost, upgrade cost, maintenance cost, and administration cost criteria [62]. The development cost and the deployment cost of the Apodini system are constrained by the resources available to the development team at the Technical University of Munich. As the artifacts designed are greenfield research projects, there is no considerable upgrade cost for the designed system. The system's maintenance and administration are performed as part of the research conducted surrounding the Apodini ecosystem. Numerous students contribute to the project as part of courses and theses during their studies and conduction the research project would not have been possible without the time and effort invested by the participating students [34, 35, 69, 83, 126, 159, 160, 162, 202, 229, 230, 270, 304, 317, 318]. Future work presented in Chapter 10 highlights the potential to continue the research beyond the scope of this dissertation, facilitating future research projects.

Maintenance Criteria

Maintenance criteria relate to the difficulty of evolving the software system and include criteria such as extensibility, modifiability, adaptability, portability, readability, and requirements traceability [62]. Extensibility refers to the ability to add functionality to the system, modifiability refers to changes in the functionality, and adaptability relates to adopting new application domains [62]. Portability defines how easy a system can be ported to different platforms, readability to the ability to comprehend the source code [62]. Requirements traceability as the last maintenance criteria has already been explained in Chapter 5. Almost all maintainability criteria are essential criteria for the Apodini ecosystem.

Apodini needs to be extensible to support different Interface Exporters in the Interface Exporter subsystem, different Migrators to migrate different web API types

in the Migrator subsystem, and different deployment strategies in the Deployer subsystem. The Apodini Interface Exporter subsystem needs to support different web service interface types, web API types, and different communication patterns. The Interface Exporters need to build on top of an extensible networking infrastructure as detailed in the subsystem decomposition in Section 6.3. The Migrator subsystem requires extensibility to support different web API types that can be migrated and deliver an extensible infrastructure for the migration guide. Annotation mechanisms in the Apodini DSL and deployment structures defined in Apodini Deployer need to be extensible for different application domains, web service hosting providers, and deployment mechanisms.

Modifiability and adaptability are not as essential as the system's extensibility but have to be considered as part of the research project. The application domain of web service development is the main focus of Apodini. The expressiveness of the Apodini DSL focuses on the support of web service interface evolution. The Apodini DSL and its components need to be modifiable to support new subdomains such as web service API evolution and web service deployment evolution.

The principles of Apodini are programming language-independent and therefore support the portability across different platforms and programming languages. Nevertheless, a reference instantiation of Apodini providing an internal domain-specific language requires web service developers to commit the usage of the programming language within the bounds of its portability across platforms and compatibility with other programming languages. The Apodini DSL should be embedded in familiar features of the embedding general-purpose programming language to benefit the comprehensibility and understandability of the Apodini DSL when developing web services. The readability of Apodini web services and artifacts supporting the web service development, such as migration guides, is an important characteristic. The development of Interface Exporters, Migrators, or Deployment Providers should be supported by documented and readable APIs.

End User Criteria

End user criteria incorporate user-driven criteria not covered in the other groups, including utility and usability criteria [62]. Similar to the readability requirement for the DSL, artifacts, and APIs, the system's usability from the perspective of all stakeholders is vital to support web service evolution. Usability includes clearly defined API extension points and support when executing the tools and incorporating the libraries. The Apodini ecosystem should provide documentation for the artifacts. Help and error messages should guide developers when the system encounters faults.

6.2 Control Flow

Dynamic models focus on the system behavior and can be depicted using UML sequence, activity, or state diagrams [63]. While dynamic models are mostly used in the analysis software engineering activity, this section uses UML sequence diagrams to demonstrate the dynamic interaction within the Apodini ecosystem. We focus on the interactions from a web service developer's perspective involving the web service and instantiations of Apodini Interface Exporters, Migrators, and Deployers.

6.2.1 Interface Exporter

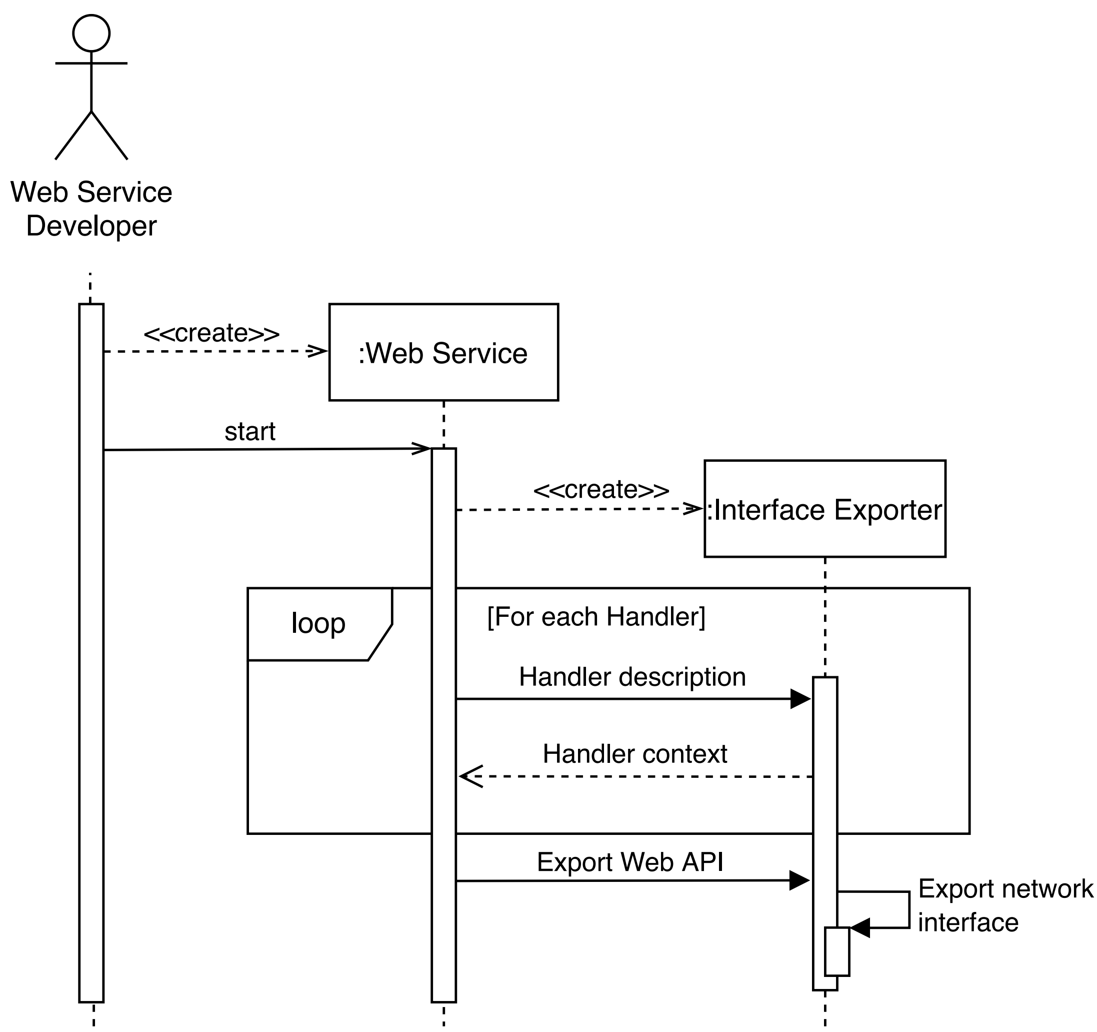


Figure 6.2: Control flow describing the interaction of a web service developer with an Apodini web service and the Apodini Interface Exporter subsystem. The web service developer instantiates the web service that creates an Interface Exporter. The Interface Exporter iterates over all exported Handlers and gathers information to export a network interface. (UML Sequence Diagram)

Figure 6.2 demonstrates the sequence of interactions involved in exporting an abstract Apodini web service description using an Interface Exporter. The dynamic model involves the functionality of the primary Apodini subsystem and the Interface Exporter subsystem.

The web service developer is interacting with an instantiation of an Apodini web service as well as an Interface Exporter. The web service developer first creates a web service, defining the functionality and other concerns in the Apodini DSL. To export a concrete web API, the web service developer creates an Interface Exporter using APIs offered by the Interface Exporter subsystem. When starting the web service, the Interface Exporter uses DSL-based mechanisms further detailed in Section 6.3.2 to pass the Handler descriptions to the Interface Exporter that creates a web API context. This context is then exported on a network interface to offer the concrete web API when the parsing of the web service DSL is complete. The mechanism enables web service interface evolution by providing an extensible interaction between the web service and Interface Exporters interpreting the abstract web service description into concrete web APIs.

6.2.2 Migrator

The Migrator subsystem demonstrated in the top level design in Figure 6.1 is responsible for enabling web service API evolution. Figure 6.3 demonstrates a sequence diagram detailing the interaction between two web service versions and an Apodini Migrator implementation.

When creating a new version of the web service, the web service developer creates a migration guide to enable client application developers to generate stable client facades as detailed in Chapter 4. To generate a migration guide, an implementation using the Apodini Migrator subsystem extracts information from the semantic model information about the web service as described in Section 6.3.1. The semantic model from the new version of the web service is compared to the semantic model of the old version to identify change patterns. These change patterns are documented in the migration guide, and if possible, automatic migrations are already incorporated in the document. The automatically created migration guide is returned to the web service developer in combination with a web API description of the old web service. Even though the automatically generated migration guide contains different best-effort migration steps, the web service developer might need to adapt the migrations and add further migrations based on their domain knowledge. The migration guide and a web API description are provided to the client developer, who can then generate a client facade as discussed in Chapter 4.

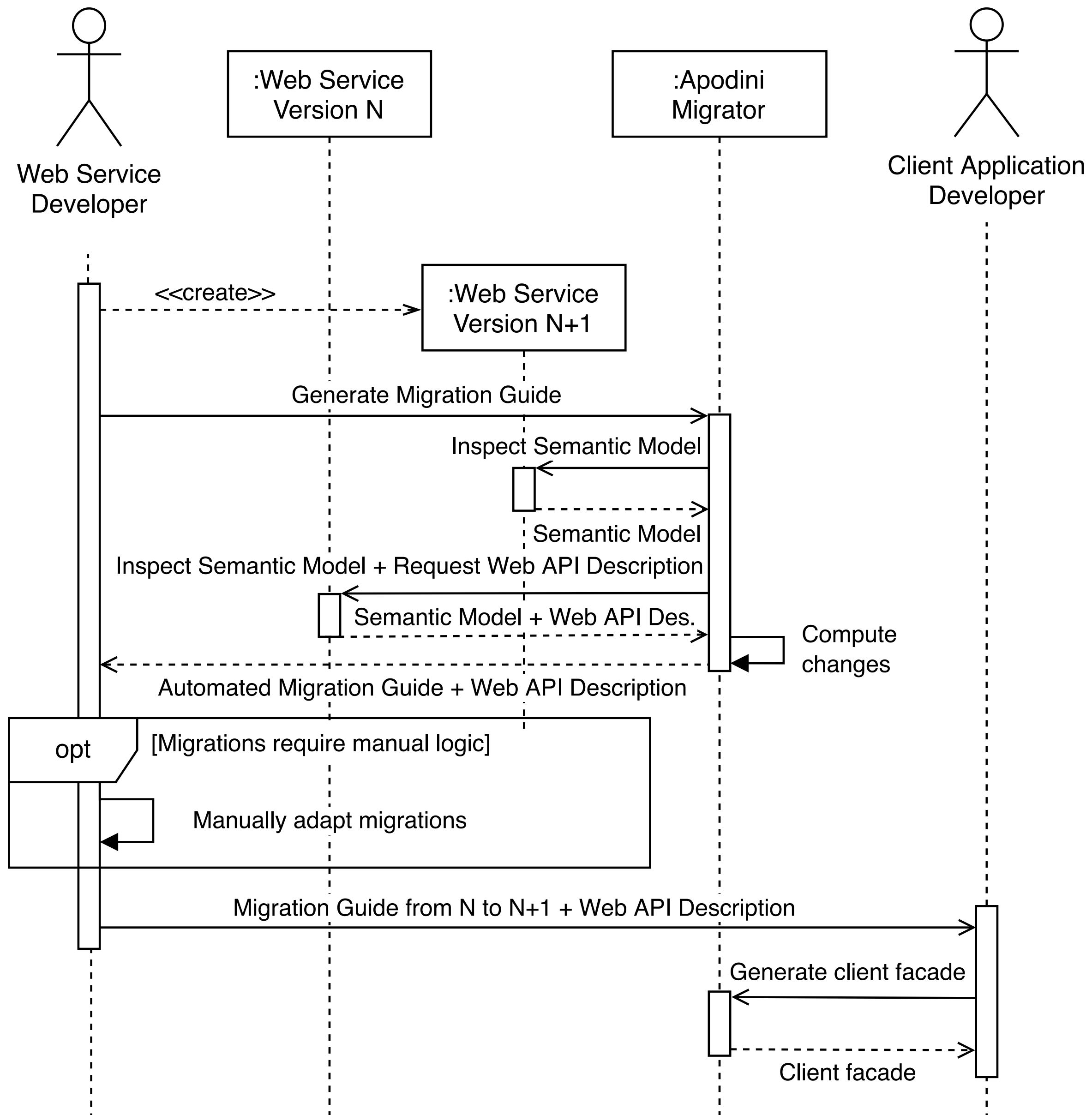


Figure 6.3: Control flow describing the inner workings of the migration mechanisms instantiated in the Apodini Migrator subsystem. The Migrator subsystem inspects the semantic model of two web service versions to create an automated migration guide and a web API description document. The web service developer can further refine the migrations before passing them to the client application developer to generate a client facade. (UML Sequence Diagram)

6.2.3 Deployer

The sequence diagram in Figure 6.4 demonstrates the interactions of an Apodini Deployment Provider based on the Apodini Deployer subsystem when generating a partitioned web service.

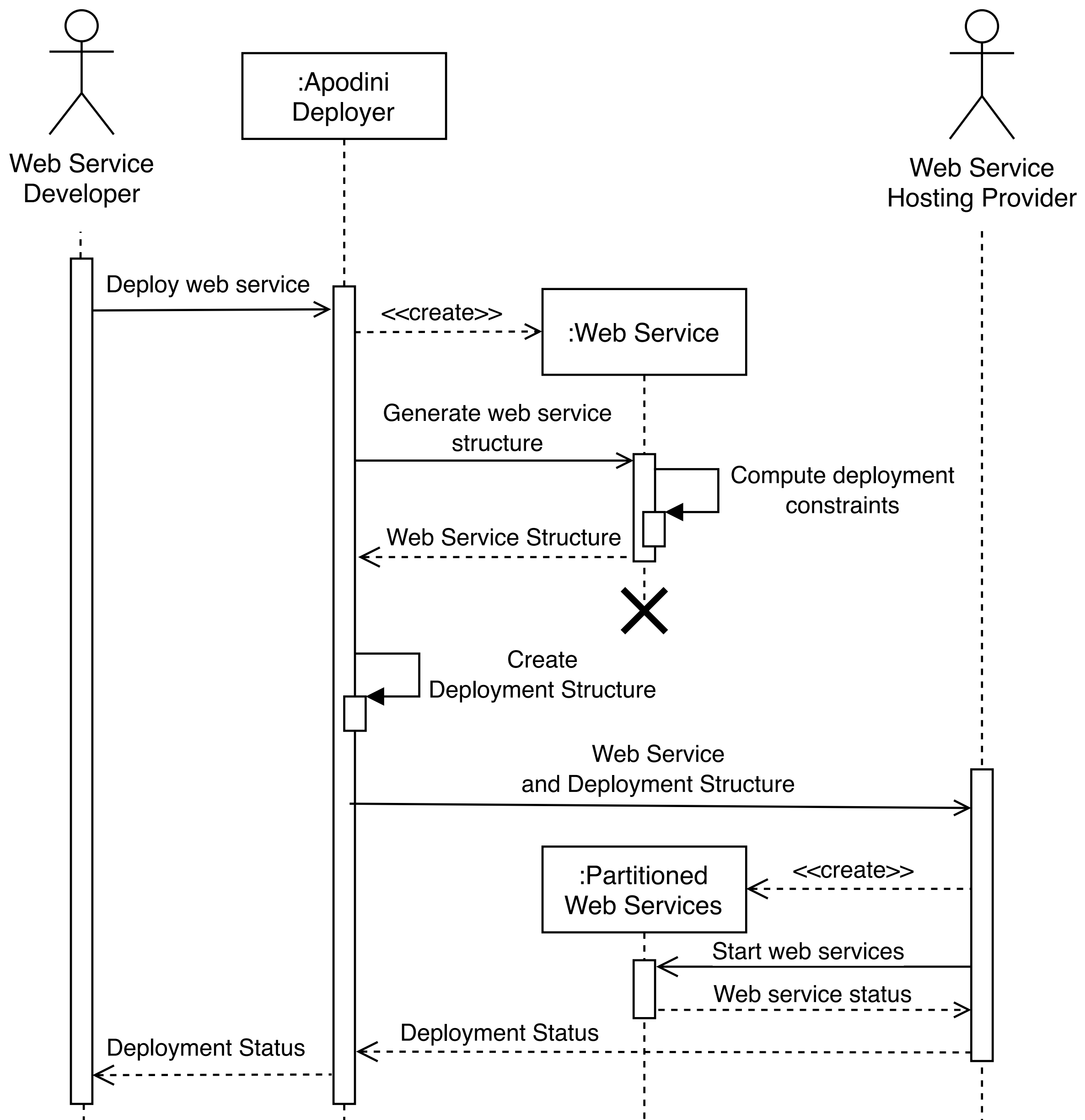


Figure 6.4: Control flow describing the web service partitioning approach of the Apodini Deployer subsystem. The web service structure is exported from the web service used to create a deployment structure as described in Section 7.4. The deployment structure and a web service executable are used to create the partitioned web service in the execution environment. (UML Sequence Diagram)

The web service developer wants to deploy a web service to a web service hosting provider that, e.g., provides a FaaS-based hosting infrastructure as described in Section 5.1.1. Apodini Deployer accesses the web service to generate the web service structure to start the deployment. This structure is computed based on the deployment constraints provided by the web service developer, including metadata annotations or other structural components in the web service. This web service

structure is then returned to an instance using Apodini Deployer that transforms the web service structure into a web service Deployment Provider-specific deployment structure. The deployment structure and the web service are then provided to the web service hosting provider in formats such as deployment commands and containers to deploy the web service. The web service hosting provider then instantiates the partitioned versions of the web service by starting the web services with information about the purpose and position of the web service in the overall deployment structure. This information is used by the web service to adapt its functionality based on the provided deployment constraints and annotations.

6.3 Software Architecture

During system design, subsystems that provide services to other subsystems are defined, interfaces are refined, and subsystems are decomposed into smaller software components [62]. Figure 6.1 provides the top level design of the system with simplified interfaces between the primary subsystems. Simplifications in the top level design are concretized as the interfaces between the components are further refined. This section further decomposes the subsystems into smaller components that provide and consume interfaces to other components, subsystems, or external stakeholders building system extensions. This decomposition is derived from the goals and design problems defined in Chapter 1 and the knowledge gained from the knowledge context and the following chapters. The section is divided up into four subsections, each investigating one of the subsystems presented in Figure 6.1.

6.3.1 Apodini

The first decomposed and further refined subsystem is the primary Apodini subsystem. This subsystem consists of the functionality to express a web service independently of a web service interface type or web API type and builds the foundation for the functionality enabled by the other subsystems. Figure 6.5 details an overview of the Apodini subsystem, including its external interfaces and internal components.

The Apodini subsystem offers the Apodini DSL to web service developers, consisting of the Apodini DSL components and interfaces to extend DSL components used to describe a web service. The functionality describing the Apodini DSL itself is the first main component in the subsystem.

The DSL is extended with the metadata system, extending the core components with extensible metadata annotations and definitions. The metadata subsystem offers a metadata extension annotations interface to web service developers and web service Deployment Providers. This API allows these stakeholders to extend the metadata definitions with additional technology or application domain context.

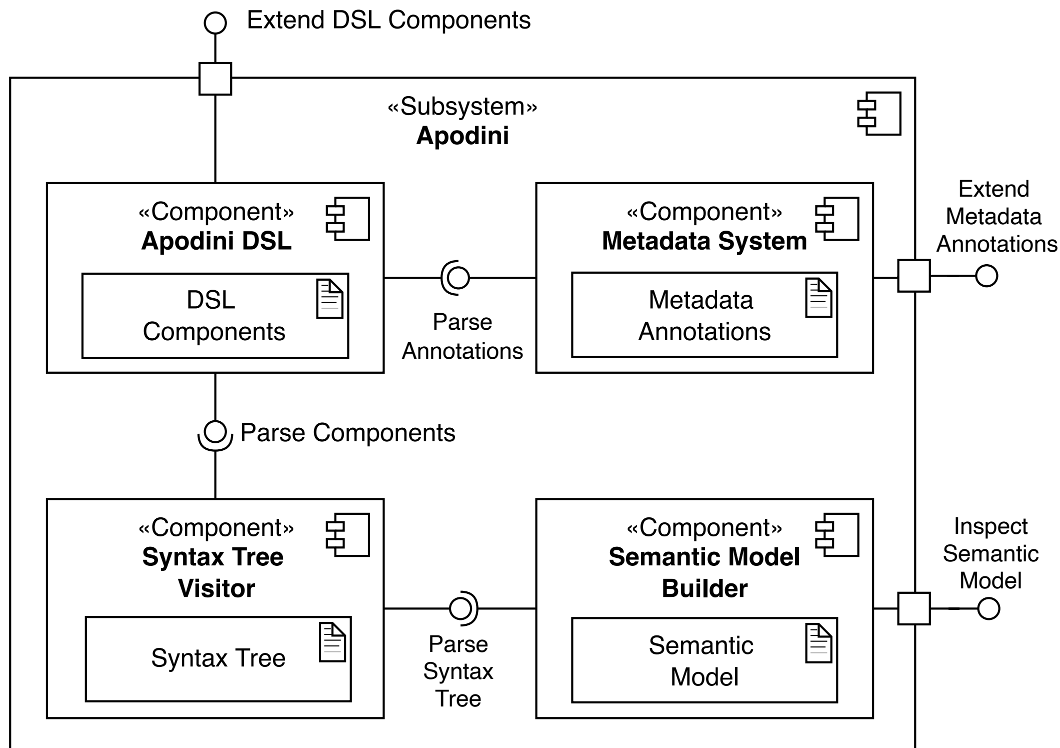


Figure 6.5: The subsystem decomposition of the Apodini subsystem. The subsystem consists of four components addressing different aspects of describing a web service and parsing this description into a semantic model that can be consumed by other subsystems. (UML Component Diagram)

Similar to compilers, parsing a DSL is often performed by building up a syntax tree that is populated from the DSL description [107]. The visitor pattern is a typical pattern used to parse such graph or tree-like structures [109]. Therefore the syntax tree visitor component is responsible for parsing the DSL components and generates a syntax tree.

This syntax tree is the primary input of the semantic model builder component. Martin Fowler defines semantic models as an in-memory representation of the DSL-subject designed to express the DSL's domain in a structured way [107]. The semantic model builder component is responsible for parsing the syntax tree and building a coherent semantic model offered to other subsystems.

6.3.2 Interface Exporter

The Interface Exporter subsystem is responsible for providing extension points for Interface Exporters to generate concrete web APIs based on the abstract definitions from the Apodini DSL. The subsystem, its components, and its internal and external interfaces are depicted in Figure 6.6.

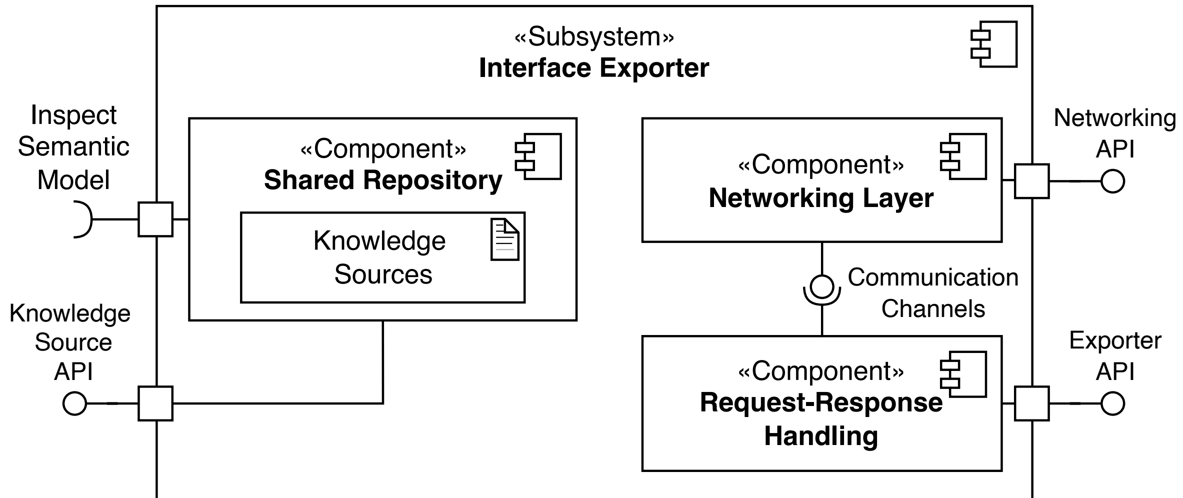


Figure 6.6: Subsystem decomposition of the Interface Exporter subsystem. The subsystem consists of four components addressing the knowledge gathering and the networking and communication pattern handlings for Interface Exporters extending Apodini. (UML Component Diagram)

The semantic model and other knowledge about the structure, functionality, non-functional requirements, and other metadata is stored in a shared repository. According to Buschmann et al., the *"SHARED REPOSITORY architecture allows integration of application functionality with a data-driven control flow to form coherent software systems"* [67]. A shared repository is *"the central control coordination entity and data access point of a data-driven application"* [67]. Knowledge sources feed the shared repository with information determined from the semantic model or based on other knowledge sources. Knowledge sources are components typically found in the more indeterministic blackboard pattern [67]. In contrast to the blackboard pattern, the knowledge sources and the shared repository try to achieve a deterministic behavior. Additional Knowledge Sources are created using a Knowledge Source API, which web API-specific Interface Exporters can use to extend the Apodini Interface Exporter subsystem.

The second set of components in the Interface Exporter subsystem is the networking input/output (Networking I/O) and the request response handling components. The networking I/O component provides a shared networking layer to different Interface Exporters. The request-response handling component builds on top of this functionality and provides a shared functionality to implement different communication patterns based on the Handlers expressed in the Apodini DSL. Different communication patterns such as service-side streams and bidirectional streams require state preserving logic provided by the component. The request-response handling component relies on communication channels structuring requests and responses on the networking layer. The exporter API is provided to Interface Exporters implementing communication patterns beyond a request-response pattern.

6.3.3 Migrator

The Migrator subsystem addresses web service API evolution. The Migrator subsystem is based on previous artifact instantiations in the dissertation research project addressing web API evolution. Andre Weinkötz developed the Pallidor project[18] as an exploratory project to automatically migrate web APIs based on changes in OpenAPI specifications [304]. Building on top of the insights from Weinkötz and the Pallidor instantiation, Eldi Cano [69] and Andreas Bauer [35] developed the Apodini Migrator subsystem[19]. The subsystem decomposition in Figure 6.7 describes the three main components that are set out to achieve client stability while enabling web service API evolvability.

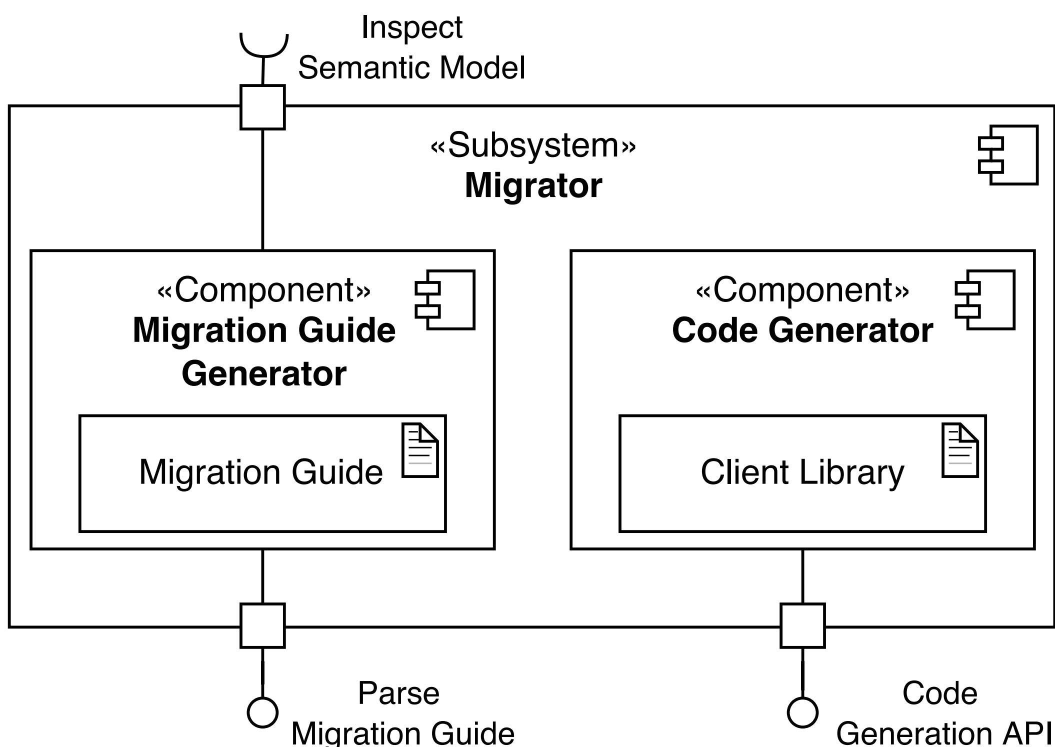


Figure 6.7: Subsystem decomposition of the Migrator subsystem. The subsystem consists of three components addressing web API evolution by generating a migration guide that can be used to create a web API-specific stability guaranteeing client stability. (UML Component Diagram)

The subsystem generates the migration guide based on the semantic model provided by the primary Apodini subsystem. The top level design (Figure 6.1) describes the interface between the Apodini subsystem and the Migrator subsystem as the migration guide to explain the overall interactions of the system. The subsystem decomposition in Figure 6.6 provides a more detailed look at the subsystem

[18]The Pallidor project can be found at https://github.com/Apodini/Pallidor.
[19]The Apodini Migrator project can be found at https://github.com/Apodini/ApodiniMigrator.

and moves the generation of the migration guide in the Migrator subsystem. This provides a lower coupling and higher cohesion between the subsystems. The migration guide generator transforms two versions of the semantic model into a web API type agnostic migration guide. The dynamic process of this transformation is detailed in Section 6.2.

The Apodini Migrator instantiation uses the web API agnostic migration guide and a web API description and identifies the concrete API changes. These API descriptions and changes are combined with the migrations and are passed to the code generator component. This component provides the functionality to generate client stubs based on an extensible API that can be adapted and refined based on the required web API type.

6.3.4 Deployer

The last subsystem that is closely inspected is the Deployer subsystem shown in Figure 6.8. The Apodini Deployer subsystem was developed as part of the Apodini research project and was instantiated by Lukas Kollmer in the bachelor's thesis *Automated and User-Configurable Deployment of Web Services* [159] and extended in *Automatic Deployment and Dynamic Reconfiguration of Web Services in Heterogeneous IoT Environments* by Felix Desiderato [83] as demonstrated in Chapter 9.

The Deployer subsystem uses the semantic model to generate a web service structure in the web service structure generator component. The component uses the web service structure, the context provided in the DSL, and metadata-based annotations to provide a deployment-focused representation of web services. The web service structure is then transformed by the deployment structure generator into a deployment structure that Deployment Providers instantiations can extend.

Different web service hosting providers require different runtime environments to start and communicate with the web service. The deployment runtime component provides a basis to build custom components and networking layers to support the web service deployment. This networking API is consumed by the cross deployment node communication component. As noted in Section 5.1, partitioning web services into smaller individually deployable nodes can be beneficial to support modern FaaS or IoT-based deployments. When a web service developed as a single deployable entity such as a microservice is partitioned into smaller elements, the communication between different microservice functions still needs to be preserved in the partitioned state. The cross deployment node communication component enables this functionality and provides extensible APIs that different Deployment Providers can support.

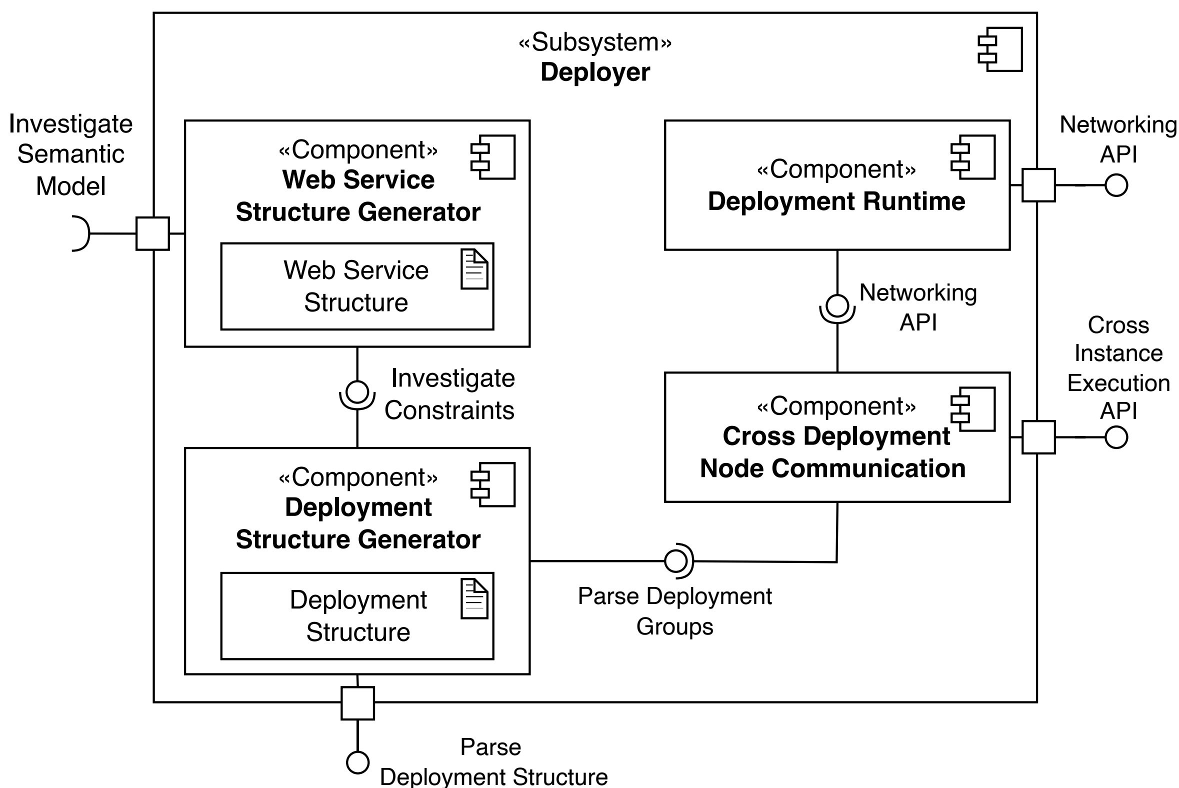


Figure 6.8: Subsystem decomposition of the Deployer subsystem. The subsystem consists of four components addressing web service deployment evolution by generating a deployment structure adapted to dynamic and static deployment-related constraints. (UML Component Diagram)

Chapter 7

Object Design

The Object Design chapter focuses on describing interfaces and key artifacts contributing to the Apodini ecosystem. The object design identifies the reuse of off-the-shelf components, details service interface specifications, and improves and optimizes the object models to address the design goals [62]. The open-source instantiations of the Apodini ecosystem include automatically generated documentation for the public interfaces of the individual subsystems, providing more insights into the exact programming language-specific design of the APIs[20]. The detailed UML class diagrams provide simplified and programming language-independent descriptions of the modeled artifacts instantiated using suiting programming language features in the instantiations.

Section 7.1 provides an overview of the object design of the Apodini internal domain-specific language used to describe evolvable web services. The section introduces an abstract class diagram modeling the main components of the DSL and introduces an instantiation using the Swift and Kotlin programming languages. Section 7.2 details the structure of the semantic model generated from the syntax tree that represents a parsed version of the DSL introduced in Section 7.1. Section 7.3 details a closer look at the migration guide documenting changes and migrations between two web service versions. The deployment structure modeled in Section 7.4 defines a high-level description of the decomposed system that Deployment Providers can use. Section 7.5 describes the API of the cross deployment node communication mechanisms, supporting partitioned web services using the Apodini Deployer subsystem.

[20]The Apodini reference instantiation, examples, templates, and documentation can be found in the Apodini GitHub organization: https://github.com/Apodini. A large proportion of the instantiations and validations are performed using the Swift programming language.

7.1 Domain-Specific Language Components

The Apodini ecosystem uses a domain-specific language to define web services. Chapter 3 describes and specifies the challenges of web service interface evolution in detail. The chapter provides an overview of related work and existing web service interface types, API types, and tools to develop evolvable web services. This section introduces the Apodini DSL and its instantiations based on the metamodel presented in Figure 3.1 (page 48).

Similar to research by Treiber et al. [278, 279, 280], the Apodini programming model focuses on enabling web service evolvability by describing the web service in serializable and composable blocks. While the Apodini ecosystem is not mainly focused on runtime adaptation, the insights provided by Treiber et al. motivates the advantages of combining the web service interface description with the service functionality. Research by Treiber et al. demonstrates an abstract, tree-based programming methodology to develop evolvable web services based on the Genesis framework [278]. The script-based programming methodology enables a dynamic binding of functionality in Java-based web services, enabling dynamic runtime adaptations [278]. The modularity and runtime-adaptation fit the goal of runtime adaptability while bringing the functionality implementation and service description closer to each other than in traditional service description languages [278].

Wittern et al. highlight two approaches of generating API specifications that often require significant manual effort and suffer from sparse user input: using source code annotations which are retroactively added to the service interfaces, and automatic web API detection approaches using dynamic runtime-based data gathering from requests and responses [313]. In contrast, Apodini presents a web service development approach using an internal domain-specific language that combines a code-first and model-based approach discussed in Chapter 3, as well as web service development concepts presented in related work.

Using an internal domain-specific language provides several key advantages when approaching different web service evolution challenges discussed in this dissertation. One key advantage is the change to express service interface, deployment structure, and configuration information in a single parsable structure while embedding and reusing functionality from a general-purpose programming language. Reusing the IDE and compiler-based infrastructure as well as DSL-language features in general-purpose programming languages reduces the complexity of parsing and validating the web service implementation [107]. Compilers and type systems of programming languages constrain the expressiveness of structural and annotation-based aspects of the web service description while providing the full power and reusability of dependencies and language features when implementing web service functionality.

Similar developments in using internal domain-specific languages can also be observed in cross-platform user interface frameworks. The Flutter ecosystem enables the development of user interfaces across the web, mobile, and desktop platforms and uses an internal DSL to describe user interfaces in the Dart programming language[21]. Jetpack Compose is a modern declarative internal DSL to develop Android applications embedded in the Kotlin programming language[22]. Compose Multiplatform extends Jetpack Compose beyond mobile applications to support web and desktop applications[23]. SwiftUI is a user interface development internal DSL embedded in the Swift programming language to develop applications running across Apple operating systems[24]. The focus on internal domain-specific languages (Definition 10, page 32) in modern, typed programming languages enables new features and language support enabling new use cases for internal DSLs. Similar to other application domains of internal DSLs, specifying the structure, constraints, and functionality in a single development artifact enables a holistic development approach for web service development. The single source of truth proposed with the Apodini DSL extends beyond current infrastructure as code approaches described in Section 5.1, providing a cohesive **Everything in Code** (EiC) approach.

Figure 7.1 presents the main components in the Apodini DSL, enabling web service developers to implement evolvable web services. The components are based on the problem investigation performed in Chapter 3. The central abstraction is the web service. The web service configuration provides an extension point for the web service developer to incorporate and configure the functionality of the web service. This includes the creation of commands and nested subcommands, exposing the functionality of the different Apodini subsystems and the configuration of these subsystems, including Interface Exporters. A web service can have several launch time arguments allowing startup time adaptation of the configurations. In addition, metadata annotations are extension mechanisms for the DSL, enabling application domain or software engineering workflow supporting annotations. The tree-based composite structure of the web service is formed by a composition of components providing the content of the web service. Components can be nested and annotated with metadata. The leaves of the tree are Handlers, uniquely identifiable in the web service structure. As introduced in the web service metamodel, Handlers are responsible for transforming input based on parameters into a response. This mechanism is achieved using a handle method. The parameter name uniquely identifies the parameter in a Handler. The possibility to provide a default value and

[21] Flutter documentation can be found at https://docs.flutter.dev/.

[22] Jetpack Compose documentation can be found at https://developer.android.com/jetpack/compose.

[23] Compose Multiplatform documentation can be found at https://www.jetbrains.com/lp/compose-mpp/.

[24] SwiftUI documentation can be found at https://developer.apple.com/xcode/swiftui/.

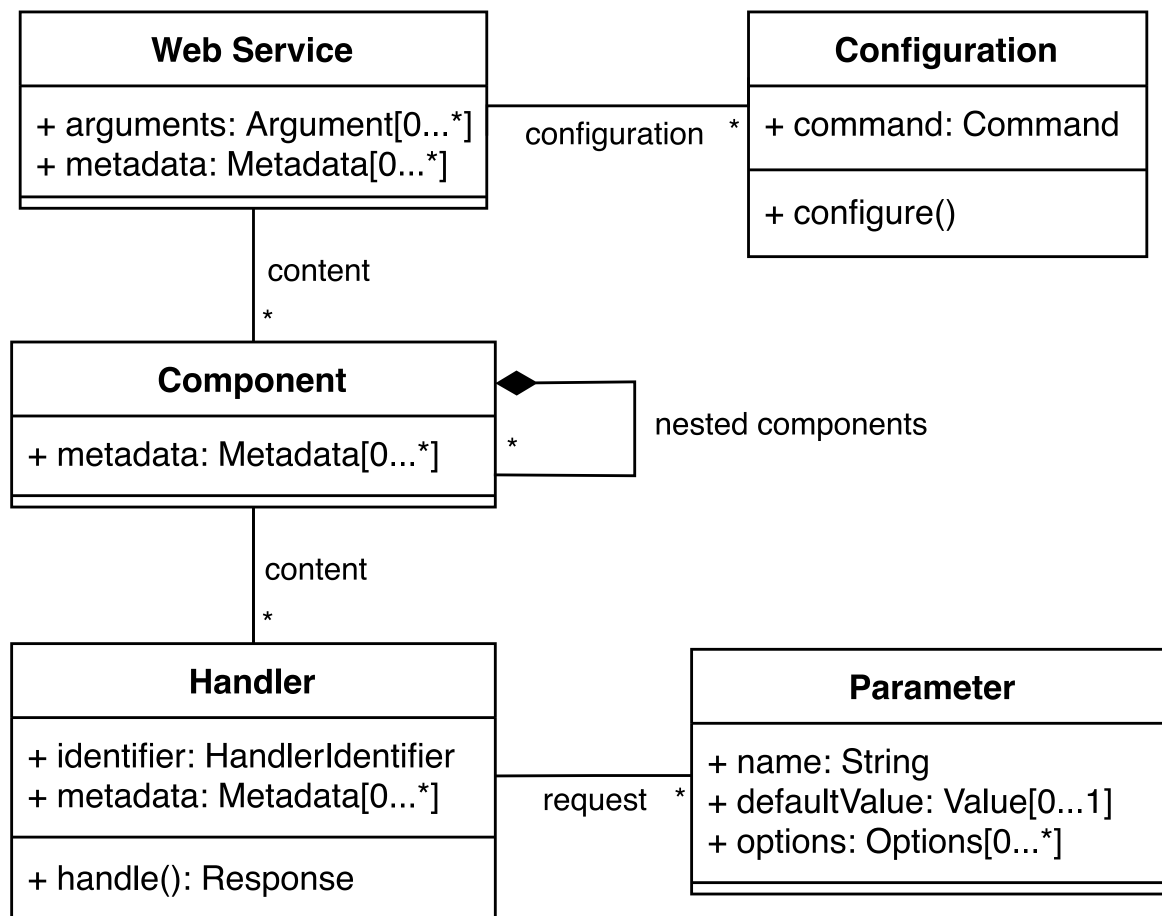


Figure 7.1: Main components of an instantiation of the Apodini DSL. The DSL is based on the web service metamodel described in Section 3.2. (UML Class Diagram)

parameter-specific options completes the core components of the DSL. The following two subsections showcase an instantiation of the DSL concepts in the Swift and Kotlin programming languages, further refining the interface of the Apodini DSL.

7.1.1 Swift-based Apodini DSL Interface

Listing 7.1 details the Swift-based instantiation of the Apodini DSL components shown in Figure 7.1. The Swift-based Apodini DSL was originally developed by Paul Schmiedmayer and further developed as part of the Server-Side Swift practical course in the winter semester of 2020/21 and further extended by several supervised bachelor's theses, master's theses, and guided research projects [249, 230, 159, 318, 83, 69, 34, 202, 35, 270, 160]. The idea of a DSL-based approach to developing web services was first exported using the declarative Swift-based web service development framework named *Corvus* developed as part of a bachelor's thesis by Berzan Yildiz supervised by Paul Schmiedmayer [317].

```
1  @main
2  struct HelloWorld: WebService {
3      var configuration: Configuration {
4          REST()
5      }
6
7      var content: some Component {
8          Text("Hello World!")
9      }
10 }
```

Listing 7.1: Swift-based executable describing a web service with a single Handler returning "Hello World!" using a RESTful Interface Exporter.

In the Swift-based Apodini DSL, a `WebService` has two computed properties: `configuration` containing instances conforming to the `Configuration` protocol and `content` containing types conforming to the `Component` or `Handler` protocols. The Swift protocols such as the `Handler` protocol are instantiations of the stereotypes found in the web service interface metamodel, applying the extensibility and adaptability of the metamodel for the DSL-based instantiation. The computed properties are evaluated using result builders, a Swift feature to enable declarative definitions in internal DSLs. The web service exposes a RESTful API and defines this configuration using the `REST()` configuration without any additional parameters. The `content` property contains a single `Handler`. The struct `Text` conforms to the Swift protocol `Handler` and contains an implementation returning the string passed into its initializer. This `Text` Handler is initialized in the `content` property and defines the only `Handler` in our hello world web service example.

```
1  struct Greeter: Handler {
2      @Parameter var country = "World"
3
4      func handle() -> String {
5          "Hello \(country)!"
6      }
7  }
```

Listing 7.2: Swift-based Handler in the Apodini DSL expecting a parameter with a default value if no value is provided. The `Greeter` Handler returns a greeting in the `handle` function.

Listing 7.2 provides an example for a Swift-based `Handler` using the Apodini DSL. The `Greeter` type conforms to the `Handler` protocol and contains a single stored property named `country`. The Swift property is wrapped using a Swift property wrapper of thee type `Parameter` using the `@Parameter` Swift syntax for property wrappers[25]. The `Parameter` property wrapper is part of the DSL API

[25]The Swift Language Guide provides an in-depth introduction to the property wrapper-feature: https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID617.

surface and enables the delegation of filling the property to the Apodini framework. The name of the Parameter is inferred from the name of the property in the Greeter type. When the handle function is executed, the Apodini framework guarantees that the property is filled with a valid value. Based on the default value assigned to the property, this value either originates from a request serialization or is initialized with the default value. The handle function uses string interpolation to create the greeting and implicitly returns the string as the return value of the function. The Apodini framework also uses Swift structured concurrency features, in particular the async keyword to enable suspendable computation in the handle function.

7.1.2 Kotlin-based Apodini DSL Interface

Listing 7.3 details a Kotlin-based version of the Apodini DSL. The Kotlin implementation uses builder functions instead of result-builder-based computed properties used in the Swift implementation to define the configuration. The Kotlin-based Apodini DSL was instantiated as part of a guided research project, investigating the principles of the Apodini DSL across different programming languages by Mathias Quintero [230].

```
1  fun main() {
2      HelloWorld.run()
3  }
4
5  object HelloWorld : WebService {
6      override fun ConfigurationBuilder.configure() {
7          use(REST())
8      }
9
10     override fun ComponentBuilder.invoke() {
11         text("Hello World")
12     }
13 }
```

Listing 7.3: Kotlin-based executable describing a web service with a single Handler returning "Hello World" using a RESTful Interface Exporter. The web service is started in the main method above the web service declaration.

The Kotlin implementation exposes the builder functionality in the DSL and uses this feature to compose Handler and Components using functions and operations defined on the ConfigurationBuilder and ComponentBuilder. The text function to create a Text Handler is implemented as an extension function on the ComponentBuilder type.

```kotlin
class Greeter : Handler<String> {
    private val country: String by parameter {
        default("World")
    }

    override suspend fun CoroutineScope.handle(): String {
        return "Hello $country!"
    }
}
```

Listing 7.4: Kotlin-based Handler in the Apodini DSL expecting a parameter with a default value if no value is provided and returning a greeting similar to Listing 7.2. The Kotlin version of the Greeter uses string templating to create the greeting returned as a response.

Listing 7.4 implements the `Greeter` `Handler` in the Kotlin-based Apodini DSL. The Kotlin implementation uses property delegates to achieve similar functionality as the property wrappers used in the Swift implementation. These features enable the wrapping and delegation of functionality to the Apodini framework, filling the property based on the incoming request or the default value. The Apodini framework uses Kotlin concurrency features, in particular Kotlin coroutines, to enable suspendable computation in the `handle` function.

7.2 Semantic Model

Semantic models depict a parsed representation of the subject detailed in the domain-specific language [107]. In Apodini, the semantic models is used to provide context and information to Interface Exporters and other Apodini subsystems.

Figure 7.2 features a simplified version of the semantic model present in the Swift and Kotlin instantiations of the Apodini DSL. The instantiations further enrich the semantic model with language constructs as well as the shared repository pattern and knowledge sources in the Swift instantiation as detailed in Section 6.3.2. Metadata information is saved and interpreted using the knowledge source and shared repository mechanism and is not represented in the semantic model in Figure 7.2.

The semantic model allows registering the web service and Handlers. The DSL components are interpreted based on a syntax tree and are appended as general information in the semantic model or transformed into endpoints. After all information from the syntax tree is parsed and finish registration is called, the semantic model can finalize the context and relationships between endpoints. The endpoint relationships indicate functional or non-functional dependencies of the endpoints. This information can be used to, e.g., generate HATEOAS links in the RESTful Interface Exporter as demonstrated in the Part IV. Each endpoint is associated with a Handler, containing further context about the exported endpoint. Path components indicate the Handler hierarchy in the DSL and are used by exporters to generate

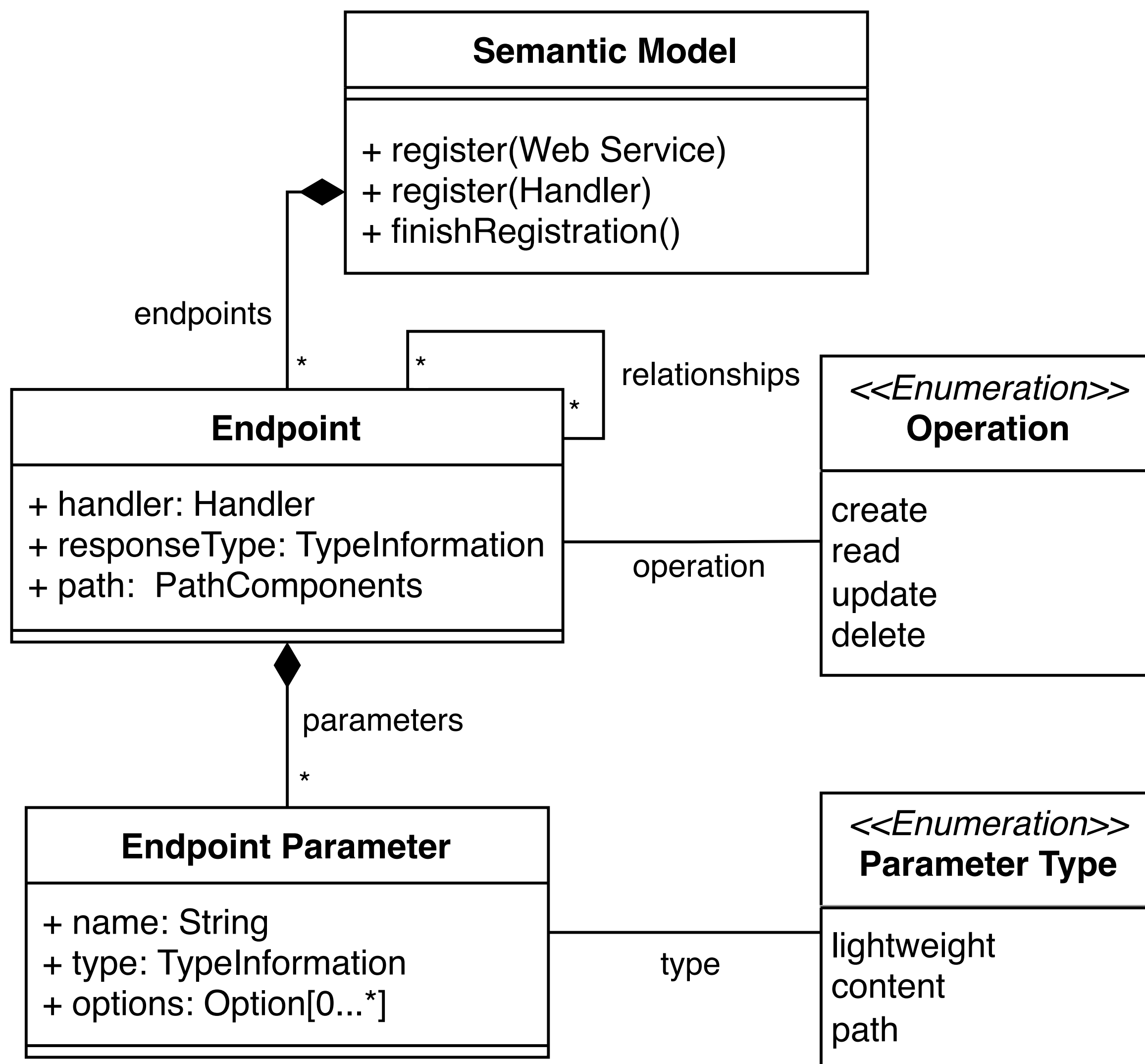


Figure 7.2: Simplified semantic model expressing an interpreted version of the core concepts of the Apodini DSL. `Endpoints` are the core construct of the semantic model, organizing `Handlers` and their context in an in-memory representation. (UML Class Diagram)

URIs, method names, or other identifying information combined with additional context provided in extensible knowledge sources. The response type type information provides type information such as fields about the return type of the handle function of the Handler. The type information is used to generate serialization descriptions and create service interfaces in Interface Exporters. Each endpoint is also associated with an operation mapping the functionality to one of the CRUD operations: create, read, update, and delete. Exporters can use this information to decide on protocol-specific mechanisms such as HTTP methods or organizing functionality into queries and mutations as done in GraphQL. Endpoint parameters contain

interpreted information about parameters and their options and occurrence in the Apodini DSL. Endpoint parameters contain a name and type information about the programming language-specific type used to express the endpoint parameter in the web APIs. The options specified on a parameter are parsed into options on an endpoint parameter. One specific option explicitly modeled in the semantic model is the parameter type. Lightweight parameters are constrained to types that can be represented using textual representations such as strings and numbers and can, e.g., be mapped to URI query parameters in HTTP-based Interface Exporters. Content parameters have no constraints and are the most typical parameter type. Path parameters are inferred from parameters defined outside of Handlers in the component tree and contain similar restrictions as lightweight parameters. The semantic model builder provides a best-effort approach to categorizing endpoint parameters in these groups. Interface Exporters can then apply this information to the protocol, middleware, and API type-specific constraints of defining parameters.

7.3 Migration Guide

This section provides a mapping of the change patterns presented in Section 4.2.1 to a concrete migration guide used in the Migrator subsystem. The class hierarchy presented in Figure 7.3 provides an overview of the core concepts of the migration guide, while the JSON-based representation of a migration guide contains more detail and information relevant for the specific implementations. The open-source documentation[26] of the Apodini Migrator subsystem provides more information about the structure and implementation details of the migration guide.

Each migration guide is associated with one or more web API type, allowing web API-specific annotations. While the structure of the migration guide is web API-independent, these extensions can simplify the reasoning about web API changes for Migrators. Each migration guide describes the migration from a version to a newer version. A migration guide consists of a collection of changes that are associated with elements in the web API. Elements are uniquely identified by identifiers in the respective web API description and can range from endpoints to model types representing requests and responses. The changes are classified into breaking and non-breaking changes following the web API evolution patterns introduced in Section 4.2.1. The solvable property indicates if Apodini Migrator could automatically solve the web API change by providing a migration for the change. The web service developer can manually provide migrations in the migration guide if there is no automatically solvable solution. The migration guide allows three different change types: additions, removals, and updates. An addition contains information about the added element and a default value to resolve breaking changes when making an

[26]The open-source Apodini Migrator is located at https://github.com/Apodini/ApodiniMigrator.

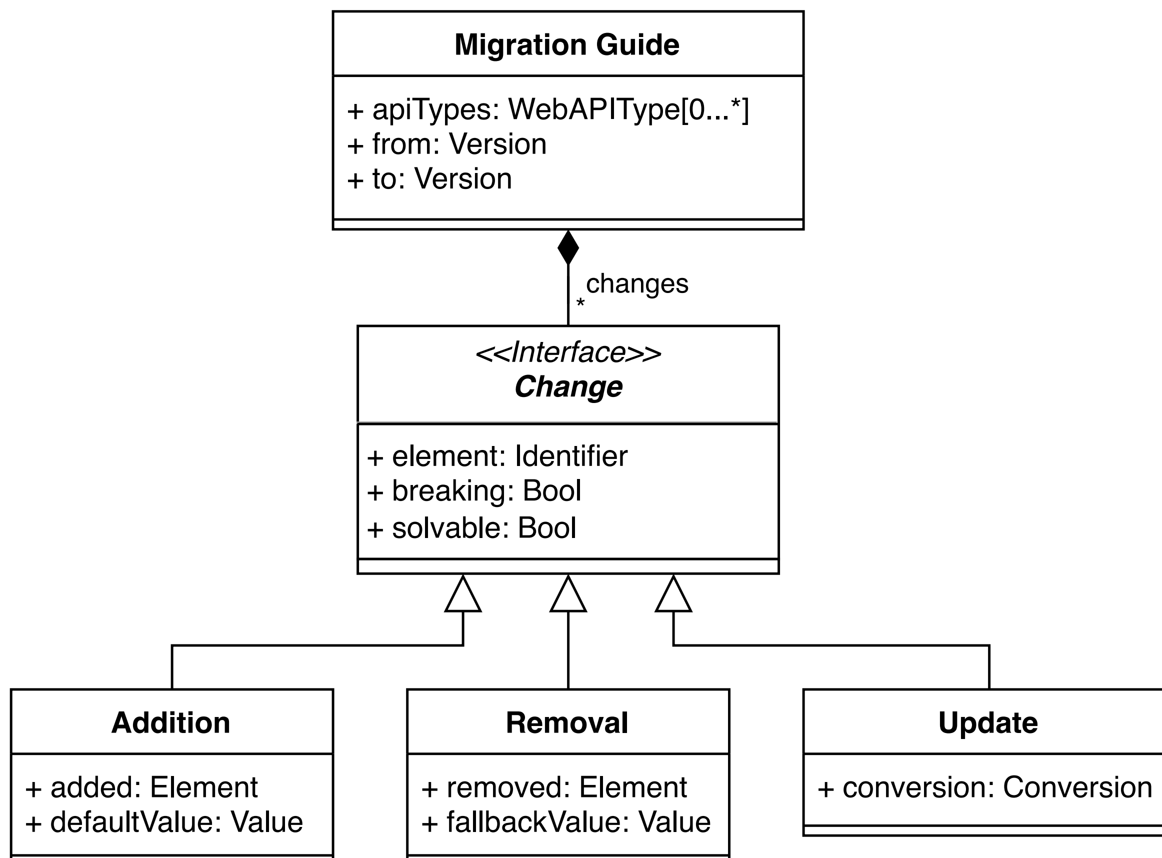


Figure 7.3: Simplified UML class diagram representing the migration guide structure in the Apodini Migrator subsystem. Changes are associated with identifiers, identifying endpoints and types in the web API descriptions. Changes contain different migration contexts based on the change subtype. (UML Class Diagram)

addition to a request serialization. A similar structure applies to removal changes. The removed attribute describes the structure of the removed element to provide information about the type represented in the fallback value used to migrate breaking changes resulting from removing an endpoint or response serialization field. The update change contains all changes updating information about different aspects of the web API, further specified by the element Identifier. The conversion property indicates any migration steps needed to provide client stability. Different subtypes of conversions address various update-based changes such as changing the field names, endpoint identifiers, and model types.

Listing 7.5 demonstrates the instantiation of the simplified UML model presented in Figure 7.3 of the Swift-based Apodini Migrator instantiation in a JSON format. The JSON representation demonstrates the structure of the migration guide, starting with context information such as a short descriptive summary, the document identifier, the version of the migration guide document format, and the web API versions that are migrated in the migration guide. The compare configura-

```json
{
    "summary": "...",
    "document-id": "...",
    "version": "2.0.0",
    "from": "v_1.0.0",
    "to": "v_2.0.0",
    "compare-config": { /* ... */ },
    "serviceChanges": [ /* ... */ ],
    "modelChanges": [
        {
            "type": "addition",
            "id": "...",
            "added": { /* ... */ },
            "breaking": false,
            "solvable": true
        },
        /* ... */
    ],
    "endpointChanges": [ /* ... */ ],
    "scripts": {
        "0": "function convert(input) {...}"},
    "json-values": {
        "0": "..."
    },
    "updated-json-representations": {
        "ModelType": "..."
    }
}
```

Listing 7.5: Structure of the JSON representation of an Apodini Migrator migration guide documenting the changes to a web service API. The JSON representation presented omits the compare configuration, changes to the service, all but one model change, and endpoint changes and replaces them with "/* ... */". The figure only presents one model change, leaving out the details of the addition change.

tion persists settings of the Apodini Migrator system. The following sections of the machine-readable migration guide document the changes categorized into three groups to support easier parsing of the migration guide: change to the service configuration, changes to model types of the service, and changes to the endpoints of the web service. Service changes include changes in the decoding configuration or exporter-specific changes that are persisted in an extensible key-value store.

The model changes include addition, deletions, or updates to model types used as parameters or return types in the Handlers. They are instantiations of the request and response serialization change patterns in Table 4.1 (page 67). Documenting the changes separate from the endpoint changes allows an easier generation of the client library as changes do not have to be documented at every endpoint that uses the model types. Changes are categorized into breaking and solvable changes as described in Figure 7.3. Endpoint changes are instantiations of change patterns concerning Handlers as documented in the change patterns in Table 4.1 (page 67).

```swift
@main
struct MigratorExampleWebService: WebService {
    var metadata: Metadata {
        Version(major: 2)
    }

    var configuration: Configuration {
        REST()
        Migrator(
            documentConfig: .export(.endpoint("api-document")),
            migrationGuideConfig: .compare(
                .resource(.module, fileName: "api_v1.0.0", format:
                ↪  .json),
                export: .endpoint("migration-guide", format: .json)
            )
        )
    }

    var content: some Component { /* ... */ }
}
```

Listing 7.6: Configuration of the Apodini Migrator subsystem in a Swift-based Apodini web service. The `Migrator` configuration enables the configuration of the API document export functionality as well as the generation of a migration guide based on a file in the Swift Package resource bundle. The generated migration guide is offered at an HTTP endpoint similar to the validation in Section 9.2.

The scripts, JSON values, and updated JSON representations provide manual customization points for web service developers. Change scripts enable the conversion of complex types, allowing developers to specify JavaScript-based conversion functions between two model types. These JavaScript-based change functions are called when a stable client library needs to perform complex and potentially non-solvable API changes. The JSON values and updated JSON representations provide default values and fallback values used as migration strategies as documented in the API change pattern classification in Table 4.1 (page 67).

Section 9.2 demonstrates a detailed look (Listing 9.5, page 165 and Listing 9.4, page 164) at the JSON representation of migrations based on web API changes in an event management platform web API used to validate the Apodini Migrator-based web API evolution approach.

The Apodini Migrator subsystem can be directly integrated into Apodini web services to export API documents and migration guides easily using subcommands or specified settings in the Apodini web service configuration property. Listing 7.6 demonstrates the Apodini Migrator configuration in an Swift-based Apodini web service. The configuration allows documents and migration guides to be exported at HTTP endpoints at runtime or exported to a file at startup time. Exporting the documents at startup time is helpful when developing the web service or distributing the API document or migration guide through external channels.

The generation of the migration guide requires API documentation of a previous web API version that can be loaded from the local file system or bundled with the web service binary using Swift Package resources. Depending on the configuration, the migration guide is exported on an endpoint or to a file on disk. Listing 9.1 describes a second example of an Apodini Migrator configuration in a web service. Section 9.2 also demonstrates the usage of the command-line interface to generate API documents and migration guides from web services, including the Apodini Migrator configuration.

7.4 Deployment Structure

This section describes the deployment structure introduced as an artifact in Figure 6.8 (page 108). Structural descriptions of web service deployments are essential to deploy and continuously update software components in distributed systems. Deployment systems such as the Disnix toolset by van der Burg and Dolstra and commonly used tools such as Kubernetes use mostly declarative configurations of the deployable services, the deployment infrastructure, and a mapping of components to infrastructure as an input [65, 286, 287].

The deployment structure describes an intermediate step in the Apodini Deployer subsystem. It is generated from the web service structure defined in the Apodini DSL and the execution environment constraints defined by an Apodini Deployer instantiation. The deployment structure provides a mapping of potentially partitioned functionality of a web service to deployment nodes. The deployment structure detailed in Figure 7.4 describes this intermediate artifact and is extensible to allow different application domains ranging from Function as a Service to IoT-based deployments such as the Web of Things. Deployment nodes are then further mapped to concrete execution environments and deployment infrastructure by the concrete Apodini Deployer instantiations as demonstrated in Chapter 9.

Figure 7.4 presents the deployment structure in a simplified version focusing on the most important aspects found in the instantiation of the Apodini Deployer system. Deployment Provider instantiations use the infrastructure described in the subsystem decomposition (Figure 6.8, page 108) to generate a deployed system. Each deployed system consists of multiple deployment nodes uniquely identified by an identifier and representing the target environment for each Deployment Providers. In a FaaS based deployment, a deployment node represents a FaaS function that can be scaled based on user demand. In a WoT based deployment, a deployment node can represent an IoT Gateway.

Deployment constraints such as performance or hardware constraints are expressed using deployment constraints. Each deployment node consists of one or more exported endpoints representing the endpoints of the semantic model mapped

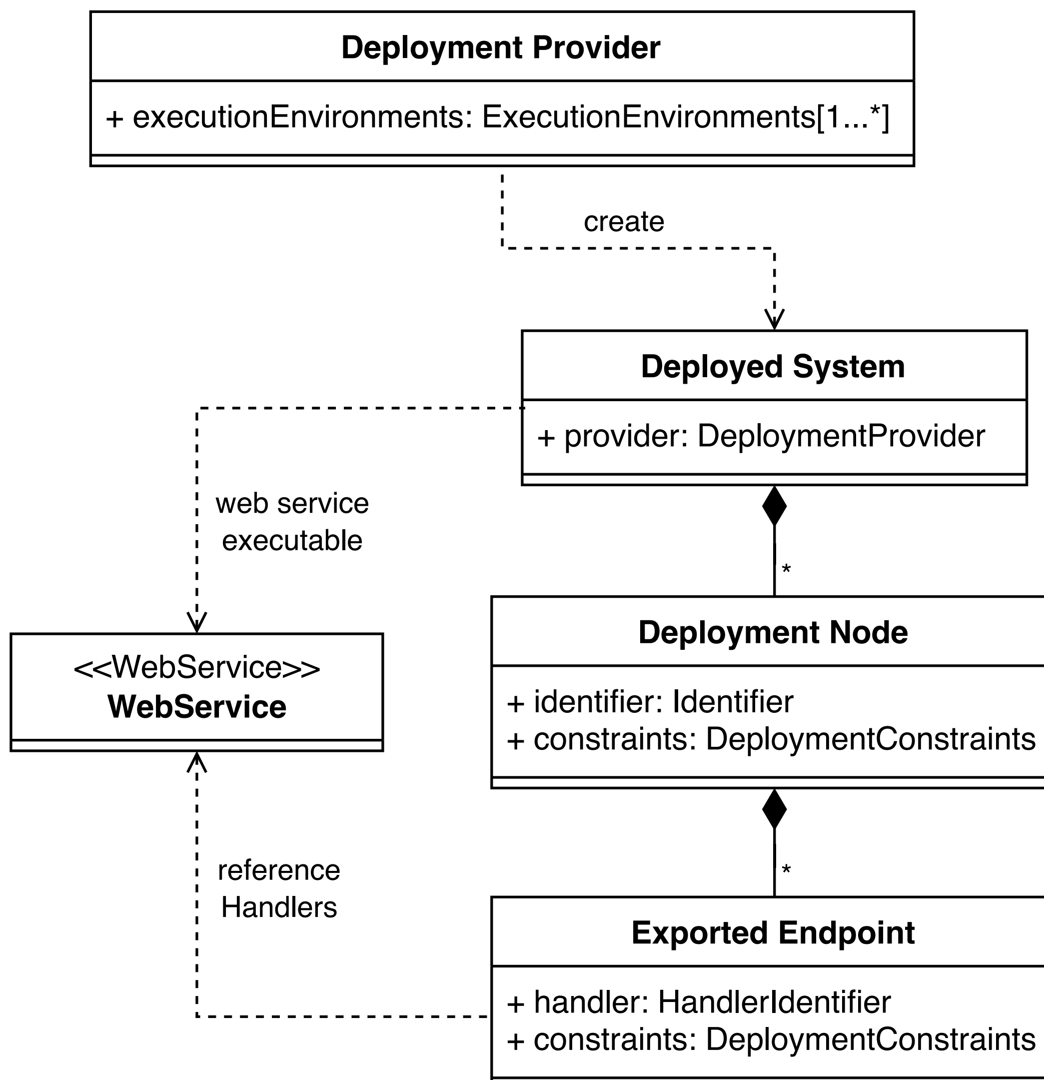


Figure 7.4: The deployment structure documents the output of Deployment Providers extending Apodini Deployer in form of the deployed system type. The structure describes the distribution of exported endpoints to deployment nodes annotated with deployment-specific constraints. (UML Class Diagram)

to the deployment nodes based on the provided deployment constraints, annotations, or other information extracted from the web service structure. Each exported endpoint is associated with a Hander encapsulating the partitioned functionality of the web service identified by a Handler identifier. Additional context required when deploying, starting, or running the functionality is provided using constraints on the exported endpoint. The exported endpoint and deployment node specific constraints are used by Deployment Providers to prepare and choose the appropriate execution environment and enable dynamic reconfigurations at runtime if the execution environment changes.

7.5 Cross Deployment Node Communication

Partitioning of a web service for FaaS and WoT-based deployment requires Apodini Deployer to use the deployment structure presented in Figure 7.4 to determine possible cross-Handler invocations that can no longer happen within a single process. Therefore, the Apodini Deployer system provides the cross deployment node communication component (Figure 6.8, page 108) offering a cross instance communication API. The cross-deployment node communication is instantiated in the Swift-based Apodini Deployer implementation using the remote Handler invocation mechanism. The mechanism uses the deployment groups to automatically determine if a Handler invocation can be dispatched locally or is sent to a remote instance using a custom HTTP-based RPC mechanism.

```swift
struct CreateUser: InvocableHandler {
    class HandlerIdentifier: ScopedHandlerIdentifier<CreateUser> {
        static let main = HandlerIdentifier("main")
    }

    let handlerId = HandlerIdentifier.main

    @Parameter var name = "Paul"
    @Parameter var age = 42

    func handle() async throws -> User {
        let user = User(name: name, age: age)
        // ... store the user in a datastore
        return user
    }
}
```

Listing 7.7: The `CreateUser` Handler showcases the conformance to the `InvocableHandler` protocol including the `HandlerIdentifier`, uniquely identifying the Handler in the Apodini DSL. The Handler returns a created `User` based on a name and the age passed to the Handler.

Listing 7.7 and Listing 7.8 demonstrate the API of the remote Handler invocation mechanism provided by the Apodini Deployer subsystem. Listing 7.7 demonstrates the usage of the `InvocableHandler` protocol, allowing the `CreateUser` Handler to be called from the `SignUp` Handler in Listing 7.8. The remote Handler invocation API automatically determines if the two Handlers are located on the same deployment node in the same deployment group. Based on this check, the `RemoteHandlerInvocationManager` then either dispatches a local method call or a network call to the invoked Handler.

The usage of the Apodini Deployer configuration in Swift-based Apodini web services and Deployment Provider instantiations are demonstrated in Chapter 9.

```
struct SignUp: Handler {
    @Environment(\.RHI) private var RHI
    @Parameter var name = "Paul"
    @Parameter var age = 42


    func handle() async throws -> String {
        let user = try await RHI.invoke(
            CreateUser.self,
            identifiedBy: .main,
            arguments: [
                .init(\.$name, name),
                .init(\.$age, age)
            ]
        )
        return "Hi, \(user.name)! You are \(user.age) years old."
    }
}
```

Listing 7.8: The `SignUp` Handler demonstrates the usage of the Remote Handler Invocation API using the `invoke` function on the `RemoteHandlerInvocationManager` (`RHI`) instance retrieved from the Apodini environment. The `invoke` function requires the Handler type, identifier, and the arguments and subsequently returns the response of the Handler in an async function.

Part IV
Treatment Validation

TREATMENT validation is the final task of the design cycle before treatments are transferred to a real-world context without interference from the researcher and evaluated as part of an engineering cycle [309]. We describe an explorative design science project investigating artifacts to address web service evolution and conduct one complete design cycle.

Wieringa states that *"To validate a treatment is to justify that it would contribute to stakeholder goals when implemented in the problem context"* [309]. In the context of design science projects, implementation refers to transferring something to the problem context and not creating a software system [309]. This part addresses the challenge of validating the artifacts before any transfer to the problem context can occur by describing several single-case mechanism experiments. Single-case mechanism experiments are performed in an artificial context designed by the researchers *"to test an artifact prototype or to simulate real-world phenomena"* [309]. These experiments demonstrate Apodini Interface Exporters (Chapter 8) and web service instantiations in different application domains, demonstrating the extensibility and applicability of the Apodini ecosystem in different application domains (Chapter 9).

Chapter 8

Apodini Interface Exporter

As defined in Definition 6 and further investigated in Chapter 3, web service interface evolution is concerned with the evolution related aspects of web service API, middleware, or protocol types of web services. This chapter validates the web service interface evolution capabilities of the Apodini artifacts described in Part III and instantiated as part of the open-source Apodini implementations. In contrast to case studies which have to be performed in a *"real-life context, and with the investigator(s) not taking an active role in the case investigated"* [314], Part IV uses single-case mechanism experiments to validate the designed artifacts. The treatment validation does not implement the treatment in the real-life problem context without any influence of the researcher as done in an engineering cycle [309]. Single-case mechanism experiments enable controlled validations of the designed artifacts, facilitating us to showcase the advantages and shortcomings of the instantiated artifacts in several experiments conducted in a wide variety of projects and experiment setups [309]. We use several single-case mechanism experiments to validate how web service interfaces designed in the Apodini DSL can be mapped to different web service interfaces and API types. We follow the single-case mechanism experiment workflow defined by Wieringa, validating the newly developed technology and investigating possible improvements based on the validation and insights gathered in the previous parts [309].

Each single-case mechanism experiment in a treatment validation is embedded in a conceptual framework involving the designed artifacts to answer knowledge questions and position the findings in a larger population of validation models [309]. The following sections demonstrate artifacts envisioned in Technical Research Goal 1: the Apodini DSL and Interface Exporter instantiations.

> **Technical Research Goal 1:**
> Design artifacts supporting web service API type agnostic development to enable web service interface evolution.

The single-case mechanism experiments in this chapter involve the creation of Interface Exporters using the conceptual framework presented by the Apodini DSL and Apodini Interface Exporter subsystem. The different Interface Exporters validate the extensibility and evolvability of the artifacts to answer Knowledge Question 2, concerning Technical Research Goal 1. The demonstrated Interface Exporters highlight the generalization approach manifested in the web service metamodel (Figure 3.1, page 48), and extension mechanisms that enable functionality beyond the shared foundation.

> **Knowledge Question 2:**
> Does the Apodini DSL empower web service interface- and web API type-independent web service development?

The design, instantiation, and testing of the Interface Exporters verify the applicability of the artifacts to the Interface Exporter-specific web API type. As the applicability of the DSL to a web API type can not be measured in a discrete value, we investigate the output and transformation from the Apodini DSL to the web service API on a web API type basis. The provided examples provide single-case mechanism experiments involving the Apodini DSL that highlight Design Problem 1.

> **Design Problem 1:**
> Develop all aspects of web services in a web service interface type, web API, middleware, and protocol-independent description so that web service developers can support different web service interface types and web API types without rearchitecting web services.

Each section demonstrates distinct functionality of the Apodini DSL and notes how the functionality can be extended beyond the presented Interface Exporter, highlighting the extensibility and applicability beyond a single web API type. The interface evolution related capabilities of the DSL are validated in three groups:

1. Section 8.1 demonstrates the applicability of RPC-based APIs using a gRPC-based Interface Exporter showcasing a modern instantiation of an RPC-based API.

2. Section 8.2, Section 8.3, and Section 8.4 demonstrate the applicability of message-based APIs using WebSocket, GraphQL, and HTTP-based Interface Exportrers.

3. Section 8.5 demonstrates the applicability of resource-based APIs by generating RESTful APIs, including HATEOAS information and OpenAPI interface definition documents.

8.1 gRPC Interface Exporter

The gRPC Interface Exporter demonstrates the applicability of the Apodini DSL and mechanisms to RPC-based APIs. As described in Section 2.2.1, gRPC is a typical representation of RPC-based APIs by structuring the web API into services and functions. In addition, gRPC features the full range of possible communication patterns between the web service and clients: request-response, service-side streaming, client-side streaming, and bidirectional streaming [138]. While gRPC APIs are usually created by specifying the web API in a Protocol Buffer specification, Apodini allows web service developers to specify the interface in the generalized Apodini DSL. This enables web service interface evolution by allowing a gRPC exporter to be replaced with a different Interface Exportrer over the lifetime of the web service. The definition in the Apodini DSL comes with tradeoffs that are discussed while showcasing the implementation of the gRPC Interface Exporter. The gRPC Interface Exporter for the Swift-based instantiation of the Apodini DSL was developed by Lukas Kollmer as part of the master's thesis investigating *Declarative Development of Interface-Type-Agnostic Web Services* [160].

gRPC is based on the HTTP/2 application layer protocol improving HTTP/1 and HTTP/1.1 with binary encodings, streaming connections, and better support for TLS-based encryptions [38]. gRPC uses these mechanisms to enable the RPC mechanisms such as streaming communication patterns building on top of the stream connections in HTTP/2 [27]. The gRPC Interface Exporter uses the Apodini networking component of the Apodini Swift-based instantiation to facilitate this communication. The Apodini networking component reuses the SwiftNIO networking I/O Swift package[28] to provide communication channels establishing the flow of data from the network interface to the Interface Exporters. In contrast to most other Interface Exporters presented in this chapter, the gRPC exporter extends Apodini networking by adding networking channels based on extension points provided by SwiftNIO. The Apodini networking implementation requires the usage of TLS-based encryption when supporting HTTP/2-based Interface Exporters. Listing 8.1 details the public interface of the gRPC Interface Exporter embedded in the configuration mechanism of an Apodini web service. The gRPC Interface Exporter requires the package name defining the namespace for the generated gRPC Protocol Buffer specifications and a service name for defining the gRPC service that the Handlers registered in the web service belong to.

The Interface Exporter registers several routes which provide access to service definition files and offers gRPC reflection API that tools can use to construct re-

[27] The gRPC project specifies a gRPC over HTTP/2 specification that the gRPC Interface Exporter adheres to https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-HTTP2.md.

[28] SwiftNIO provides an event-driven networking I/O framework to implement web services and clients: https://github.com/apple/swift-nio.

```
@main
struct GRPCExample: WebService {
    var configuration: Configuration {
        HTTPConfiguration(tlsConfiguration:
            TLSConfigurationBuilder(
                certificatePath: "~/cert.pem",
                keyPath: "~/key.pem"
            )
        )
        GRPC(
            packageName: "grpcexample",
            serviceName: "GRPCExample"
        )
    }

    var content: some Component {
        Text("Hello World!")
    }
}
```

Listing 8.1: Swift-based Apodini web service demonstrating the gRPC Interface Exporter. Lines 4-9 detail the `HTTPConfiguration` specifying the certificate and private key used for the TLS-based encryption required by Apodini Networking when using HTTP/2-based Interface Exporters. Lines 10-13 demonstrate an example of the configuration registering the `GRPC` Interface Exporter.

quests at runtime without the need to generate client stubs[29]. The reflection API enables tools like `grpcurl`[30] to test and explore gRPC APIs. The Protocol Buffer specification document of the web service demonstrated in Listing 8.2 shows how the single Handler is transformed to a function in the `GRPCExample` service.

The gRPC exporter iterates over all Handlers defined in the Apodini DSL and transforms each Handler in a gRPC function of the specified service. The Protocol Buffer specification defines that each RPC function must contain a request and response message type [114]. Therefore Handlers such as the `Text` Handler that do not contain any parameters are mapped to the `.google.protobuf.Empty` message type imported at line 4 of the specification in Listing 8.2. The return type is also specified by a message type generated for the `Text` Handler and contains a single scalar value of type `string`. The gRPC exporter generates a response message type for each Handler present in the gRPC service and defines the fields based on the implementation language-specific type layout. Apodini Type Information [31] is the Swift-based type information retrieval library developed as part of the Swift-based Apodini project abstracting the retrieval across different Interface Exporters. The `Text` Handler does not specify a dedicated return type and returns a string

[29]The gRPC reflection API is documented at https://github.com/grpc/grpc/blob/master/doc/server-reflection.md.
[30]The grpcurl tool can be found at https://github.com/fullstorydev/grpcurl.
[31]ApodiniTypeInformation can be found at https://github.com/Apodini/ApodiniTypeInformation.

```
syntax = "proto3";
package grpcexample;

import "google/protobuf/empty.proto";


service GRPCExample {
  rpc GetText(.google.protobuf.Empty) returns
↪   (.grpcexample.TextResponse);
}

message TextResponse {
  string value = 1;
}
```

Listing 8.2: `grpcexample.proto` file generated by the Apodini gRPC Interface Exporter based on the web service demonstrated in Listing 8.1 after removing comments and empty lines. The Protocol Buffer description demonstrates the `GRPCExample` service with a single function named `GetText` that expects an `Empty` argument and returns the `TextResponse` containing a single scalar `string` value.

scalar value. The gRPC exporter, therefore, infers the name of the response message type from the Handler name and appends `Response`. The gRPC message names can also be specified manually using the `HandlerInputProtoMessageName` and `HandlerResponseProtoMessageName` metadata shown in Listing 8.4.

The example in Listing 8.1 and Listing 8.2 demonstrates the capability of the Apodini DSL to transform a primitive web service into a gRPC specification. One limitation demonstrated in Listing 8.2 is the automatic generation of method and request/response names. The gRPC exporter and other exporters address this shortcoming of the generalized DSL and infer names based on the context in the DSL by providing custom modifiers and metadata that can be incorporated in the generation of the web API. Listing 8.3 demonstrates how such a modifier can be used to provide a clearer gRPC method name with additional information that is not affecting the generation in other Interface Exporters:

The Apodini DSL and the gRPC exporter also support mapping streaming communication pattern Handlers with parameters to gRPC functions. Interface Exporters in Apodini rely on a common infrastructure enabling exporters to infer the communication pattern based on the usage of aspects of the Apodini DSL in Handlers. `@ObservedObject` annotations trigger the execution of the `handle` function if the state of the annotated instance changes and can indicate service-side streams. `@State` annotations preserve the state of an object across requests in a single connection, enabling the Handler to preserve state across different client-side streaming requests. While the inference can already provide a best-guess for the Interface Exporters, metadata annotations such as the `Pattern` annotation can override it.

```
@main
struct GRPCStreamingExample: WebService {
    var configuration: Configuration {
        // ...
    }

    var content: some Component {
        BidirectionalHandler()
            .endpointName(fixed: "RepeatTwice")
    }
}
```

Listing 8.3: Line 9 demonstrates the `endpointName` modifier used to improve the output of the gRPC Interface Exporter. The `endpointName` is attached to the `BidirectionalHandler` showcased in Listing 8.4. The output in a Protocol Buffer specification can be observed in Listing 8.5.

```
struct BidirectionalHandler: Handler {
    @Parameter
    var element: String
    @ObservedObject
    var messageQueue = MessageQueue<String>(emitMessage: 2)

    var metadata: Metadata {
        Pattern(.bidirectionalStream)
        HandlerInputProtoMessageName("Input")
        HandlerResponseProtoMessageName("Response")
    }

    func handle() -> Response<String> {
        if $messageQueue.changed, let next = messageQueue.next {
            return .send(next)
        } else {
            messageQueue.enqueue(element)
            return .nothing
        }
    }
}
```

Listing 8.4: `BidirectionalHandler` supports bidirectional streams by returning every String passed to the Handler twice. The Handler contains a custom `@ObservedObject` of the type `MessageQueue` that can `enqueue` messages which triggers the Handler and returns the same element as often as defined in the `MessageQueue` initializer in line 5. The `handle` function checks if the function was triggered due to the observed object and returns the `next` element in the queue or returns nothing and enqueues the `element` passed in as a request to the Handler. The metadata annotation in line 8 overrides the default inference of a service-side stream to a bidirectional stream. Lines 9 and 10 demonstrate the extension points to refine the Protocol Buffer message names for the Handler.

```
syntax = "proto3";
package grpcexample;


service GRPCExample {
  rpc RepeatTwice(stream .grpcexample.Input) returns (stream
  ↪ .grpcexample.Response);
}

message Input {
  string element = 1;
}

message Response {
  string value = 1;
}
```

Listing 8.5: `grpcexample.proto` file generated by the Apodini gRPC Interface Exporter based on the web service demonstrated in Listing 8.3 and the bidirectional streaming Handler defined in Listing 8.4 after removing comments and empty lines. The Protocol Buffer description demonstrates the `GRPCExample` service with a single function named `RepeatTwice` based on the modifier in Listing 8.3 that expects an stream `Input` messages containing the `element` used in the Handler and returns a stream of `Response` messages containing a single scalar `string` value derived from the metadata in Listing 8.4.

The Swift-based Apodini instantiation also includes a web API, middleware, and protocol-independent `Response` type that can be used to indicate connection state-related information to Interface Exporters. The response type can be instantiated using a Swift initializer or a convenience API indicating the connection effect. The `.send` and `.nothing` instantiations indicate that the connection should stay open if the Interface Exporter supports streaming connections. The `.send` response also allows Handlers to include a response while `.nothing` does not return any response to a client. The `.end` response closes the connection without sending a response to a client while `.final` closes the connection after or with sending a response.

Listing 8.4 demonstrates the usage of the `Pattern` metadata. The Listing also demonstrates the Apodini `@ObservedObject` and `@State` properties, and the `Response` type in a Handler that can be exported as a gRPC method supporting bidirectional streams. The `BidirectionalHandler` uses an `@ObservedObject` property wrapper to return every request passed to the Handler twice as detailed in the caption of Listing 8.4. The metadata annotation overrides the default inference of a service-side stream to a bidirectional stream which allows the gRPC exporter to export the Handler as a bidirectional streaming gRPC function.

The gRPC exporter uses the communication pattern information provided by the metadata annotations overwriting the default inference provided by the knowledge sources in the global repository. The exporter creates a networking infrastructure

managing the bidirectional stream and exports this information in the service definition specifications such as the Protocol Buffers file shown in Listing 8.5. The names of the messages used as request and response types of the gRPC method named `BidirectionalHandler` are inferred from the name of the Handler and contain fields for the necessary request and response information.

The gRPC Interface Exporter demonstrates the applicability and extension points of the Apodini DSL to support RPC-based web service interface types. It also highlights shortcomings of the generalized approach of the Apodini DSL, such as the inference of method and message names. It also demonstrates that the usage of an internal DSL uniquely enables the definition of the web service interface that interweaves with the functionality defined in the `handle` function of a Handler.

8.2 WebSocket Interface Exporter

Specified in RFC 6455, *"The WebSocket Protocol enables two-way communication between a client running untrusted code in a controlled environment to a remote host that has opted-in to communications from that code"* [189]. The protocol defines an opening handshake opening up a TCP-based message-based communication, enabling browser-based client applications to establish bidirectional communication with web services [189]. The protocol handshake starts with an HTTP GET message with a connection upgrade header to the WebSocket protocol, which is answered from the web service using an HTTP 101 *Switching Protocol* response [189]. After a successful handshake, the communication partners can transfer information using messages that can be composed of one or more frames [189]. The WebSocket protocol differentiates between several frame types that can be grouped into transmitting UTF-8 encoded textual data, binary data, and control frames to close connections or communicate errors, including WebSocket-specific error codes [189].

The WebSocket protocol provides an instantiation of a modern, message-based communication protocol and mechanism widely used in web-browser-based applications. Web service developers can establish their own WebSocket-based messaging protocol building on top of the foundation provided in RFC 6455 [189]. We have developed a Swift-based WebSocket Interface Exporter that enables Apodini web services to offer a WebSocket-based API supporting all Apodini communication patterns. The WebSocket Interface Exporter was primarily developed by Max Obermeier as part of the Server-Side Swift practical course offered in the winter semester 2020/21 and further improved in the bachelor's thesis *Improving Runtime Performance and Maintainability of the Apodini Server-Side Swift Framework* [202]. The Interface Exporter demonstrates the mapping of the Apodini DSL to a message-based protocol. This section demonstrates Apodini DSL-based mechanisms for refining parameter and connection handling, which other exporters can also use, and

demonstrates how the WebSocket Interface Exporter goes beyond the default extension points to refine the WebSocket-based APIs and interactions further.

The WebSocket exporter offers a single HTTP endpoint that is offered at a path defined in the Apodini DSL web service configuration. Clients can establish a web socket connection by conducting the handshake defined, sending an HTTP request to the specified endpoint [189]. Following the handshake, the web socket uses UTF-8 text-based communication mechanisms encoding messages using JSON serializations. The message-based mechanism is specified in the open-source instantiation of the Apodini framework, including the WebSocket exporter[32]. The message-based mechanisms use a JSON-based structure using contexts initialized with an open context message and property, closed with a close context message, or unexpectedly terminated using an error message. In addition to the three connection-managing message types, client messages and service messages are used to communicate information between the involved communication partners. A context is identified using Universally Unique Identifiers (UUIDs) described by RFC 4122 [170].

The UUID-based context identifier is used to refer to the context in all following messages after a client has established a context by sending a UUID and an identifier of the desired Handler using a JSON-encoded message. The Handler is identified by the endpoint name in the opening context message and is inferred from the Handler position in the tree structure of the Apodini DSL.

```
1 {
2     "context": "123E4567-E89B-12D3-A456-426614174000",
3     "endpoint": ""
4 }
```

Listing 8.6: The JSON serialization of the open context message used in the WebSocket Interface Exporter. The context is established using a client-provided UUID using the string-based representation using hexadecimal values as described by RFC 4122 [170]. The endpoint is identified using the tree structure of the Apodini DSL. An empty endpoint name (`""`) identifies the endpoint at the root of the component tree.

Listing 8.6 demonstrates the JSON structure of an open context message. After establishing the context, client and service messages can be exchanged. The context is closed by sending a JSON formatted message as shown in Listing 8.9 containing the context key and value without any additional information. An error is transmitted using the error message that also contains the context as shown in Listing 8.9 and replaces the `"endpoint"` key with an `"error"` key containing the error message.

Property wrappers such as `@Parameter` and `@Throws` can be extended with additional options built into the Apodini DSL or extended by Interface Exporters.

[32] A more detailed description of the WebSocket Interface Exporter mechanisms can be found in the documentation at https://github.com/Apodini/Apodini/blob/develop/Sources/Apodini/Apodini.docc/Basics/Exporters/WebSocket.md.

```
struct AnimalHandler: Handler {
    @Parameter(.mutability(.constant))
    var species: String
    @Parameter
    var height: Double

    @Throws(
        .badInput,
        reason: "Hight below 0 is not allowed.",
        .webSocketConnectionConsequence(.closeChannel)
    )
    var ageInvalidError: ApodiniError

    func handle() throws -> Response<Animal> {
        guard height >= 0 else {
            throw ageInvalidError
        }
        return .send(Animal(species: species, height: height))
    }
}
```

Listing 8.7: The `AnimalHandler` demonstrates a Handler using the Apodini DSL property options and the usage of `ApodiniErrors`. The mutability property options for the `@Parameter` property wrapper of `.constant` defines that the parameter can only be set once and can not be changed in subsequent requests. The `@Throws` property wrapper demonstrates the usage of an `ApodiniError` in a Handler. The `@Throws` property wrapper is initialized with an error type, a reason, and WebSocket exporter-specific property options for the `@Throws` property wrapper.

The property options API provides an extension point that exporters can use to further refine the API exported by Interface Exporters. Listing 8.7 demonstrates the usage of build in and exporter-specific property options. The `@Parameter`-specific option defining the mutability of a parameter is built into the Apodini core DSL and is used by several exporters such as the WebSocket Interface Exporter. Parameters annotated with a `.constant` property options define that the parameter can only be set once and can not be changed in subsequent requests when using a client-streaming or bidirectional streaming pattern. The default value for all parameters not annotated with the mutability option is the `.variable` option indicating that the parameter can be updated with every subsequent request.

The `@Throws` property wrapper can be used to define `ApodiniErrors` that are thrown in a Handler and should be exposed on the web API level. The property wrapper requires an `ErrorType` which can be mapped to different protocol-specific error codes or error types. Addition information such as a public reason as well as an internal description can be used to refine the `ApodiniError` further. The `@Throws` property wrapper also enables the usage of property options which can be used to extend the `ApodiniError` with additional context that can be used by the Interface Exporters when encountering an error.

Client:

```
{
    "context": "123E4567-E89B-12D3-A456-426614174000",
    "parameters": {
        "species": "Apus apus",
        "height": 42
    }
}
```

Web Service:

```
{
    "content": {
        "height": 42,
        "species": "Apus apus"
    },
    "context": "123E4567-E89B-12D3-A456-426614174000"
}
```

Client:

```
{
    "context": "123E4567-E89B-12D3-A456-426614174000",
    "parameters": {
        "height": 43
    }
}
```

Web Service:

```
{
    "content": {
        "height": 43,
        "species": "Apus apus"
    },
    "context": "123E4567-E89B-12D3-A456-426614174000"
}
```

Listing 8.8: Message exchange with the Handler shown in after the context has been established in Listing 8.7 with a message as shown in Listing 8.6 using a UUID of `123E4567-E89B-12D3-A456-426614174000`. The client only sets the `"species"` in the first message and does not need to provide the parameter in subsequent messages as it is a `.constant` parameter. The responses from the web service demonstrate the message response structure of the WebSocket Interface Exporter mapping the Apodini Handler to the WebSocket-based communication mechanism.

The usage of WebSocket Interface Exporter Specific options is demonstrated in Listing 8.7. The WebSocket Exporter maps the information from the Apodini DSL found in the semantic model and knowledge sources to a WebSocket-based API using JSON messages. Two exemplary message exchanges for the Handler in Listing 8.7 are demonstrated in Listing 8.8 and Listing 8.9.

Listing 8.8 shows the message exchange after the context has been established with the Handler shown in Listing 8.7. The message exchange demonstrates the effects of the mutability parameter option. Trying to change constant parameters in subsequent messages results in an error from the Apodini Interface Exporter sub-

```
Client:
1 {
2     "context": "123E4567-E89B-12D3-A456-426614174000",
3     "parameters": {
4         "height": -1
5     }
6 }
Web Service:
1 {
2     "context": "123E4567-E89B-12D3-A456-426614174000",
3     "error": "The operation couldn't be completed. Hight below 0 is not
  ↪   allowed."
4 }
Connection Closed, Error Code: 1011.
```

Listing 8.9: The message exchange follows the messages exchanged in Listing 8.8. The client sends a message containing a negative height which throws an error as shown in the `handle` function implementation in Listing 8.7 (Line 16). The WebSocket exporter transforms the Apodini DSL `@Throws` property wrapper-based information into an error message and closes the connection with an error.

system used by the Interface Exporters. The WebSocket Interface Exportrer directly embeds this mechanism in the JSON-based communication mechanism by providing the optimization that parameters do not need to be provided with subsequent messages. Interface Exporters can specify this behavior based on the protocol constraints of the respective middleware or protocol type.

Listing 8.9 demonstrates how an error message is returned from an WebSocket Interface Exporter-based API and incorporates the WebSocket exporter-specific property options. The usage of property options and Apodini Errors demonstrates how web API information and errors are described in the Apodini DSL. The aspects of the DSL can be interpreted and used by Interface Exporters supporting different web API protocols to export a web API. Extension points such as property options can be used to extend the DSL components with additional context that one or more Interface Exporters can use. The Apodini DSL can be mapped to a message-based API structure demonstrated by a WebSocket-based communication mechanism using JSON-based messages. The following sections describe other message-based Interface Exporters such as the GraphQL and HTTP exporter, further demonstrating the message-based interface evolution capabilities of the Apodini ecosystem.

8.3 GraphQL Interface Exporter

As described in Section 2.2.1 and Figure 3.3 (page 52), GraphQL is a message-based web API type. The GraphQL specification defines a language that can be used to query web services for information, independent of a storage system or programming language [93]. The GraphQL specification follows several design principles,

```swift
@main
struct GraphQLExample: WebService {
    var configuration: Configuration {
        GraphQL(enableGraphiQL: true)
    }

    var content: some Component {
        Greeter()
            .endpointName("greet")
        SaveCountry()
            .operation(.update)
            .endpointName(fixed: "saveCountry")
    }
}
```

Listing 8.10: The `GraphQLExample` web service demonstrates the usage of the Apodini DSL `endpointName` and `operation` modifiers and the `GraphQL` configuration of the GraphQL Interface Exporter. The GraphQL exporter configuration allows the configuration of the GraphQL endpoint, and if the GraphiQL IDE described and displayed in Figure 8.1 should be presented.

including a product-centric approach that encourages the usage of hierarchies in GraphQL queries to represent the data that is requested [93]. GraphQL APIs offer strongly typed interfaces that are defined by web services that clients can introspect to decide how to consume the published API [93]. The GraphQl specification serves as a reference for tools and libraries conforming and benefitting to the GraphQL ecosystem [93]. The GraphQL exporter demonstrates how the concepts of the Apodini interface type and web API type agnostic DSL can be used to expose GraphQL APIs. Similar to other Interface Exporters, the GraphQL exporter demonstrates the web service interface evolution-related advantages of the Apodini DSL. The Interface Exporter subsystem and the GraphQL exporter enable the adoption of a new web API type with minimally invasive changes in the web service configuration as demonstrated in Listing 8.10.

The GraphQL Interface Exporter for the Swift-based instantiation of the Apodini DSL was developed by Lukas Kollmer as part of the master's thesis investigating *Declarative Development of Interface-Type-Agnostic Web Services* [160]. The GraphQL Interface Exporter of the Swift-based Apodini DSL instantiation reuses the GraphQL Swift Package[33]. The GraphQL Swift Package contains a core implementation of the GraphQL language for Swift-based web services. The GraphQL Interface Exporter uses the semantic model-based information and additional context provided by knowledge sources to map the Apodini Handlers to GraphQL queries and mutations. The Apodini-DSL operation modifier and metadata are used to identify if a Handler is a mutation or query. Handlers using the default read operation are clas-

[33]The GraphQL Swift Package is part of the GraphQLSwift GitHub organization: https://github.com/GraphQLSwift/GraphQL.

sified as queries, while create, update, and delete operations are interpreted as mutations. The `operation` modifier demonstrated in Listing 8.10 is also used by other Interface Exporters such as the HTTP and REST interface to specify HTTP methods or when automatically generating endpoints names for Interface Exporters.

```
1  class MoviesHandler: Handler<List<SimplyfiedMovie>> {
2      private val movieModel by environment { movieModel }
3  
4      override suspend fun CoroutineScope.handle(): List<SimplyfiedMovie>
   ↪   {
5          return movieModel.movies.values.map { SimplyfiedMovie(it) }
6      }
7  }
```

Listing 8.11: The `MoviesHandler` demonstrates the dependency injection mechanism in the Kotlin-based Apodini DSL. The `environment` property delegate loads the `movieModel` from the Apodini environment. The `MoviesHandler` is used in Listing 8.18 (page 149) to demonstrate the RESTful Interface Exporter in Section 8.5.

```
1  extension Application {
2      var countryStore: CountryStore {
3          guard let store = self.storage[\Application.countryStore] else {
4              self.storage[\Application.countryStore] = CountryStore()
5              return self.countryStore
6          }
7          return store
8      }
9  }
10 
11 struct SaveCountry: Handler {
12     @Environment(\.countryStore) var countryStore: CountryStore
13 
14     @Parameter var country: String
15 
16     func handle() -> String {
17         countryStore.insert(country)
18         return countryStore.countries
19     }
20 }
```

Listing 8.12: The `SaveCountry` Handler demonstrates a mutating Handler that changes the state of the web service. The `@Environment` property wrapper ensures access to shared information in Apodini web services and offers a dependency injection mechanism to access shared information in Handlers. The extension to the Apodini `Application` type in lines 1-9 demonstrate an implementation of using the environment using the `Application storage` mechanism and a predefined `CountryStore` type. The `CountryStore` instance is injected in the Handler using the `@Environment` property wrapper in line 12 and used in the `handle` function to insert a new country (line 17) and then return all countries (line 18).

Listing 8.10 also demonstrates the usage of the GraphQL Interface Exporter in the Swift-based Apodini DSL. The GraphQL Exporter maps the `Greeter` Handler to a query in the GraphQL API and the `SaveCountry` Handler to a mutation in the GraphQL API. The `SaveCountry` Handler is a demonstration of a Handler modifying the internal state of the web service.

While Handlers can use techniques of the hosting programming languages to share state in the web service, using the Apodini Environment enables dependency injection mechanisms facilitated by the Apodini subsystem. Listing 8.11 demonstrates the usage of the Apodini environment dependency injection mechanism for the Kotlin-based instantiation. The Swift-based instantiation of the Apodini DSL uses the `@Environment` property wrapper as demonstrated in Listing 8.12 to enable the dependency injection mechanism.

Listing 8.12 shows the `SaveCountry` and surrounding Swift-based code using the Apodini framework to enable dependency injection. The GraphQL Interface Exporter also offers the possibility to expose the GraphiQL browser-based GraphQL IDE[34] that can be used to explore and test out the offered GraphQL API. The IDE enables the creation of queries and mutations that can be sent to the GraphQL API offered by the web service. GraphiQL also offers a documentation explorer, investigating the structure of the GraphQL Schema offered by the web service. Figure 8.1 displays the GraphiQL IDE for the Apodini DSL-based web service using the GraphQL exporter shown in Listing 8.12.

The GraphQL Interface Exporter demonstrates the mapping of the Apodini DSL to a GraphQL-based web API. The mechanism of getting started with a new web API type by adding the Interface Exporter to the web service and refining the output using Apodini DSL-based or custom Interface Exporter specific modifiers and metadata demonstrates the web service interface evolution related capabilities. We must also acknowledge that the generalized approach of specifying the web service functionality and interface in the Apodini DSL comes with tradeoffs. One of the main advantages of GraphQL is the reduction in under and over-fetching compared to more static web API types by allowing the client to precisely specify what information is requested. GraphQL-based APIs, therefore, offer complex and nested return types of queries and mutations that can be several layers deep. While the GraphQL implementation allows querying fields of the returned types of the Handlers, a deeper nesting would require more Apodini-Infrastructure specific to the GraphQL Interface Exporter.

[34]The GraphiQL IDE is part of the GraphQL open-source project managed by the GraphQL Foundation: https://github.com/graphql/graphiql.

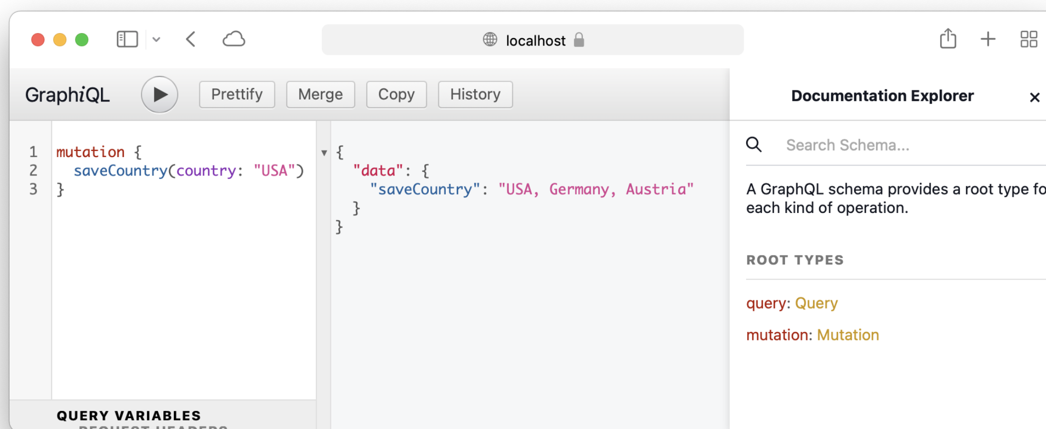


Figure 8.1: The GraphiQL IDE offered by the GraphQL Interface Exporter. The GraphiQL IDE is used to send a request to the `saveCountry` mutation as shown on the left side of the IDE. The mutation is mapped to the `SaveCountry` Handler shown in Listing 8.10 and Listing 8.12. The data returned by the GraphQL request is displayed in the middle, showing that the new country is added to the existing list of countries. The right side of the IDE displays the documentation explorer enabling a web service client to interactively explore the queries and mutations offered by the web service.

8.4 HTTP Interface Exporter

The HTTP Interface Exporter offers a JSON-based web API using HTTP messages multiplexed using the HTTP request URI. While the structure of the offered API is similar to the web API provided by the RESTful Interface Exporter, it does not emit HATEOAS information typically offered by RESTful APIs or is constrained to a stateless request-response communication pattern [96]. The HTTP exporter demonstrates the usage of the Apodini Interface Exporter subsystem and APIs to offer an HTTP-based API that supports all communication patterns using an HTTP/1 compatibility layer. The compatibility layer maps the request-response pattern supported by HTTP/1-based connections to all supported communication patterns in Apodini. The HTTP exporter can be interpreted as a message-based and resource-based exporter, depending on the modeling of the application domain to the HTTP endpoints. An essential insight of this section is the mechanism that maps the tree structure of the Apodini DSL to HTTP endpoints. This mapping highlights the HTTP-enabling Apodini mechanism to address Knowledge Question 2 and Design Problem 1.

In addition to injecting shared state into Handlers using the `@Environment` Apodini property, the `@Binding` property wrapper enables dependency injection for Apodini properties themselves. Bindings enable dependency injection from outside of the Handler as they can be bound to a constant value, a `@Parameter` or `@Environment` property of the type associated with the binding. Bindings are injected by the initializer of the Handler using the binding where the Handler is used in the Apodini DSL. Listing 8.13 demonstrates the usage of the `@Binding` Apodini property in a Handler in the Swift-based Apodini DSL. The Swift-based web service in Listing 8.15 shows how the binding is injected from within the Apodini-DSL component hierarchy.

Similar to other Interface Exporters, the HTTP exporter uses property options to enable web service developers to refine the API with HTTP-specific context further. The `.http` options allow developers to annotation a parameter with the `.body`, `.path`, and `.query` option. Body parameters are sent using the HTTP body and are expected to be encoded in an encoding supported by the respective exporter. Path parameters are encoded as part of the URI path of an HTTP request. Query parameters use the URI schema to append query parameters at the end of the URI sent in HTTP requests. Listing 8.13 demonstrates the usage of the HTTP options in a Swift-based Handler.

```swift
struct Greeter: Handler {
    @Parameter(.http(.query)) var times = 1
    @Binding var name: String

    func handle() -> String {
        let capacity = (9 + name.count) * times
        var greeting = String(reservingCapacity: capacity)
        for _ in 0..<times {
            greeting.append("Hello, \(name)! ")
        }
        return greeting
    }
}
```

Listing 8.13: The `Greeter` Handler demonstrates the usage of the HTTP-specific options for the `@Parameter` Apodini property. The `.http(.query)` option indicates that the `times` property with a default value of `1` should be exported as a query parameter if the Interface Exporters use the HTTP protocol and the communication protocol or middleware type supports this distinction. The `@Binding` property wrapper enables passing in the Apodini property used to retrieve the name from the DSL-Context as demonstrated in Listing 8.15. The `handle` function then returns the greeting as often as defined by the `times` parameter.

Handlers can use the `Connection` type stored in the Apodini environment to react to connection-related events and retrieve lower-level information about the request. The `Connection` type's `information` property includes additional infor-

mation that can be mapped to HTTP headers or other context provided by the communication protocol. Handlers can also use instances of the `Connection` type to retrieve the remote address of the request and access to the Apodini `Request` type to try to retrieve parameters manually. This mechanism is not generally encouraged as it limits the Interface Exporter subsystem's ability to parse interface specification-related information from the Handlers. The `Connection` type's `state` property defines the current state of the connection with the `.open` case representing a normal open connection. When the connection is in the `.end` state, the connection will be closed with the ability to send one last response to the client. The connection is in the `.close` state if the connection was terminated without the chance to send one last response. Listing 8.14 demonstrates the usage of the `Connection` type instance stored in the `@Environment`.

```
1  struct CollectingGreeter: Handler {
2      @Parameter var name: String
3  
4      @Environment(\.connection) var connection
5  
6      @State var names: [String] = []
7  
8      func handle() -> Response<String> {
9          switch connection.state {
10         case .open:
11             names.append(name)
12             return .nothing
13         case .close:
14             return .final("Hello, \(names.joined(separator: ", "))!")
15         case .end:
16             return .nothing
17         }
18     }
19 }
```

Listing 8.14: The `CollectingGreeter` demonstrates the usage of the `Connection` type stored in the Apodini environment. The `CollectingGreeter` uses the `connection.state` and a Swift switch-statement to append the name to the collection of names that is preserved between executions using the `@State` Apodini property when a request arrives. The Handler returns a greeting containing all names once the connection is ended from the client-side after all names have been sent. The Apodini Interface Exporter subsystem can infer that the Handler is suitable for a client-side stream based on the usage of the `@State` Apodini property.

The Apodini-DSL uses components to generate a tree structure with the web service type at the root and Handlers forming the leaves of the tree. The DSL component class diagram (Figure 7.1, page 112) in Section 7.1 demonstrates the composite of components as tree nodes as manifested in the Apodini DSL instantiations. Web service developers create components that structure the DSL and only contain the `content` property in the Swift-based DSL or the `ComponentBuilder` of the Kotlin

DSL to describe the components and Handlers that are part of the custom components. The DSL also provides a group component that can be used to structure the DSL levels and provide additional information for each level, such as URI path components or modifiers that are applied to all Handlers further down the DSL tree. The Swift-based implementation uses the `Group` type to structure the DSL in the Swift result builders. The Kotlin-based implementation uses `group` functions to achieve the same structure in the `ComponentBuilder`.

Apodini also enables web service developers to embed path parameters in the component hierarchy of `Groups` by generating the URI path identifying the HTTP endpoints. Inference Exporters such as the HTTP exporter add parameters within a Handler annotated with the `.http(.path)` option at the end of the URI path. Using bindings and the `@PathParameter` Apodini property within the Swift-based Apodini DSL allow developers to refine the placement of path parameters further. The Kotlin-based version of the DSL enables the usage of the `pathParameter` convenience function generating a parameter with an HTTP-path option. Listing 8.15 demonstrates the usage of `Groups` and a `@PathParameter` in the Swift-based DSL. Listing 8.15 demonstrates the same concepts in the Kotlin-based Apodini DSL.

```
1  @main
2  struct HTTPExample: WebService {
3      @PathParameter var name: String
4
5      var configuration: Configuration {
6          HTTPConfiguration(/* ... */)
7          HTTP()
8      }
9
10     var content: some Component {
11         Text("Hello World!")
12         Group("greet") {
13             CollectingGreeter()
14             Group($name) {
15                 Greeter(name: $name)
16             }
17         }
18     }
19 }
```

Listing 8.15: The `HTTPExample` web service using the `HTTP` exporters demonstrates the usage of a string-based `@PathParameter` in the Swift-based component and Handler hierarchy. The `Group` component can be used to group Handlers and components that are defined in the result builder of the `Group` initializer (lines 13-18 and lines 15-17). Web service developers can also pass several path components to a Group. Strings and instances of the `PathParameter` type conform to the protocol and can be passed to the `Group` initializer. The `PathParameter` instance can be accessed from a wrapped property using the projected value mechanism in the Swift language using the `$` sign (lines 13 and 15).

```
object HTTPExample : WebService {
    private val name = pathParameter()

    override fun ConfigurationBuilder.configure() {
        // ...
    }

    override fun ComponentBuilder.invoke() {
        text("Hello World")
        group("greet", name) {
            +Greeter(name)
        }
    }
}
```

Listing 8.16: The `HTTPExample` web service demonstrates the usage of a string-based path parameter in the Kotlin-based component and Handler hierarchy. The Kotlin-based instantiation uses the convenience `pathComponent` to generate a parameter with an HTTP-path option. The parameter can then be passed to the group function of the `ComponentBuilder`. The example demonstrates passing multiple path components to a group (line 10) which is also possible in the Swift-based instantiation. Handlers are preceded by the unary plus operator to register the Handler in the semantic model.

When starting the web service, the HTTP exporter maps the semantic model containing the paths and all parameters to URIs where the Handlers are reachable using HTTP requests. Clients can access the Endpoint by sending an HTTP request to the web service, e.g., using `curl`[35] as demonstrated in Listing 8.17.

```
$ curl "https://localhost/greet/Paul?times=3"
"Hello, Paul! Hello, Paul! Hello, Paul! "
```

Listing 8.17: The `curl` command in line 1 sends out a request to the Swift-based Apodini web service running locally and configured to use TLS with a trusted certificate. The web service described in Listing 8.15 responds with an HTTP response of status code 200 containing the string detailed in line 2.

The Swift-based HTTP exporter demonstrates that Handlers and web services developed in the Apodini DSL can be mapped to an HTTP-based message-based API that can also be used to build resource-based APIs. The Apodini DSL components and features such as groups, path parameters, and HTTP-specific parameter options found in the Swift-based DSL are used by the HTTP exporter to support web service APIs. While the HTTP Interface Exporter only offers an HTTP/1-based web API, extending the HTTP exporter to support native streaming behavior without a compatibility layer is promising future work, further demonstrating the extensibility of the Apodini DSL and Interface Exporter subsystem.

[35] `curl` is an open-source command-line tool: https://github.com/curl/curl.

8.5 RESTful Interface Exporter

The RESTful Interface Exporter uses the Interface Exporter subsystem, knowledge sources, and the semantic model to export a resource-based web API based on the web API and interface type independent Apodini DSL. The RESTful exporter is instantiated for the Kotlin-based and Swift-based versions of the DSL. The exporter emits HATEOAS information typically offered by RESTful APIs and is constrained to a stateless request-response communication pattern [96]. Leonard Richardson describes four maturity levels[36] for fulfilling the RESTful constraints in HTTP, starting from level 0 with no RESTful characteristics such as XML-RPC and SOAP web services. Level 1 maturity is reached when an API uses multiple URIs to offer functionality but only uses one HTTP method for all requests[37]. Level 2 indicates a proper mapping of resources and operations on these resources to URIs and suiting HTTP methods[38], and the usage of caching and media-type mechanisms [240]. Level 3 is reached when hypermedia is used to describe resources and their interconnections[39]. Rodríguez et al. conducted a study of RESTful APIs and demonstrated that the biggest part of the self-proclaimed RESTful APIs in their collected dataset reached maturity level 2 while most APIs reached level 1 and only a few APIs recached level 3 [240]. The study concludes that a large percentage of APIs are not taking advantage of the capabilities of the HTTP protocol, and the support for level 3 hypermedia support is limited due to a missing standard for presenting and describing hypermedia information for RESTful APIs in responses [240]. Petrillo et al. also demonstrate that of several RESTful APIs offered by cloud providers, more than half of the collected best practices, while not fully reaching level 3 maturity and not following a large percentage of bests practices [216]. Similar findings have also been reported by Kotstein and Bogner when surveying participants about the relevance of RESTful API design rules that indicated that not all current RESTful API design rules are perceived as essential to implement [161]. The Delphi Study with industry experts showed that a large proportion of rules to reach maturity level 2 are perceived necessary while rules to reach maturity model level 3 are not perceived as essential to invest resources to achieve level 3 for the industry applications [161]. The complexity of generating the hypermedia information and a lack of common standards to represent the hypermedia information probably contribute to the lack of interest in investing resources to develop RESTful APIs emitting hypermedia information.

[36] Leonard Richardson presented the maturity heuristics as part of a talk titled *Justice Will Take Us Millions Of Intricate Moves* in November 2008: https://www.crummy.com/writing/speaking/2008-QCon/act3.html.
[37]See footnote 36.
[38]See footnote 36.
[39]See footnote 36.

The RESTful Interface Exporter demonstrates the advantage of the internal DSL-based approach of combining the interface description with implementing the functionality integrated into the parsed semantic model. The approach of generating the RESTful interface based on the abstract DSL allows the exporter to ensure that REST design rules such as the proper usage of URI paths, HTTP Methods, and other design best practices are adhered to. New best practices or design rules can be centrally implemented in the RESTful Interface Exporter. All web services using the exporter can benefit from the added RESTful API conformance. The Interface Exporters can use the information of the semantic model to automatically generate relationships between Handlers that are mapped to HTTP endpoints. This information can be used to automatically provide a best-effort approach to generating hypermedia information and other context needed to reach the Richardson maturity model level 3. Different DSL annotations can further refine the relationship information, as demonstrated by the Swift-based Apodini implementation. The RESTful architectural style does not define an explicit mapping to HTTP or how the uniform interface is structured in detail, including the representation of hyperlinks in responses [96]. The *REST API Design Rulebook* by Mark Massé introduces the Web Resource Modeling Language (WRML) that can be used to define JSON-based representations of hypermedia information in responses [187]. The JSON-based representation builds a uniform structure containing links, resource descriptions, and other metadata information needed for a client to process and act upon the hypermedia information [187]. The Hypertext Application Language (HAL) also demonstrates an approach to defining a resource serialization format containing hypermedia information but never moved past the stage of an IETF Internet-Drafts that expired 2016 [152, 131]. The RESTful Interface Exporters for the Kotlin and Swift-based DSL both build upon aspects of the WRML and HAL-based representations. While the hypermedia information currently emitted by the Interface Exportrers is limited to hypermedia links, the semantic model and the context provided by the Apodini DSL provide the potential for future work extending responses to provide more detailed hypermedia information.

Listing 8.18 demonstrates a Kotlin-based Apodini web service that offers a RESTful API to get information about movies, the participating cast and crew, and filming locations of the movies. The web service demonstrates a typical resource-based structure that can be mapped to a RESTful API but is also compatible with the other Interface Exporters demonstrated in Chapter 8. The web service in Listing 8.18 offers a web service interface to get information about movies. The RESTful Interface Exporter parses the information found in the Kotlin-based semantic model to generate REST API endpoints. Similar to the Swift-based HTTP Interface Exporter in Section 8.4, the exporter uses the DSL tree structure and path parameters to generate URIs for the endpoints.

```kotlin
object MoviesWebService : WebService {
    private val movieId = pathParameter()

    override fun ConfigurationBuilder.configure() {
        use(REST())
    }

    override fun ComponentBuilder.invoke() {
        text("Welcome to the movies API")
        group("movies") {
            +MoviesHandler()
            group(movieId) {
                +MovieHandler(movieId)
                group("cast") {
                    +MovieCastHandler(movieId)
                }
                group("crew") {
                    +MovieCrewHandler(movieId)
                }
                group("locations") {
                    +MovieLocationsHandler(movieId)
                }
            }
        }
    }
}
```

Listing 8.18: The `MoviesWebService` demonstrates a Kotlin-based Apodini web service that uses the `REST` Interface Exporter as configured in line 5. The component and Handler structure of the web service consists of a `Text` Handler providing a greeting at the root of the API (line 9). The other Handlers provide information about movies. As noted in Section 8.4 `movieId` is using a path parameter to inject the parameter in the DSL hierarchy indicated by the `groups`. All movies are returned by the `MoviesHandler` while the `MovieHandler` expects a `movieId` that identifies a movie and returns more information about each movie then the movies list returned by the `MoviesHandler`. The `MoviesCastHandler`, `MoviesCrewHandler`, and `MoviesLocationsHandler` return details about the cast, crew, and filming locations of the movie identified by the `movieId`.

The exporter uses the structure in the semantic model to define relationships between the different Handlers. These relationships are provided to the RESTful Interface Exporter that transforms the relationships into links that are transmitted to offer hypermedia information in responses. Listing 8.19 contains the body of an HTTP response to the root of the web service shown in Listing 8.18. The links in the response follow a simplified adaption of the WRML and HAL-based representations to demonstrate the usage of hypermedia information. The Interface Exporter automatically generates the links based on the Apodini-Internal path representation. Links containing path parameters are represented using the URI template syntax defined in RFC 6570 [119]. Listing 8.20 demonstrates the usage of path parameters in the links section of a HATEOAS-based JSON response.

```
{
    "data": "Welcome to the movies API",
    "_links": {
        "self": "http://localhost",
        "movies": "http://localhost/movies"
    }
}
```

Listing 8.19: JSON response returned when sending an HTTP GET request to / (root) of the web service shown in Listing 8.18. The response details the text returned by the `Text` Handler as well as hypermedia links including the `self` link and a link to the endpoint for retrieving movies found at `/movies`.

```
{
    "data": [
        {
            "description": "...",
            "id": 1,
            "title": "Schindler's List"
        }
    ],
    "_links": {
        "self": "http://localhost/movies",
        "movie": "http://localhost/movies/{id}"
    }
}
```

Listing 8.20: JSON response returned when sending an HTTP GET request to `/movies` of the web service shown in Listing 8.18. The response contains a list of movies with reduced information. The `_links` section contains the `self` link and a link to a movie using the URI template syntax.

```
{
    "cast": [2],
    "crew": [1],
    "description": "...",
    "id": 1,
    "locations": [1],
    "title": "Schindler's List",
    "_links": {
        "self": "http://localhost/movies/1",
        "cast": "http://localhost/movies/1/cast",
        "crew": "http://localhost/movies/1/crew",
        "locations": "http://localhost/movies/1/locations"
    }
}
```

Listing 8.21: JSON response returned when sending an HTTP GET request to `/movies/1` of the web service shown in Listing 8.18. The response contains the details of the movie with the id `1`, including the movie identifier, title, description, and identifiers for the cast, crew members, and filming locations. The `_links` section contains links to more information about the cast, crew, and filming locations.

```
{
    "data": [
        {
            "id": 2,
            "name": "Liam Neeson",
            "biography": "..."
        }
    ],
    "_links": {
        "self": "http://localhost/movies/1/cast",
        "person": "http://localhost/persons/{entityId}"
    }
}
```

Listing 8.22: JSON response returned when sending an HTTP GET request to `/movies/1/cast` of the web service shown in Listing 8.23. The response contains a list of all casts members including some information about each cast member such as a name and the biography. The `_links` section of the JSON-based response contains the `self` link as well as a link to the `EntityHandler` in Listing 8.23 (line 30) using the relationship modifiers described in the figure caption.

If the context provided by requests defines a path parameter, the link formatting automatically fills in any template values in the URI description. The prefilled URI templates enable client applications to rely on the web service API for further information about the requested resource. The HATEOAS mechanism enables more flexible clients and potential web API evolution stability by not relying on predetermined URI paths for subsequent requests. Listing 8.21 presents a request to the `MovieHandler` in Listing 8.18 that requires a path parameter to be specific.

The functionality of the Swift-based Apodini DSL and the RESTful Interface Exporter exceeds the capabilities of the Kotlin-based implementation. The Swift-based Apodini DSL offers additional relationship-related annotation and customization mechanisms that can be used to further refine the relationship information used by the RESTful Interface Exporter. The metadata information and modifiers changing the metadata information enable the addition of new relationships, hiding automatically generated relationships and further refining information about automatically generated relationships between Handlers. The mechanisms and implementations were developed as part of the Server-Side Swift practical course and the Bachelor's Thesis of Andreas Bauer titled *Requirements Traceability for Web Services* [34]. The metadata annotations enable the definition of relationship information on the `Content` type and Handler level of the Apodini DSL. Using the `Relationship` type also enables the definition of relationships between components as part of the component definition in the `content` property of a web service or component.

Listing 8.23 demonstrates the usage of the `Relationship` type to create a relationship between the Handler returning the cast of a movie and the Handler providing additional information about a person, including crew and cast members.

```
@main
struct MoviesWebService: WebService {
    // The @PathParameters and location relationship ...
    private static let personRelationship = Relationship(name: "person")

    var configuration: Configuration {
        REST {
            OpenAPI()
        }
    }

    var content: some Component {
        Text("Welcome to the movies API")
        Group("movies") {
            AllEntitiesHandler(\.movies)
                .response(SimplyfiedMovieResponseTransformer())
            Group($movieId) {
                EntityHandler($movieId, movieModelKeyPath: \.movies)
                Group("cast") {
                    AllMovieRelatedEntitiesHandler(/* ... */)
                        .relationship(to: Self.personRelationship)
                }
                // Crew and locations ...
            }
        }
        // Locations ...
        Group("persons") {
            AllEntitiesHandler(\.persons)
            Group($personId) {
                EntityHandler($personId, movieModelKeyPath: \.persons)
                    .destination(of: Self.personRelationship)
            }
        }
    }
}
```

Listing 8.23: The `MoviesWebService` demonstrates a Swift-based Apodini web service that uses the `REST` Interface Exporter and exports an OpenAPI specification as configured in lines 7-9. The web service offers the same functionality as the web service in Listing 8.18 and additionally offers Handlers for more information about persons and locations (lines 26-34). The declaration of the `@PathParameters` is omitted in the figure (line 3). The location and crew Handlers and Groups are omitted in the figure and replaced with comments to reduce the size of the figure (line 23 and line 26). The web service demonstrates the usage of manually specifying relationships as done using the `personRelationship`. The destination of the relationship is defined in line 31. The person-related Handlers are modified to use this relationship (line 21) which the REST exporter uses to generate hypermedia links.

The relationship information is then combined with the automatically inferred relationships and used by the REST exporter to populate the links section of HTTP responses. Listing 8.22 demonstrates a response from the cast endpoint showing the additional person link using the URI template syntax.

The REST exporter demonstrates the applicability of the Apodini DSL to resource-based APIs while allowing web service interface evolution using the extensible Interface Exporter subsystem. The section shows how using the exporter-based infrastructure generates RESTful APIs by automating aspects, including the generation of hypermedia information.

8.5.1 OpenAPI Document Generation

Section 2.2.1 describes how web service description language or interface description languages are used to describe web APIs to generate client stubs or compose web APIs automatically. Even though REST's original indentation to be discoverable and traversable using HATEOAS information, interface definition languages such as OpenAPI are commonly used to describe RESTful web APIs. The Swift-based OpenAPI exporter builds on top of the infrastructure provided by the REST exporter to export an OpenAPI specification describing the RESTful API of a web service. The OpenAPI exporter iterates over the semantic model and information shared with the REST exporter to define the HTTP endpoints, parameter types, and response types manifested in an OpenAPI specification. The specification is used by other subsystems such as the Apodini Deployer subsystem to enable the deployment in FaaS-based environments to generate API gateways.

The OpenAPI specification can be downloaded at a configurable endpoint to provide it to web service consumers. The exporter additionally provides the option to present a Swagger UI[40]-based online documentation reading in the OpenAPI specification. Figure 8.2 demonstrates the generated user interface that enables web service clients to explore the interface and test out requests to the web service.

[40]The Swagger UI project is an open-source project allowing web service developers to integrate a rendered OpenAPI documentation in web services: https://github.com/swagger-api/swagger-ui.

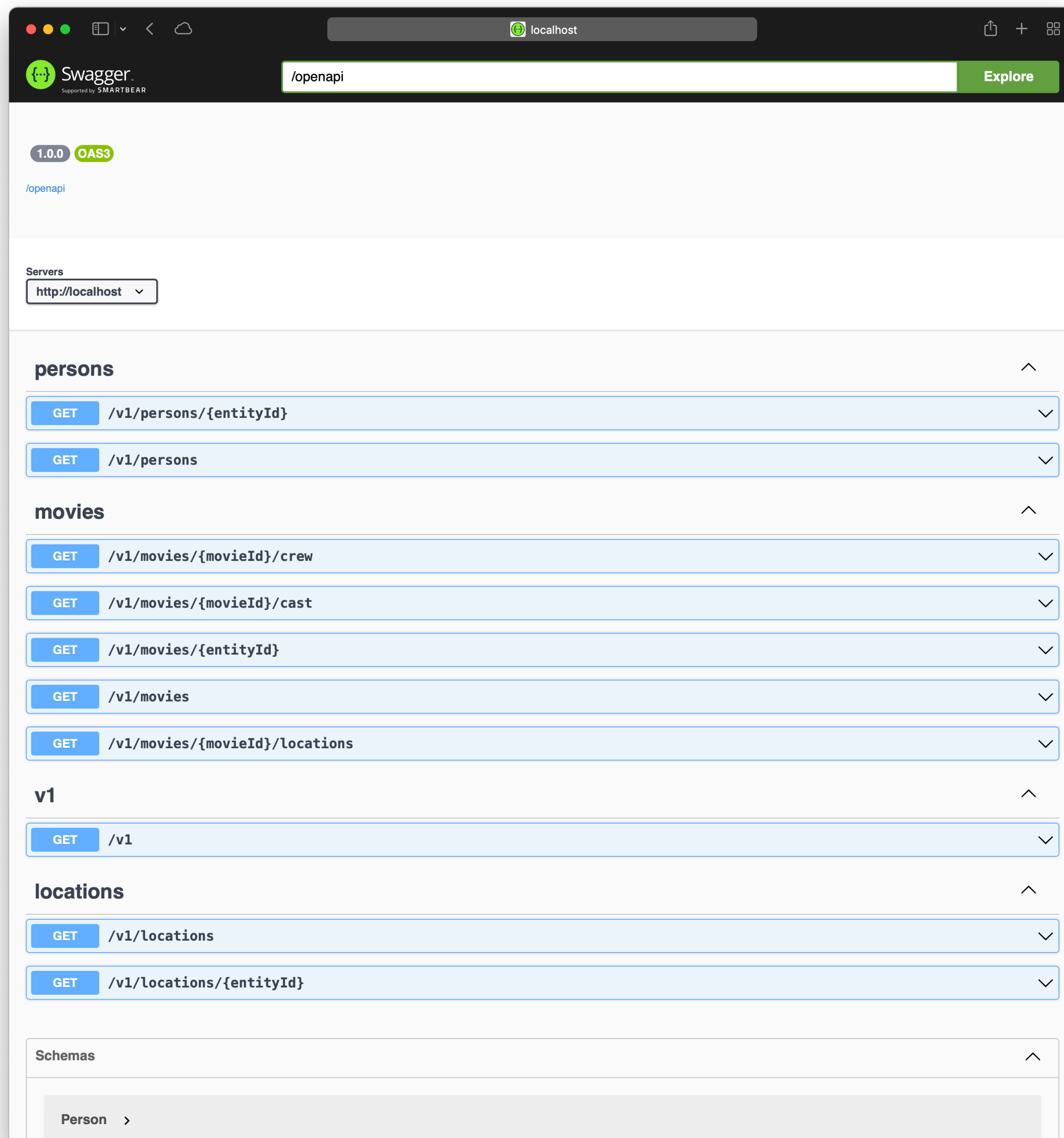


Figure 8.2: The Swagger user interface shows a graphical representation of the OpenAPI specification generated by the OpenAPI exporter. The OpenAPI specification is exported from the configuration and structure of the web service demonstrated in Listing 8.23. The Swagger UI provides a graphical representation to explore the API and test out requests to the web service.

8.6 Summary

The Interface Exporters and experiments in this chapter demonstrate the applicability of the Apodini DSL and Interface Exporter mechanisms to different web service interface types and web API types. The Apodini DSL, the Interface Exporter subsystem, and Interface Exporter instantiations fulfill the tasks set out in Technical Research Goal 1 and Design Problem 1.

> **Technical Research Goal 1:**
> Design artifacts supporting web service API type agnostic development to enable web service interface evolution.

Each web service interface type is instantiated by at least one Interface Exporter. The Interface Exporters validate the DSL and subsystem functionality by demonstrating strongly specified web API types such as gRPC and GraphQL and more loosely specified message structures, e.g., in the WebSocket instantiation. All exporters are added, removed, and modified in minimally invasive Apodini web service configuration changes.

> **Knowledge Question 2:**
> Does the Apodini DSL empower web service interface- and web API type-independent web service development?

The gRPC, WebSocket, and HTTP Interface Exporter sections demonstrate the applicability of the Apodini DSL for communication patterns beyond the request-response pattern. Apodini DSL features such as the state, environment, and observable objects highlight a declarative Handler-based approach extended for different communication patterns. The `endpointName` and `operation` modifiers provide knowledge source-based information that is shared across Interface Exporters and used to refine different aspects of the gRPC and GraphQL web API specification documents. Metadata and option annotations at different levels of the web service, Handlers, and parameters enable extension mechanisms. Several Interface Exporters use these mechanisms to refine web API-specific information. Apodini features, such as the relationship information, enable advanced Interface Exporter features, including HATEOAS information for RESTful web APIs.

All these aspects demonstrate that the core Apodini DSL and Interface Exporter mechanisms empower web service interface- and web API type-independent web service development, positively answering Knowledge Question 2. The single-case mechanism experiments achieve the validity of the artifacts designed to address Technical Research Goal 1 and Design Problem 1.

Chapter 9

Web Service Instantiations

Five single-case mechanism experiments were performed that encompass web services built using the Apodini DSL and the Interface Exporters validated in Chapter 8 to demonstrate the extensibility and applicability of the Apodini ecosystem. The extensibility is demonstrated by showing different application domains and extension points, enabling web service developers to reuse software components in the Apodini ecosystem. The evolution and deployment of these web services is supported using Apodini Migrator and Deployer artifacts to validate their evolution-related capabilities. Knowledge Questions 4 and 6 aim to investigate how web service API evolution (Definition 7, page 7) and web service deployment evolution (Definition 8, page 8) related challenges are addressed using the Apodini ecosystem artifacts.

> **Knowledge Question 4:**
> How do web API type-independent migration guides translate to client-side web API-specific migration mechanisms?

> **Knowledge Question 6:**
> How do Deployment Providers enable deployment evolution in different deployment environments?

Figure 9.1 demonstrates the focus of the single-case mechanism experiments on the Apodini subsystems. While all subsections use the shared Swift-based Apodini DSL, different subsections put an emphasis and focus on specific subsystems of the ecosystem. The validation also demonstrates the applicability to different application domains by demonstrating the usage of Apodini in a wide variety of projects.

Section 9.1 demonstrates a basketball player health monitoring system. The section demonstrates the extensibility of the Swift-based Apodini DSL when incorporating databases, mobile push notifications, and serving web pages using Apodini.

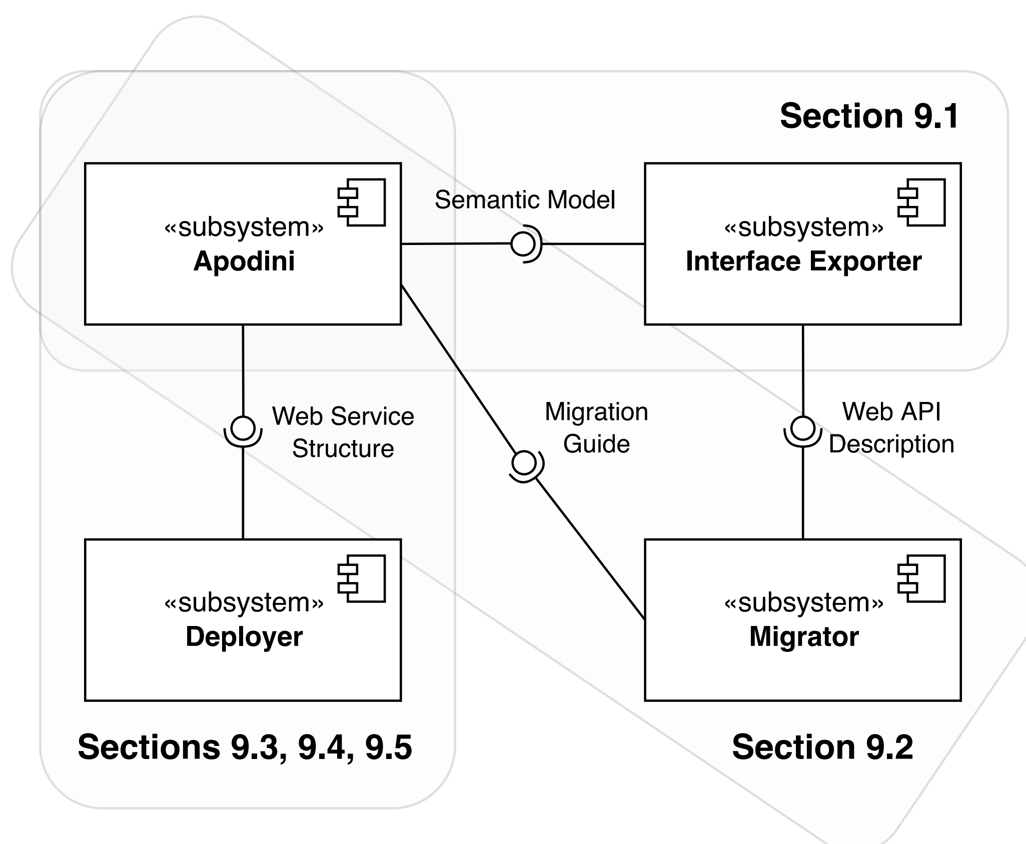


Figure 9.1: The validation sections are mapped to the Apodini subsystems shown in Figure 6.1 (page 93). All sections use the shared Apodini DSL functionality and demonstrate the usage of the subsystems in different application domains. (UML Component Diagram)

Section 9.2 details a greenfield event management platform, presenting a large proportion of possible API changes supported by Apodini Migrator. The single-case mechanism experiment demonstrates API changes to the project and how Apodini Migrator enables client stability while allowing web API evolvability. The section highlights the automatic capabilities of the Apodini ecosystem while also demonstrating weaknesses and manual intervention mechanisms.

The following three sections demonstrate several functionalities of the Apodini Deployer subsystem and showcase how the different extension mechanisms can be applied to various deployment structures and deployment environments. Section 9.3 demonstrates an expense and income tracking application that can be decomposed into subprocesses and to a FaaS-based hosting environment using different Apodini Deployment Providers. Sections 9.4 and 9.5 both demonstrate the usage of a constraint-based WoT-based deployment in two IoT-based projects. Section 9.4 describes a smart city project using heterogeneous IoT devices. Section 9.4 describes a heterogeneous WoT water quality measurement system.

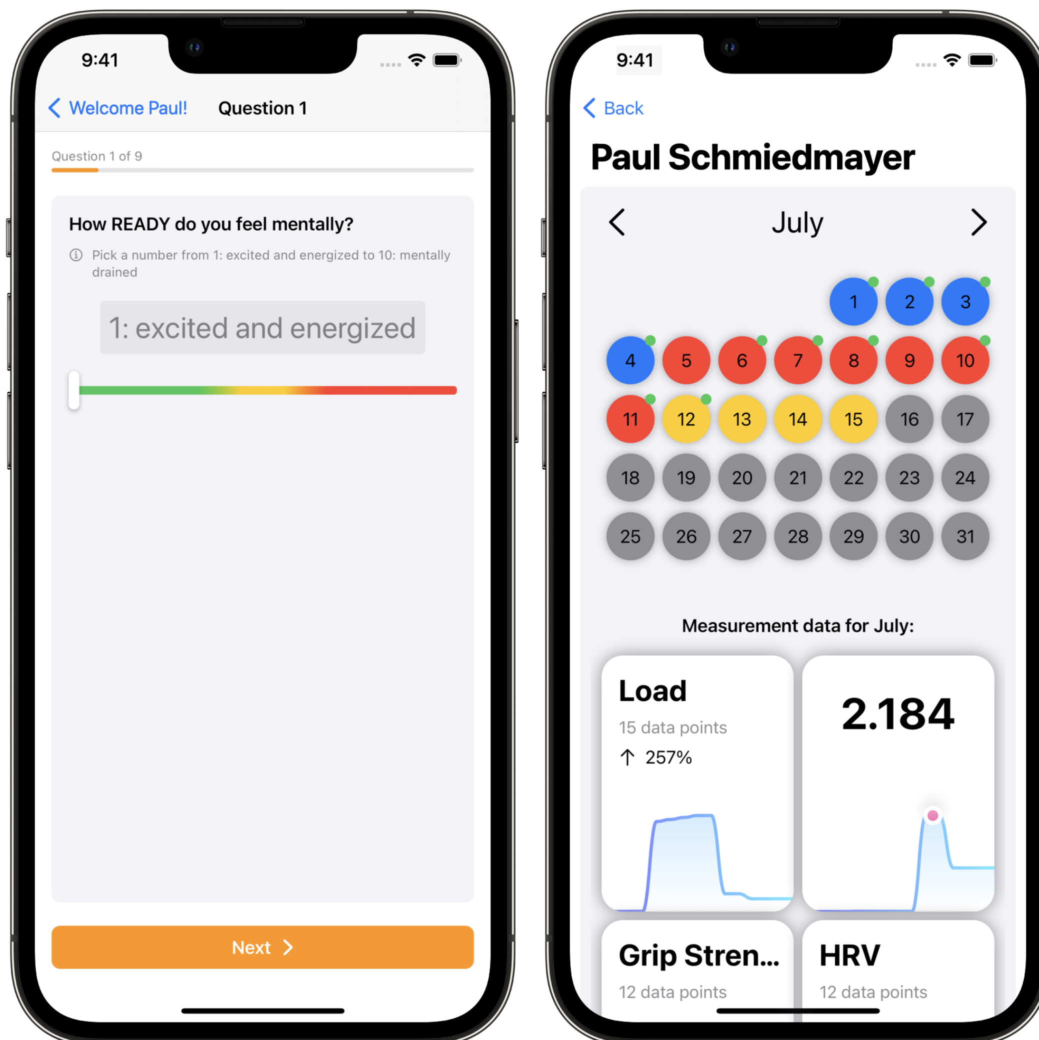
9.1 Basketball Player Health Monitoring System

The basketball player health monitoring system demonstrates a web service and mobile application developed in the Swift programming language. The project provides a prototype for a health monitoring and injury prevention platform for a German first-league basketball club. The mobile application named TrainLens allows basketball players to voluntarily conduct a self-evaluation questionnaire and record several exercises in the morning to assess their daily fitness level and mental well-being. The collected data also includes heart rate measurements conducted using devices provided by the sports club. The application guides players through an onboarding process and reminds them to record their state every morning. The application aggregates the information and sends it to Apodini-based web services. The web service aggregates the data and analyzes it using proprietary algorithms developed based on sports and health science research, including heart rate information [143, 217, 218]. The application presents coaches and the club management with insights and predictions about the players' performance, past fitness, and well-being level. Figure 9.2 details two screens of the TrainLens application showing the data entry and analyzed data inspection functionality.

The usage of the system for the description of this single-case mechanism experiment focuses on the functionality of the web service relevant for the dissertation and does not incorporate health data or an analysis of the developed algorithms. The system demonstrates the web service development capabilities of the Apodini ecosystem, including the usage of extensions to enable additional functionality in the Apodini DSL context. These extension points include sending push notifications to client devices, communicating with a database, and providing an HTML template-based web page. The web page allows administrators to create user accounts, manage teams, and set parameters for the algorithms performed on the collected data.

The basketball player health monitoring system is one of twelve mobile application-related projects conducted during the iPraktikum 2021. The iPraktikum is a project-based capstone course teaching students software engineering using projects involving a wide variety of project partners and project topics [64]. The project course consists of several student teams, distributed at the beginning of the semester and working on challenges proposed by the project partners [87]. Students learn about Swift-based mobile application and web service development as part of a two-week project course at the beginning of the semester [251].

During the iPraktikum in the summer semester of 2021, more than 70 students learned how to develop web services using the Swift-based Apodini DSL. Six out of twelve teams in the iPraktikum preceded to develop a web service using the Swift-based Apodini DSL for their project. Five out of six Apodini iPraktikum



(a) TrainLens Questionnaire (b) Player Overview

Figure 9.2: Two screens of the TrainLens iOS Application communicating with the Swift-based Apodini web service. The first screen displays the first question of the questionnaire that players can answer every morning. The second screen displays the user interface of the data entry overview of a single player that is retrieved from the web service. The color of the days in the month overview indicates the assessment of the algorithm based on the player data.

projects used the ApodiniDatabase[41] extension target providing the FluentKit[42] object–relational mapping framework to Apodini Developers. The TrainLens web service also uses the ApodiniNotifications[43] to send push notifications to the TrainLens

[41]The ApodiniDatabase provides database-related functionality: https://github.com/Apodini/Apodini/tree/develop/Sources/ApodiniDatabase.

[42]FluentKit is part of the Vapor project: https://github.com/vapor/fluent-kit.

[43]The ApodiniNotifications enables developers to send Apple Push Notification service (APNs) push notifications to mobile devices: https://github.com/Apodini/Apodini/tree/develop/Sources/ApodiniNotifications.

application. The ApodiniLeaf[44] dependency in the TrainLens web service enables using the LeafKit[45] enables serving HTML pages using templating mechanisms. The students were supported by experienced students and continuously provided feedback to improve Apodini and fix bugs in different Interface Exporters and the main Apodini DSL. The ApodiniLeaf project was developed based on the need in the TrainLens project to serve static web pages, and the Apodini Interface Exporter system was extended to return binary large objects (`Blob`-type) as Handler responses. Additional connection context such as the `Information` functionality in the Apodini DSL was developed to retrieve further context about the requests and store context in the web service responses.

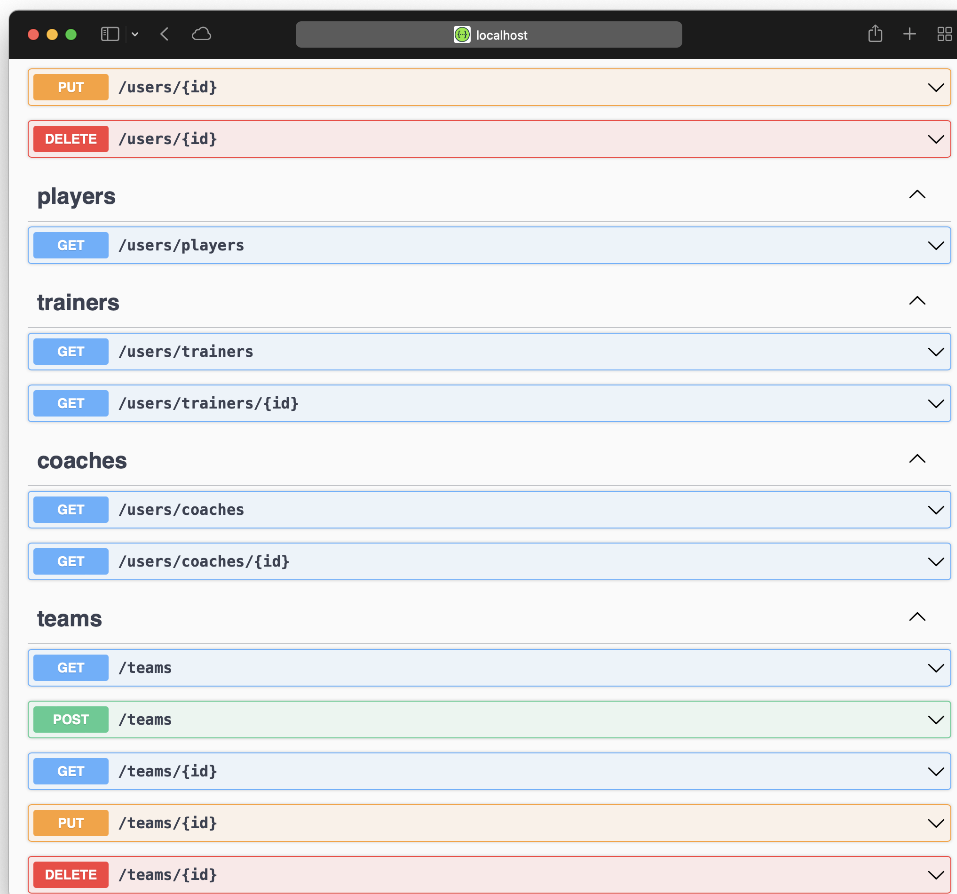


Figure 9.3: The Swagger UI of the TrainLens web service exported by the OpenAPI Interface Exporter after updating the web service to latest Apodini version and addressing implementation inconsistencies. The screenshot details the routes to manage users and retrieve users by their role including players, trainers, and coaches. The `/teams` endpoints are used to manage the teams of the basketball club.

[44] The ApodiniLeaf Swift Package can be found at https://github.com/Apodini/ApodiniLeaf.
[45] LeafKit is part of the Vapor project: https://github.com/vapor/leaf-kit.

The Apodini RESTful Interface Exporter described in Section 8.5 maps HTTP headers to `Information` in request and serializes `Information` returned by the web service into HTTP headers in the HTTP response. The web service includes 43 HTTP-based endpoints documented in an OpenAPI-based documentation and presented in a SwaggerUI web page offered by the OpenAPI Interface Exporter described in Section 8.5.1. Figure 9.3 displays the SwaggerUI interface, documenting the endpoints of the TrainLens project offered by the RESTful Interface Exporter combined with the OpenAPI Interface Exporter after updating the web service to the latest Apodini version and addressing implementation inconsistencies.

9.2 Event Management Plattform

The Apodini web service offering an event management platform demonstrates the web service API evolution-related capabilities of the Apodini ecosystem. The web service describes the web service interface design capabilities of the Swift-based Apodini DSL and how Apodini Migrator can create client libraries to communicate with the web service. The web service is subsequently improved by adding, removing, and updating functionality affecting the RESTful web API. The RESTful Apodini Migrator automatically migrates the client library to a new version, enabling client stability when communicating with an evolving web service.

The project was developed as part of the master's thesis titled *Automated Generation of Machine-Readable Migration Guides for Web Services* by Eldi Cano [69]. Cano extended the existing basketball player health monitoring system described in Section 9.1, the event management platform described in this section, and an existing building savings contract information service initially developed in JavaScript and ported to the Swift-based Apodini DSL to validate the Apodini Migrator subsystem [69]. The event management platform web service[46] offers a RESTful API using the RESTful Interface Exporter and the OpenAPI Interface Exporter. Listing 9.1 displays the web service of version one of the web service.

The API contains endpoints to perform CRUD operations on events, read and create categories, and retrieve category groups. Users can register, log in, retrieve user information, and delete an account. The user's home feed can display information about events and other user-related information, while the experiences endpoint provides detailed information about events. As detailed in Section 7.3, the `Migrator` configuration enables the configuration of the API document export and the export options for a migration guide. The configurations of the Apodini Migrator subsystems can be overwritten by command-line arguments passed to the automatically added `migrator` subcommand.

[46]The implementation of the event management platform is available at https://github.com/Apodini/ApodiniMigratorExample. The instantiation returns mock data for all endpoints.

```swift
@main
struct EventManagementPlatform: WebService {
    var metadata: Metadata {
        Version(major: 1)
    }

    var content: some Component {
        EventsComponent()
        CategoriesComponent()
        HomeFeedComponent()
        ReviewsComponent()
        UserComponent()
    }

    var configuration: Configuration {
        // gRPC Configuration ...
        REST {
            OpenAPI()
        }
        Migrator(documentConfig: .export(.endpoint("api-document")))
    }
}
```

Listing 9.1: Swift-based Apodini DSL Web service version one of the event management platform. The web service explicitly specifies the version of the web service using the metadata property of the web service. The components listed in the `content` property each describe a part of the web service. The `configuration` details the REST and OpenAPI exporter as well as the `Migrator` configuration. The gRPC and TLS configurations are omitted in the listing. The `Migrator` configuration in the web service defines that the API document used by the REST Migrator is exported at the `/api-document` endpoint.

Listing 9.2 displays the command needed to export the API Document in its JSON-based representation. The exported API document builds the basis for the migration guide and client-based API migration demonstrated in this section. The API document details the current API structure of the web service in an Interface Exporte-independent structure representing elements of the semantic model described in Section 7.2.

```
$ swift run QONECTIQV1 migrator document \
    --doc-directory=./Documents --doc-format=json
[...]
API Document exported at [...]/Documents/api_v1.0.0.json
```

Listing 9.2: The `migrator document` command is used to generate a Migrator API document from version 1 of the web service. The command starts parsing the Apodini DSL and generates a JSON-based API document in the specified Documents directory. The listing omits log messages and absolute file paths.

```
$ swift run QONECTIQV2 migrator compare \
    --old-document-path ./Documents/api_v1.0.0.json \
    --guide-directory ./Documents
[...]
Migration Guide exported at [...]/Documents/migration_guide.json
```

Listing 9.3: Generation of the migration guide based on the API document exported from version one of the web service. The `--old-document-path` argument details the path to the API document, and the `--guide-directory` passes the directory where the migration guide should be saved to the Migrator subsystem. The final log message describes the successful export of the migration guide. The listing omits log messages and absolute file paths.

```
{
    "type": "update",
    "id": "User",
    "updated": {
        "type": "property",
        "property": {
            "type": "idChange",
            "from": "ownEvents",
            "to": "myEvents",
            "similarity": 0.750000,
            "breaking": true,
            "solvable": true
        }
    }
}
```

Listing 9.4: Model-related API changes between version one and two of the event management web service documented in the JSON-based migration guide. The property `ownEvents` was renamed to `myEvents` which was detected with a similarity score of 75%. The change is classified as a braking change with an automatically solvable solution contained in the migration guide.

Version two of the web service introduces several breaking and non-breaking changes as classified in Section 4.2.1. The API Document of the first web service version is subsequently used to generate a migration guide using version two of the event management web service. Listing 9.3 displays the command and partial log output of the generation of the migration guide.

The migration guide documents all changes of the web API between two versions of an API document in an API type-independent description based on the concepts found in the API document structure. As detailed in Section 7.3, the migration guide is divided into model and endpoint-related changes that are supported by changes on a service level, scripts, values, and representations. The resulting migration guide is more than 1500 lines long, correctly detailing all changes between the two versions, and can be found in the open-source repository of the validation. It includes three service level changes documenting the new version num-

```
{
    "type": "update",
    "id": "getHomeFeedForUserWithID",
    "updated": {
        "type": "identifier",
        "identifier": {
            "type": "update",
            "id": "EndpointPath",
            "updated": {
                "from": {
                    "id": "EndpointPath",
                    "value": "/home-feed/{userID}"
                },
                "to": {
                    "id": "EndpointPath",
                    "value": "/home/{userID}"
                }
            },
            "breaking": true,
            "solvable": true
        }
    },
    "breaking": true,
    "solvable": true
}
```

Listing 9.5: Endpoint-related change documented in same the migration guide as detailed in Listing 9.4. The change documents a change in the endpoint path of the Handler with the identifier `getHomeFeedForUserWithID`. The change was automatically detected based on the identifier. The old and the new value are documented in the migration guide (line 12 & 16), resulting in a breaking but solvable API change.

ber and configurations injected by the RESTful and gRPC-based Apodini Migrator. There are 56 total model changes, four additions, one removal, and 51 update changes. Forty-nine of the model changes are classified as breaking, and 55 are classified as solvable, while one change is classified as non-solvable: The removal of the `CategoryStatus` type. There are 22 total endpoint changes, three additions, one removal, and 18 update changes. Seventeen of the model changes are classified as breaking, and 21 are classified as solvable, while one change is classified as non-solvable: The removal of the `usersOfExperience` endpoint. The migration guide also contains 30 change scripts, 8 JSON values, and 11 updated JSON representations as described in Section 7.3. Listing 9.4 and Listing 9.5 both display segments of the migration guide documenting one model and one endpoint related change documented in the migration guide.

The migration guide is the primary input for the different Apodini Migrator implementations. The Apodini Migrator currently supports a RESTful API Migrator and a gRPC-based Migrator. The gRPC-based Migrator uses the Protocol

Buffer specifications provided by the gRPC Interface Exporter as an input in addition to the migration guide. The REST Apodini Migrator uses the web service API type-independent API document and the migration guide as input when generating client libraries. The REST Apodini Migrator used in this section can derive the RESTful API of an Apodini web service based on the API document exported using the Apodini integration.

> **Knowledge Question 4:**
> How do web API type-independent migration guides translate to client-side web API-specific migration mechanisms?

The instantiations address Knowledge Question 4 by demonstrating the extensibility of the Apodini Migrator mechanism for two web API types. The Apodini Migrator instances rely on a shared foundation provided by the Apodini Migrator subsystem. This subsystem enables the generation and parsing of migration guides as well as a code generator component described in Figure 6.7 (106). The Apodini Migrator instantiation features a code generation DSL that provides reusable components to the Apodini Migrators. This mechanism enables Migrators to focus on the web API type-specific migration pattern instantiations for different web API types.

We demonstrate this functionality using the REST Apodini Migrator. The client library migrations presented in Listing 9.7 and Listing 9.8 migrations are both generated as part of the REST Apodini Migrator's client library generation shown in Listing 9.6. The Swift-based Apodini REST Migrator creates a Swift Package that contains a shared networking layer and a stable client facade offering a Swift interface for the client to use.

```
1 $ swift run migrator migrate rest \
2     --package-name ApodiniMigratorExampleClient \
3     --target-directory ../ \
4     --document-path ../Documents/api_v1.0.0.json \
5     --migration-guide-path ../Documents/migration_guide.json
6 [...]
7 Starting migration of package ApodiniMigratorExampleClient
8 Package ApodiniMigratorExampleClient was migrated successfully. You can
   ↪  open the package via ApodiniMigratorExampleClient/Package.swift
```

Listing 9.6: Usage of the RESTful Migrator in the Apodini Migrator project to create a stable client library with a version one interface communicating with version two of the web service. The listing omits some log messages until the tool successfully creates the client library as a Swift Package. The command requires a name (line 2) and target directory (line 3) for the Swift Package and a path to the API document (line 4) for the old version of the web service. The migration guide passed in as the last argument (line 5) documents the changes between the old version and the version the client library should be migrated to.

```swift
public struct User: Codable {
    private enum CodingKeys: String, CodingKey {
        // ...
        case eventsOfInterest = "interestedIn"
        // ...
        case ownEvents = "myEvents"
        // ...
    }
    // ...
}
```

Listing 9.7: Migration of a property rename documented in Listing 9.4 by the RESTful Apodini Migrator for a Swift-based client library. The REST Apodini Migrator uses the Swift `Codable` coding keys feature to rename the serialization key while guaranteeing source stability for the Swift-based client interface.

```swift
public extension HomeFeed {
    static func getHomeFeedForUserWithID(
        showPreviousEvents: Bool = try! Bool.instance(from: 7),
        userID: UUID,
        authorization: String? = nil,
        httpHeaders: HTTPHeaders = [:]
    ) -> ApodiniPublisher<HomeFeed> {
        // ...
        let handler = Handler<HomeFeed>(
            path: "home/\(userID)",
            // ...
        )
        // ...
    }
}
```

Listing 9.8: Migration of an endpoint-related rename of the URI path for the HTTP-based request to the web service documented in Listing 9.5. The REST Apodini Migrator offers a stable Swift interface using the `getHomeFeedForUserWithID` function by only altering the function body with the new path parsed from the change description in the migration guide.

Listings 9.7 and 9.8 demonstrate the stable client facade with a model type in Listing 9.7 and a function sending a network request to an endpoint in Listing 9.8. The REST Migrator uses Swift programming language features to provide a stable interface to the client. The `rest` migrate subcommand provides access to the RESTful Apodini Migrator, the `grpc` subcommand would provide access to the gRPC Apodini Migrator. The Migrator changes the serialization and deserialization functions for model types to communicate with a newer web service version. The functions making networking calls to the web service also provide stable Swift signatures. At the same time, the Apodini REST Migrator alters the function bodies hidden behind the signature to create requests and parse responses of the new web service version. This web API evolution supporting mechanism enables client stability while allowing web service API evolvability for web service developers.

The Listings in this section and all further examples detailed in the open-source version of the event management platform demonstrate the capabilities and extensibility of the Apodini DSL and Apodini Migrator subsystem. The Apodini Migrator components take advantage of the DSL-based web service description to parse a detailed representation of all aspects of the web service interface translated into the web API documents. A tree-based comparison algorithm provided by the Apodini Migrator subsystem enables different Migrators to create a client library and provide client-stable migrations empowering web API evolution for web service developers. While the Migrator subsystem cannot automatically provide migrations for all API changes, web service developers can use the JavaScript-based migration functions documented in the migration guide to refine further migration steps. These functions are incorporated in the generation of interface-stable client libraries. The Migrator instantiations demonstrate the client library generation and migration techniques for various model and endpoint-related change types. The mappings demonstrated show the mapping of an abstract migration description to a concrete Swift-based instantiation for RESTful APIs, providing empirical evidence answering Knowledge Question 4. The techniques demonstrate the web API evolution-related capabilities of the Apodini Migrator system enabled by the unique DSL-based approach of developing evolvable web services.

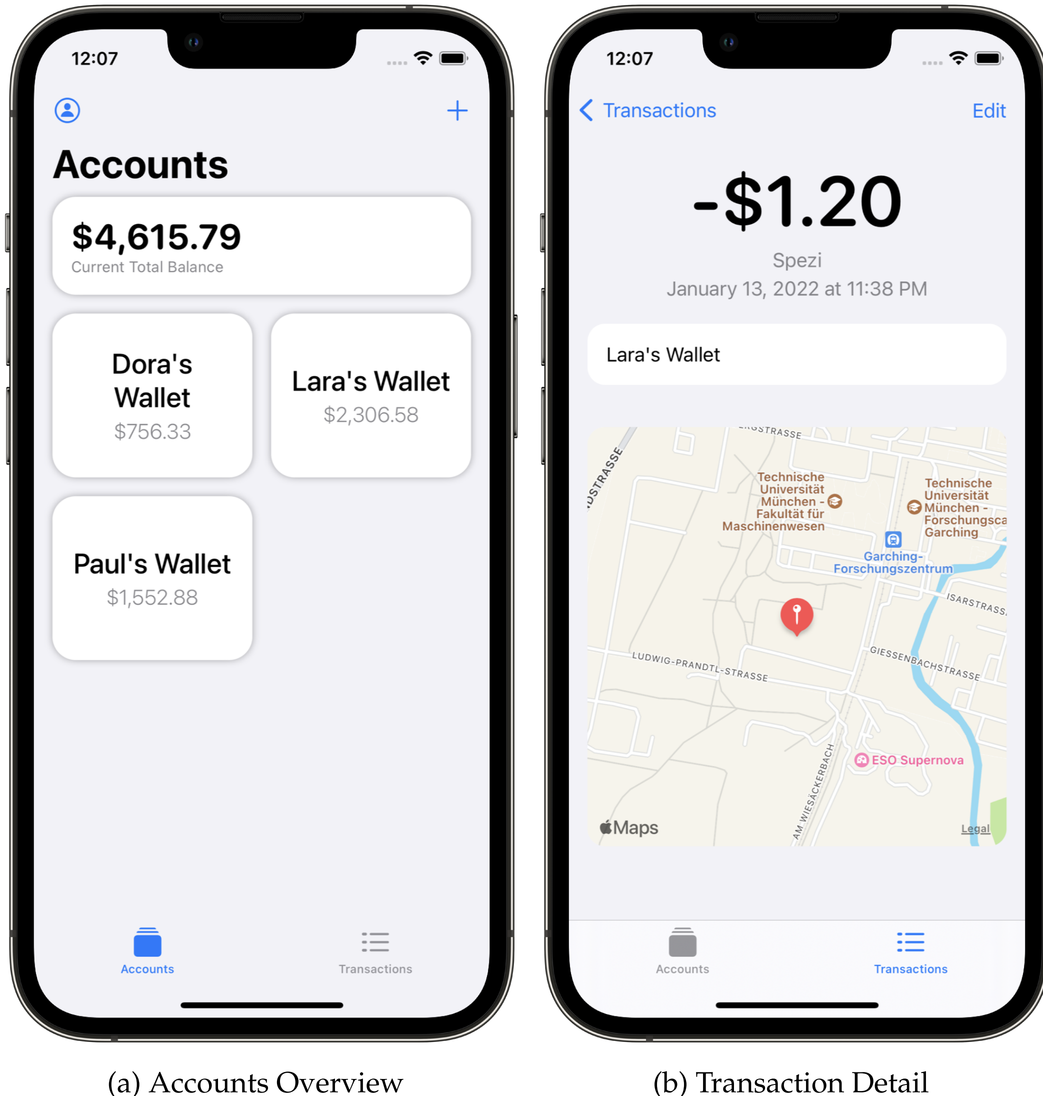
9.3 Expense and Income Tracking Application

The Expense and Income Tracking Application (Xpense) is a Swift-based software system used to highlight recent DSL-based software development techniques in the client and web service subsystems. The system is used to teach Swift and iOS development as part of the Swift Bootcamp preceding the iPraktikum capstone course at the Technical University of Munich [251]. The system demonstrates the usage of the Apodini DSL for web service development and SwiftUI for client-side development. Students learn the Swift programming language by following alternating Swift and app development sessions that demonstrate building the Xpense application throughout eight lectures [251]. The application is documented and available as an open-source project[47].

The application allows users to keep track of their income across different accounts that can be created, modified, and deleted in the iOS application. Transactions within these accounts can be associated with a description, date, and location. A user is identified by an account using a user-password combination to authenticate users in the client application. Figure 9.4 displays the user interface of the client application detailing the accounts overview as well as the detail screen of a transaction. The client application and the web service demonstrate the usage of the Swift

[47]The Xpense application can be found at https://github.com/Apodini/ApodiniXpenseExample.

structured concurrency features introduced in Swift 5.5. The client and web service share the same Swift-based model types and functionality extensions, demonstrating the advantage of developing the client application and web service in the same programming language enabled by the Swift-based Apodini DSL.



(a) Accounts Overview

(b) Transaction Detail

Figure 9.4: User interface of the Xpense iOS application. Figure 9.4a displays the accounts overview showing three accounts as well as the total balance across all accounts. Figure 9.4b shows the transaction view for a single transaction that is associated with a balance, description, data, the related account, and a location.

The web service uses the REST and OpenAPI exporters to provide a web API to the client application. Listing 9.9 provides an overview of the main web service type described in the Apodini DSL. The web service consists of 13 Handlers annotated with modifiers and metadata information. The web service demonstrates the usage of the Apodini Authorization extensions, allowing developers to annotate elements of the DSL components with security-related metadata. In the Xpense web

```swift
@main
struct XpenseWebService: WebService {
    @Option
    var port: Int = 80

    var configuration: Configuration {
        HTTPConfiguration(bindAddress: .interface(port: port))
        REST {
            OpenAPI()
        }
        ApodiniDeployer(runtimes: [
            Localhost.self,
            AWSLambda.self
        ])
    }

    var content: some Component {
        Text("Welcome to the Xpense Web Service!")
        Group {
            AccountComponent()
            TransactionComponent()
        }.metadata {
            Authorize(
                User.self,
                using: BearerAuthenticationScheme(),
                verifiedBy: UserTokenVerifier()
            )
        }
        UserComponent()
    }
}
```

Listing 9.9: The Xpense web service shows the authorization metadata annotations for the `AccountComponent` and `TransactionComponent` subsystems. The web service uses the `REST` & `OpenAPI` Interface Exporters and the `HTTPConfiguration` to alter the binding port based on a launch option. The `ApodiniDeployer` configuration is defined to use two runtimes: The `Localhost` (Section 9.3.1) and `AWSLambda` (Section 9.3.2) Deployment Providers.

service, the client application uses the Basic HTTP authentication scheme [237] to allow a user to log in using a username-password combination. The user retrieves a token that can be used to further authenticate requests to the accounts and transactions Handlers using the Bearer token HTTP authentication scheme [146]. The web service does not include a build-in TLS configuration and needs to be deployed in conjunction with a reverse proxy or an API gateway to ensure a secure connection to the client, adhering to security best practices for basic and bearer authentication [237, 146]. The web service also incorporates configurations and subcommands to automatically generate deployment structures for the process-based Localhost Deployment Provider described in Section 9.3.1 and the FaaS-based AWS Lambda Deployment Provider demonstrated in Section 9.3.2.

9.3.1 Localhost Deployment Provider

The Localhost Deployment Provider demonstrates the automatic deployment structure creation enabled by the Apodini Deployer subsystem. The Deployment Provider is build based on the deployment structure defined in Figure 7.4 (page 122). The Deployment Provider creates a subprocess for each deployment node of the deployed system. The default configuration is to create a deployment node for each exported endpoint, resulting in 13 different subprocesses, one for each Handler in the web service. A proxy server forwards the requests to the deployment nodes using the mapping defined in the deployed system based on the deployment structure.

```
$ swift run DeploymentTargetLocalhost ../WebService \
    --product-name WebService
[...]
Compiling target 'WebService'
[...]
Server starting on 0.0.0.0:80
[...]
Server starting on 0.0.0.0:52011
Server starting on 0.0.0.0:52003
Server starting on 0.0.0.0:52007
Server starting on 0.0.0.0:52005
Server starting on 0.0.0.0:52009
Server starting on 0.0.0.0:52001
Server starting on 0.0.0.0:52002
Server starting on 0.0.0.0:52012
Server starting on 0.0.0.0:52006
Server starting on 0.0.0.0:52000
Server starting on 0.0.0.0:52010
Server starting on 0.0.0.0:52004
Server starting on 0.0.0.0:52008
```

Listing 9.10: The `DeploymentTargetLocalhost` in the Apodini Deployer subsystem requires a path to the Swift Package directory (line 1) and a name of the Swift package product (line 2) to create a localhost subprocess deployment. The Deployment Provider subsequently compiles the product, extracts the deployment structure, and starts the proxy service and a subprocess running a web service for each Handler. The listing omits irrelevant log messages.

The `LocalhostRuntime` Apodini Deployer integration shown in Listing 9.9 provides subcommands to interact with the web service at deployment time. The `deploy export-ws-structure` subcommand and further nested Deployment Provider-specific subcommands enable the extraction of the deployment structure. The `deploy startup` subcommand and further nested Deployment Provider-specific subcommands enable Deployment Providers to modify the startup behavior of the web services in the execution environment. The Localhost Deployment Provider uses the subcommands to export the deployment structure, including relevant metadata that is then subsequently used to startup multiple subprocesses using the Local-

host-specific `deploy startup` subcommand. Listing 9.10 demonstrates the usage of the Localhost Deployment Provider for the Xpense web service. The Deployment Provider demonstrates the usage of the deployed system to easily prototype the decomposition of an Apodini web service on a single machine.

9.3.2 AWS Lambda Deployment Provider

The AWS Lambda Deployment Provider demonstrates the generation of a deployment structure and decomposition of FaaS-based deployments as discussed in Section 5.1.1. The AWS Deployment Provider incorporates deployment-specific metadata and groupings into creating the final deployed system. It uses Docker to create suitable binaries for deployment to a FaaS environment. The AWS Lambda Deployment Provider integrates additional functionality in the Apodini Networking layer to translate AWS Lambda executions to requests processed by Apodini and forwarded to Interface Exporters. Listing 9.11 demonstrates the AWS Lambda Deployment Provider CLI and log output.

```
1  $ swift run DeploymentTargetAWSLambda deploy ../WebService \
2      --product-name WebService \
3      --s3-bucket-name apodinixpenseexample
4  [...]
5  Preparing docker image
6  [...]
7  Successfully built docker image. image name: apodini-lambda-builder
8  Generating web service structure
9  [...]
10 Successfully generated web service structure
11 [...]
12 Creating lambda package
13 Zipping lambda package
14 Uploading lambda package to
    ↪  s3://apodinixpenseexample/apodini-lambda/lambda.out.zip
15 S3 upload done.
16 Creating lambda functions for nodes in the web service deployment
    ↪  structure (#nodes: 13)
17 [...]
18 Importing API definition into the API Gateway
19 Updating API Gateway name
20 Deployed 13 lambdas to api gateway w/ id '2rf6y3467d'
21 Invoke URL: https://2rf6y3467d.execute-api.eu-central-1.amazonaws.com/
22 Done! Successfully applied the deployment.
```

Listing 9.11: In addition to the same input as the Localhost Deployment Provider, the CLI of the AWS Lambda Deployment Provider requires an AWS S3 storage bucket name and an identifier for an AWS API Gateway. The AWS Lambda Deployment Provider creates a new API Gateway as no API Gateway identifier was passed to the command. The log messages are explained in the description deployment process in this section. The listing omits non-essential log messages.

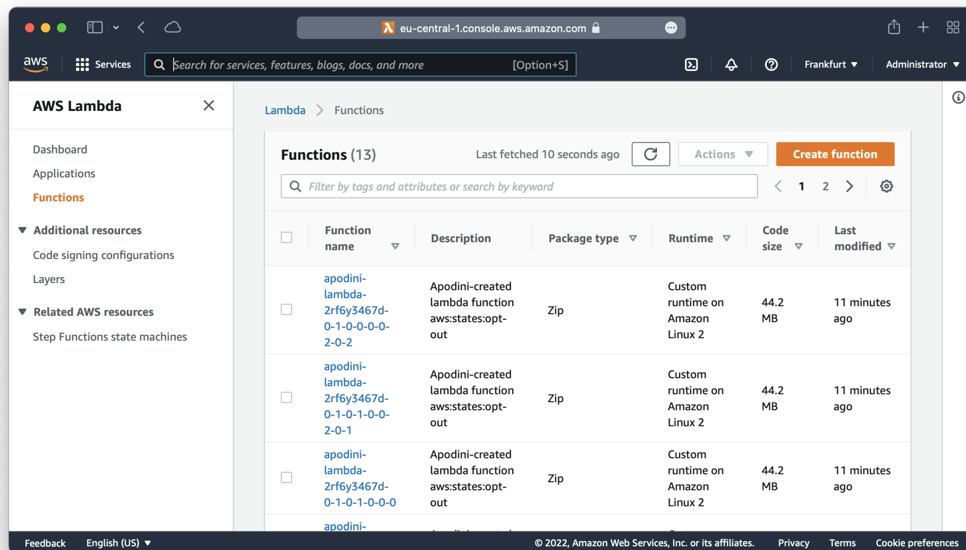


Figure 9.5: Screenshot of the AWS Lambda Dashboard showing part of the list of the 13 deployed AWS Lambda functions. The Lambda functions are reachable using the API gateway shown in Figure 9.6. Each Lambda function was created using the zip file uploaded to the S3 storage bucket defined in Listing 9.11. The AWS Account-ID has been redacted.

The AWS Lambda Deployment Provider first prepares an Amazon Linux 2-based Docker image used to compile the web service binary executable in the AWS Lambda environment. The Deployment Provider prepares the Docker image and compiles the web service in the Docker image. The compiled web service is subsequently used to extract the web service structure based on the Apodini DSL using the Apodini Deployer subcommands. The deployment structure and its metadata-based constraints are parsed by the AWS Lambda Deployment Provider and are used to define the deployed system structure.

After defining the deployed system structure, the web service binary is packaged in a zip file to be uploaded to the user-defined S3 storage bucket. The storage bucket is used to generate the AWS Lambda functions following the deployed system structure. Like the Localhost Deployment Provider, the AWS Lambda Deployment Provider creates one deployment node corresponding to one AWS Lambda function for each exported endpoint. Figure 9.5 provides a screenshot of the AWS Lambda dashboard after the creation of the Lambda functions is complete.

After creating the FaaS functions, the AWS Lambda Deployment Provider configures the AWS API Gateway to forward HTTP requests to the appropriate Lambda functions. The API Gateway serves as a load balancer and automatically translates RESTful HTTP requests to Lambda execution events. The API Gateway is configured using an OpenAPI document extracted from the OpenAPI exporter that is de-

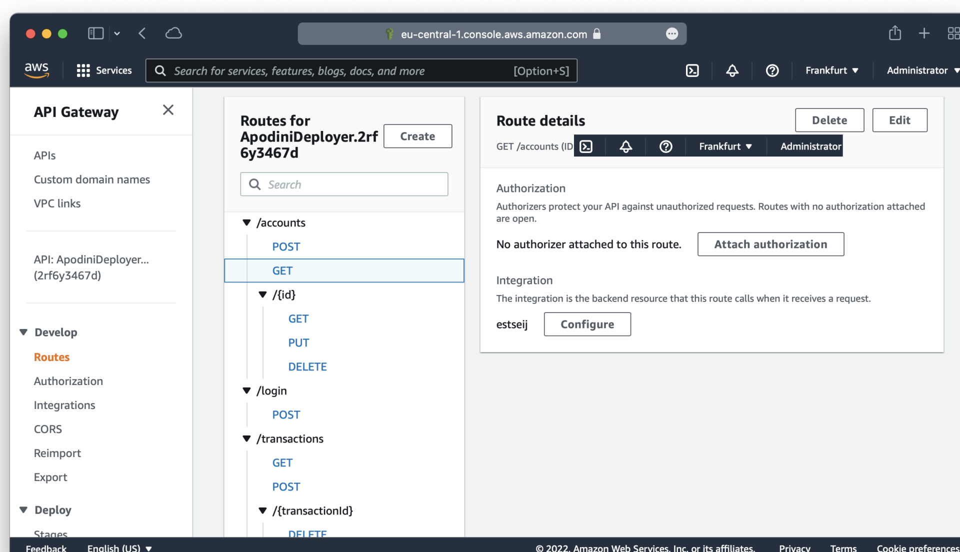


Figure 9.6: Screenshot of the AWS API Gateway Dashboard showing the API Gateway configured by the AWS Lambda Deployment Provider execution shown in Listing 9.11. The API Gateway is configured to forward HTTP requests defined by the OpenAPI specification to the deployed AWS Lambda functions shown in Figure 9.5. The AWS Account-ID has been redacted.

fined as part of the Xpense web service. Figure 9.6 shows a screenshot of the configured API Gateway after the execution of the AWS Lambda Deployment Provider shown in Listing 9.11. After configuring the API Gateway and the AWS Lambda functions, the AWS Lambda Deployment Provider exits with information about the deployed system and how the API Gateway can be reached.

> **Knowledge Question 6:**
> How do Deployment Providers enable deployment evolution in different deployment environments?

The Apodini DSL-based approach enables a complete overview of a web service's structure and deployment constraints, enabling the partitioning and deployment functionality of the Apodini Deployer subsystem and concrete Deployment Providers demonstrated in this section. The Xpense web service single-case mechanism experiment demonstrates how the Apodini DSL's extensibility enables security-related metadata annotations translated into authentication requirements. Adding a subprocess-based or FaaS-based deployment only requires adding an entry in the web service configuration, automatically configuring additional subcommands and runtime networking behavior without the need to adapt the web service components. The AWS Lambda instantiation demonstrates the usage of a FaaS-based Deployment Provider automatically partitioning the web service.

9.4 Smart City IoT System

External domain-specific language-based deployment specifications are often used to persist and automate the deployment of distributed systems as discussed in Chapter 5. External DSLs, such as the TOSCA specification [180], allow developers to define the structure of the distributed components and the process of deploying a distributed system. The TOSCA specification supports two approaches to describe a distributed deployment: An imperative describing build plans and a declarative approach defining the topology model of the distributed system. [108, 180]. Research by Franco da Silva et al. and Li et al. demonstrates the applicability of the declarative approach of the TOSCA standard for IoT-based deployments by specifying artifacts that are described in the XML-based TOSCA specifications [108, 176].

Apodini embraces the declarative definition of the deployment structure by embedding it in the web service definition. This section focuses on using containerization techniques and the metadata annotation-based insights provided by the Apodini DSL to improve WoT based deployments. Instead of relying on different deployment artifacts, we use the Everything in Code approach described in Section 7.1. Combining a model and code-first approach provides a comprehensive description of the deployable web service that the Apodini Deployer subsystem utilizes.

The smart city IoT system demonstrates the applicability of the Apodini ecosystem to the IoT and WoT deployment domain as described in Section 5.1.3. The project showcases the constraint and metadata-based deployment mechanisms described in Chapter 5. The web service developer annotates the WoT web service elements with device-specific and deployment-specific constraints. The Apodini IoT Deployment Provider uses these constraints and annotations to create a deployment structure that is transformed into a deployed system.

The single-case mechanism experiment was developed during and based on the smart city IoT system developed during the Joined Advanced Summer School (JASS) 2021. JASS is a multi-university project course involving the St. Petersburg State Electrotechnical University (LETI) and the Technical University of Munich (TUM) [164]. The 2021 edition of the two-week course was conducted as an online distributed global software engineering course involving the Imperial College London (Imperial), St. Petersburg State Electrotechnical University, and the Technical University of Munich [250]. The focus on the 2021 edition of the course was continuous software engineering in IoT systems conducted in a global classroom setting [250]. The miniaturized smart city setup of the JASS 2021 project consistes of education-focused Tello drones, Raspberry Pi-based model cars named DuckieBots, and UDP/IP enabled smart light bulbs representing traffic lights [250]. The students were tasked to develop a continuous software engineering-based system, incorporating all device types to simulate a smart city infrastructure setup [250].

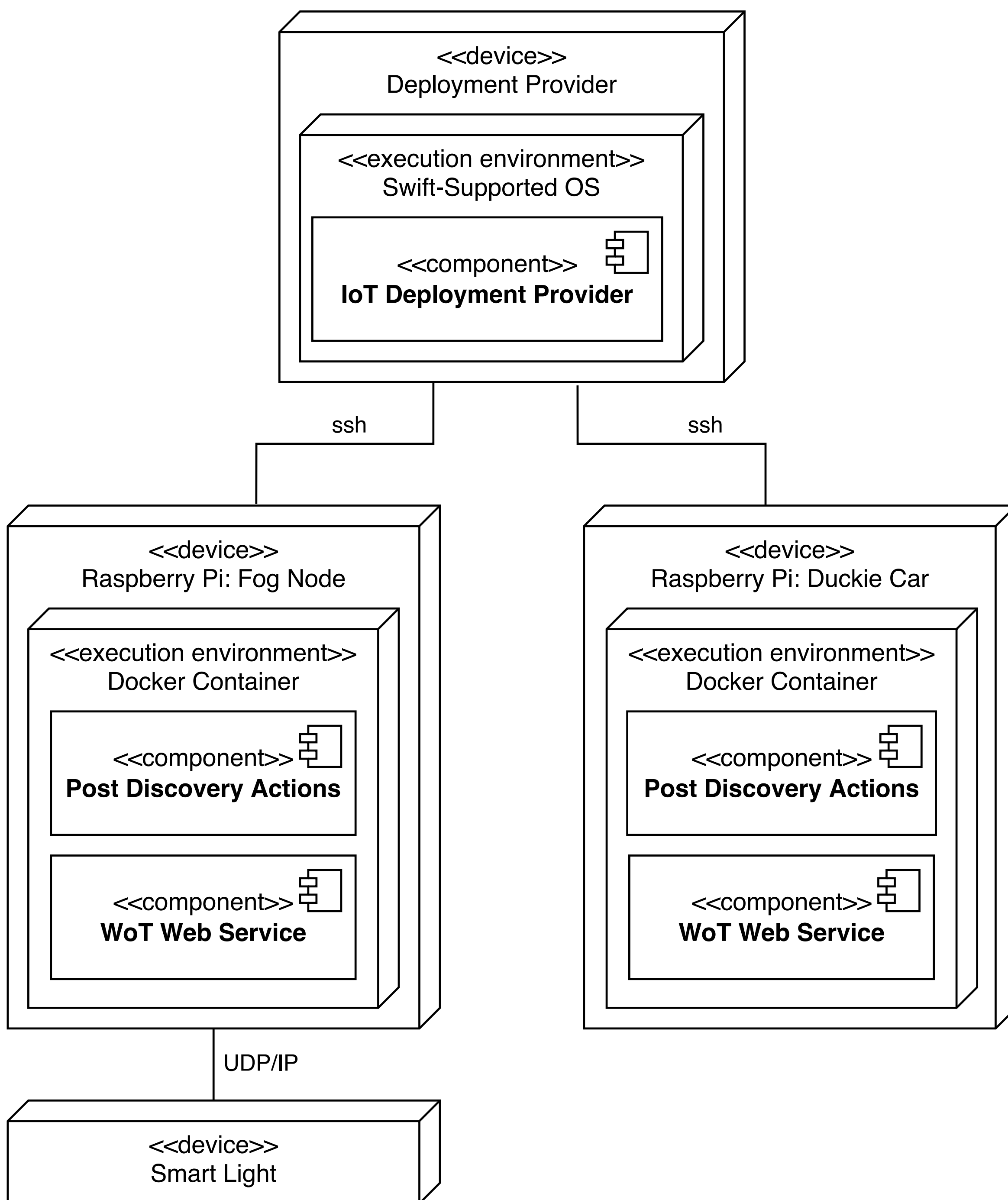


Figure 9.7: Execution environment/software/protocol mapping for the smart city IoT system, including the Apodini Deployer IoT Deployment Provider. The Deployment Provider runs on a Swift-supported operating system and uses ssh connections to the fog nodes and Duckie cars in the miniaturized smart city setup. Each fog node and Duckie car contains a Docker execution environment that can run post discovery actions and the deployed WoT web service. The fog nodes use a UDP/IP-based connections to communicate with the smart lights representing traffic lights. (UML Deployment Diagram)

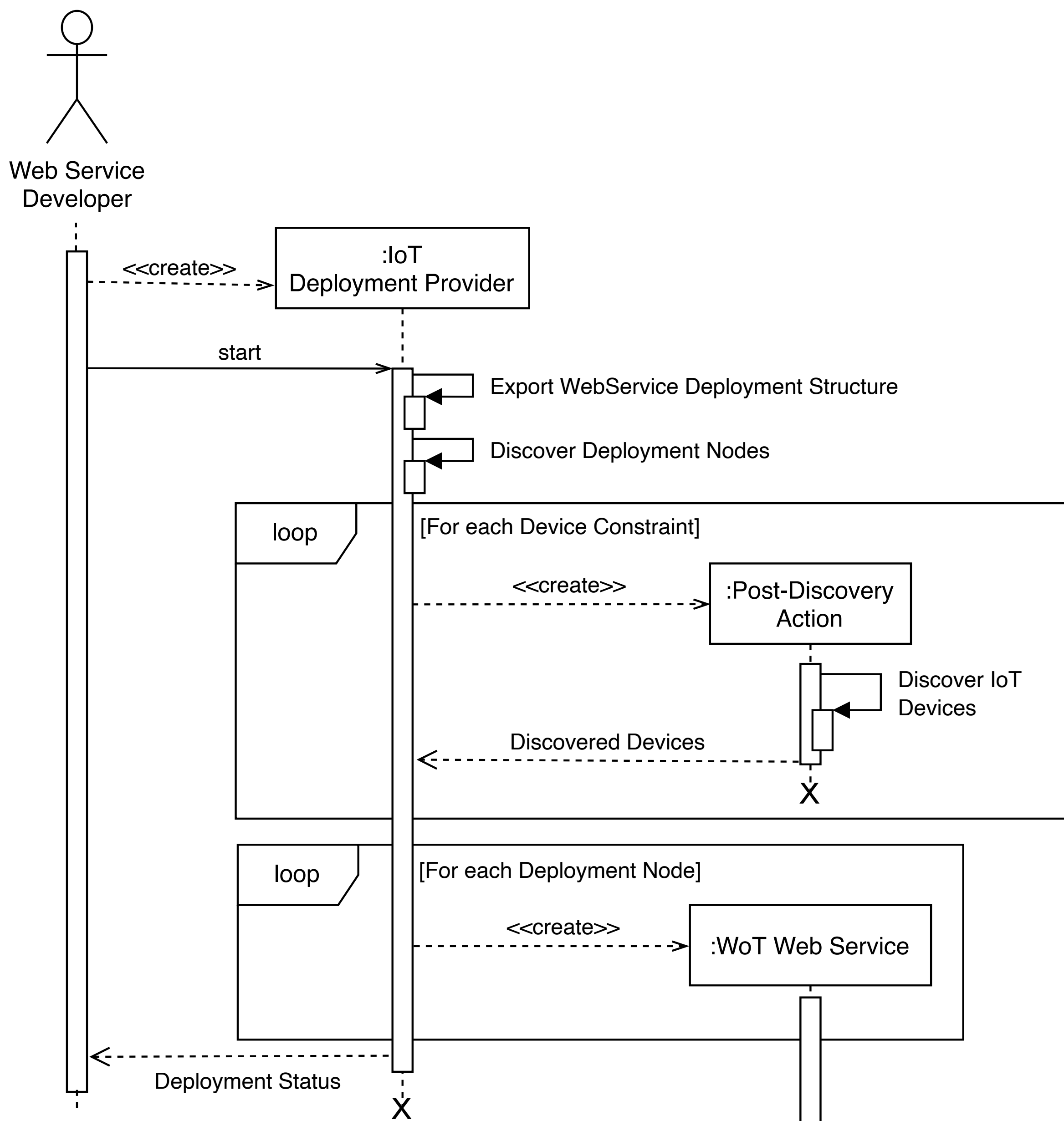


Figure 9.8: Deployment process of the IoT Deployment Provider. The web service developer starts the deployment by creating an instance of the IoT Deployment Provider that parses the deployment structure from the web service (simplified in the sequence diagram). The Deployment Provider uses several post-discovery actions for different device constraints to discover devices or execution environments shaping the deployed system structure. The IoT Deployment Provider subsequently starts up a WoT web service in each suitable execution environment and configures it according to the device constraints retrieved by the post-discovery actions. (UML Sequence Diagram)

The Swift-based Apodini DSL was used to build and deploy WoT web services in the fog-based architecture of the smart city infrastructure setup. The primary use case of the WoT web services was the control of the smart lamps simulating traffic lights in the road system used by the DuckieBots. Students reused existing dependencies to discover and control smart lights in a Swift-based Apodini WoT web service. The IoT Deployment Provider developed by Felix Desiderato as part of the master's thesis titled *Automatic Deployment and Dynamic Reconfiguration of Web Services in Heterogeneous IoT Environments* enables the constrained-based deployment of the WoT web services [83]. The IoT Deployment Provider was retroactively validated using the JASS 2021 smart city infrastructure setup. The setup is demonstrated by a simplified validation[48] setup using Raspberry Pi-based fog nodes offering WiFi networks for the UDP/IP-enabled smart light bulbs.

Figure 9.7 displays the execution environment/software/protocol mapping for the smart city IoT system validation using the IoT Deployment Provider. The UML deployment diagram demonstrates the Raspberry Pi-based setup using fog nodes and the Duckie cars running software components in the Docker execution environment. Listing 9.12 shows the web service used in the validation, including the device-specific deployment metadata used to define deployment constraints for the IoT Deployment Provider.

Figure 9.8 demonstrates the dynamic behavior of the IoT Deployment Provider, deploying the WoT web service to the Raspberry Pi-based hardware nodes. The IoT Deployment Provider uses an mDNS-based [74, 73] device discovery[49] to identify deployment-relevant devices in the network of the Deployment Provider hosting machine. Each device constraint in the JASS 2021-based validation is associated with a post-discovery action that is performed to determine if a device is connected or a deployment-relevant device fulfills deployment-related requirements. The included `DuckiePostDiscoveryAction` uses SSH to log into potential deployment nodes and identify if the device is associated with a DuckieBot. The `LIFXPostDiscoveryAction`[50] also uses ssh to execute a Swift command-line application using a UDP-based multicast to detect smart lamps in the WiFi network created by the Raspberry Pi-based fog nodes. The IoT Deployment Provider then generates a deployed system that reflects the collected constraint-relevant information about the deployment nodes and discovered associated devices. The Deploy-

[48]The JASS 2021 smart city IoT system validation setup is available as an open-source project, including further instructions on how to replicate the setup and the functionality of the IoT Deployment Provider: https://github.com/JASS-2021/JassDeploymentProviderValidation.

[49]The Swift Device Discovery framework provides a Swift interface for the mDNS based discovery functionality used in the IoT Deployment Provider: https://github.com/Apodini/SwiftDeviceDiscovery.

[50]The LIFXPostDiscoveryAction is available as a standalone Swift Package to be reused across different IoT Deployment Provider implementations: https://github.com/JASS-2021/LIFXPostDiscoveryAction.

```swift
@main
struct JASS2021WebService: WebService {
    var configuration: Configuration {
        REST {
            OpenAPI()
        }
        ApodiniDeployer(runtimes: [IoT.self])
    }

    var content: some Component {
        Group("lifx") {
            LIFXHandler()
        }.metadata(DeploymentDevice(.lifx))
        Group("duckie") {
            DuckieBotHandler()
        }.metadata(DeploymentDevice(.duckie))
        Group("common") {
            FogNodeStatusHandler()
        }.metadata(DeploymentDevice(.default))
    }
}
```

Listing 9.12: The Swift-based Apodini smart city IoT web service demonstrates the usage of the `IoT` Deployment Provider. The groups corresponding to different functionalities of the web service are annotated with `DeploymentDevice` metadata annotations providing deployment-related constraints. The `.default` device option defined in the IoT Deployment Provider expresses that a Handler or group of Handlers should always be deployed. The `.lifx` and `.duckie` device options are limited to the IoT Deployment Provider implementation for the JASS 2021 project.

ment Provider creates a WoT web service on each deployment node that meets the specified requirements and configures the web service in accordance with the defined metadata annotations. A deployment node that is associated with LIFX smart lamps offers a web service interface enables the `.lifx` and `.default`-related Handlers defined in Listing 9.12. A WoT web service deployed to a DuckieBot enables the `.duckie` and `.default`-related Handlers defined in Listing 9.12.

In addition to a static deployment at startup time, the IoT Deployment Provider also provides the option to run the device discovery and post-discovery actions regularly. The redeployment functionality allows the Development Provider to continuously listen to changes in the deployment environment and adapt the deployment based on changes at runtime. Passing in the `--automatic-redeploy` option and changing the redeployment internal using the `--redeployment-interval` argument provides the web service developer the options to enable the automatic redeployment functionality. The Deployment Provider triggers a redeployment when changes in the device setup or with connected devices are detected.

The IoT Deployment Provider and the JASS 2021 system demonstrate the functionality of a constraint-based deployment in a miniaturized smart city IoT application domain. The constraint-based deployment and extensibility of the metadata functionality as well as the Apodini Deployer subsystem highlight solutions to the investigation defined by Knowledge Question 6. The unique DSL-based approach of developing WoT-based web services enables a deployment based on the context provided by the web service without the need to specify additional deployment artifacts simplifying web service deployment evolution. The extensible post-discovery mechanisms enable deployment evolution in the initial deployment or using the redeployment mechanisms detecting runtime changes in the deployed system. These configurations can be extended and altered based on different deployment environments and are further highlighted in the water quality measurements system demonstrated in Section 9.5. Future work could extend the functionality of the IoT Deployment Provider to benefit from existing cloud-native orchestration mechanisms such as Kubernetes to export Kubernetes configuration files instead of observing the execution environment itself.

9.5 Water Quality Measurement System

The water quality measurement system demonstrates the Apodini ecosystem's applicability to the WoT and data processing application domains. The system was developed during a two-week project course named Ferienakademie and offered by the Friedrich–Alexander University Erlangen–Nürnberg, University of Stuttgart, and the Technical University of Munich while collaborating with researchers from the University of Abomey-Calavi in Godomey, Benin. The water quality measurement system was developed as a successor of the SWARM system prototyped during a previous iteration of the two-week project-based course [165]. The goal of the SWARM and the 2021 project was to develop a water quality measurement system to continuously measure water pollution in Lake Nokoué, Benin [165]. The projects aim to contribute to goal 6 of the United Nations *The 2030 Agenda for Sustainable Development*, addressing water availability and sustainability [284].

While the SWARM system used drones to collect water samples carried to a mobile laboratory and take areal imagery, the water quality measurement system developed in 2021 uses buoys to measure the water quality continuously [165]. Researchers deploy several buoys across the area that should be monitored that contain several water quality sensors connected to microcontrollers and small Linux-based machines such as Raspberry Pis. Buoys use the LoRa (long-range) low-power wide-area network technology to communicate and share measurements in a mash network. As buoys or groups of buoys are deployed across a wide area, guaranteeing a continuous mesh network is not possible. Drones are used to fly to remote locations,

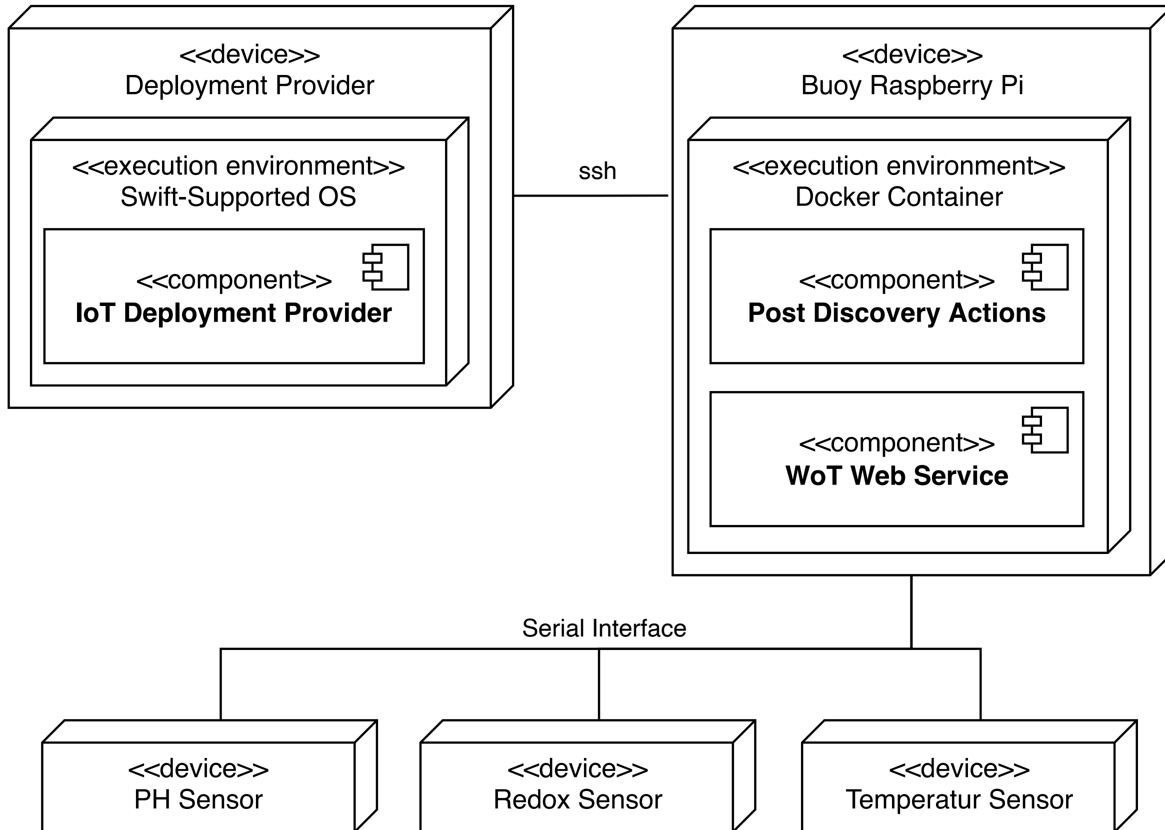


Figure 9.9: Execution environment/software/protocol mapping for the water quality measurement system. Similar to Figure 9.7, the IoT Deployment Provider is using ssh to connect to the buoy device running a Docker-based execution environment. The post discovery actions and the WoT web service are both deployed in the execution environment as detailed in Figure 9.8. The buoy sensors are connected to the buoy using serial interfaces controlled by a microcontroller that discovers the devices and collects measurements and forwards them to the WoT web service. (UML Deployment Diagram)

collect measurement data by connecting to buoys, and carry that data of one or multiple buoys back to a base station, where the data is transmitted to a data repository. The data repository is connected to a web application allowing researchers to monitor the system and export data for further analysis. Apodini is used in two parts of the system. The first part is a WoT web service running on the buoys. The web service enables computation devices carried by a drone to connect to a local WiFi network and communicate with the buoy as the pre-build drones used in the prototype project cannot directly communicate with the buoys. The second application domain is the science lab subsystem. Apodini and its extension points are used to store information in a database, provide authentication mechanisms, and continuously observe the system using observability techniques discussed in Section 5.1.2.

Figure 9.9 demonstrates the execution environment/software/protocol mapping of the buoy setup containing the WoT web service and the Apodini IoT Deployment Provider demonstrated in Section 9.4. A buoy Raspberry Pi is connected to several sensors using a microcontroller-based serial interface. The prototype developed during the Ferienakademie 2021 uses PH, reduction/oxidation (redox) potential, and temperature sensors. The IoT Deployment Provider[51] is used to automatically deploy the WoT web service to the buoys when they are in a maintenance mode and connected to a wired network. The IoT Deployment Provider uses the service discovery mechanisms and post-discovery actions to identify which sensors are connected to a buoy and automatically deploys the web services. It subsequently configures the web services based on deployment-related constraints expressed as metadata annotations in the web service. This mechanism is integrated into a continuous integration pipeline, allowing developers to push code to a source code repository that is subsequently automatically deployed to all buoys currently in maintenance mode. Changes in the sensor configuration are also automatically detected and trigger a redeployment to the changed buoys. The setup demonstrates static and dynamic deployment evolution-related capabilities of the Apodini Deployer system. It validates the approach in a WoT system consisting of several smart devices and is used in a research prototype-based water quality measurement application domain.

The science lab subsystem[52] is implemented as a Swift-based Apodini web service and connected components. Figure 9.11 displays the web application presenting the measurements and buoys in a web browser. The Apodini instantiation providing the data demonstrates the usage of the Authentication mechanisms and the database connection and observability extensions developed alongside the main Apodini DSL. Figure 9.10 describes the observability setup of the science lab subsystem incorporating components to store and analyze logs and metrics emitted from the web service. The setup is configured using a Docker Compose configuration file that describes the orchestration of the different custom build and off-the-shelf components. The ApodiniObserve target in the Apodini Swift Package enables developers to configure logging, metrics, and tracing functionalities that different configurations can extend. The ApodiniObserve functionality builds

[51]The Ferienakademie 2021 GitHub Organization contains the source code for different water quality measurement system components. Similar to Section 9.4, we provide a dedicated validation setup to test out the functionality described in this section: https://github.com/fa21-collaborative-drone-interactions/BuoyDeploymentProviderValidation.

[52]The implementation of the science lab subsystem, including the web application, web service, and Docker Compose setup, can be found at https://github.com/fa21-collaborative-drone-interactions/ScienceLabWebservice.

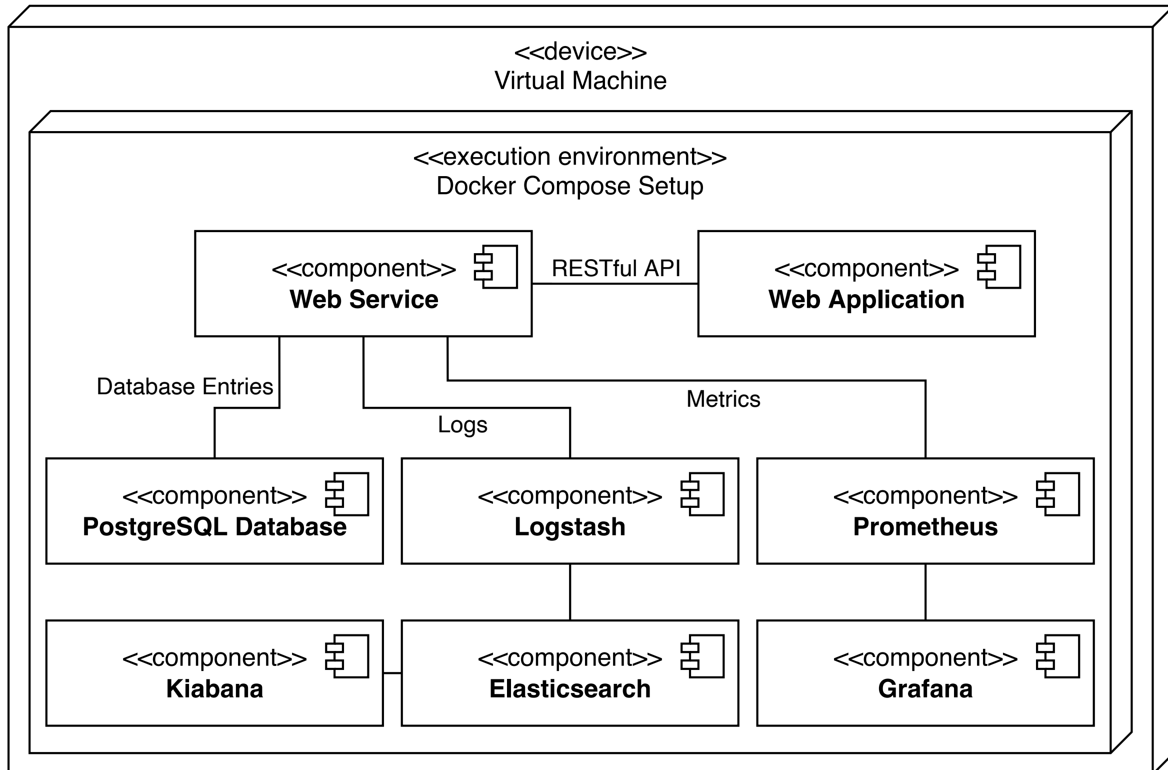


Figure 9.10: Components of the data collection web service and the web application displayed in Figure 9.11. The PostgreSQL database is used to store the data offered by the web service. The ELK-stack (Elasticsearch, Logstash, and Kiabana) stores and enables the inspection logs emitted by the Apodini web service. The Prometheus and Grafana components enable storing and querying metrics periodically pulled from the web service. (UML Deployment Diagram)

on top of the swift-logs[53], swift-metrics[54], and swift-distributed-tracing[55] libraries. The logging and metrics Apodini extensions have been developed by Philipp Zagar as part of the bachelor's thesis named *Decentralized Observability of Distributed Web Services* [318]. Apodini Observe has been extended to support distributed tracing by Moritz Sternemann in the thesis titled *Reliability and Observability of Declarative Web Services* [270]. The science lab web service uses the logs mechanisms supported by the swift-log-elk[56] library, extending swift-log to support the ELK-stack (Elasticsearch, Logstash, and Kiabana)[57] stack of tools. The ApodiniObserve-

[53]Open-sourced at https://github.com/apple/swift-log.
[54]Open-sourced at https://github.com/apple/swift-metrics.
[55]Open-sourced at https://github.com/apple/swift-distributed-tracing.
[56]The Swift Package was developed as part of the thesis *Decentralized Observability of Distributed Web Services* by Philipp Zagar and can be found at https://github.com/Apodini/swift-log-elk.
[57]The ELK-stack is a commonly used collection of tools that are used to retrieve, store, and display logs collected in a distributed system. The Elasticsearch implementation can be found at https://github.com/elastic/elasticsearch, Logstash at https://github.com/elastic/logstash, and Kiabana at https://github.com/elastic/kibana.

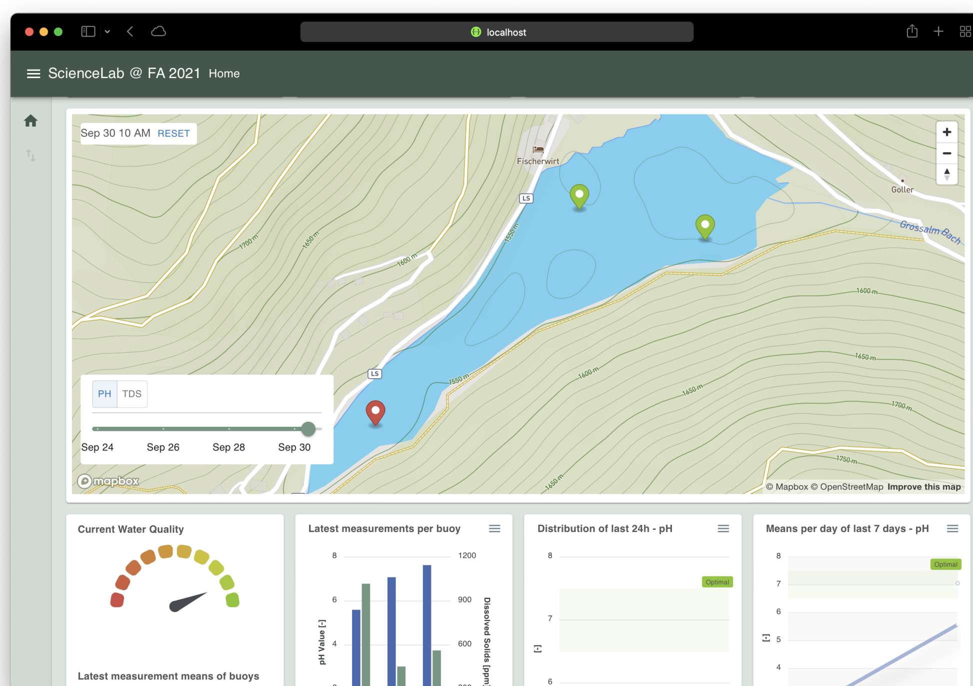


Figure 9.11: Dashboard of the water quality measurement system displaying measurements received from three buoys as created by the validation script. The dashboard provides an overview of the measurements using different graphical representations based on data retrieved from the Apodini web service.

Prometheus[58] ApodiniObserve extension is used to to provide an HTTP endpoint that the Prometheus[59] tool periodically queries to retrieve metrics recoded by the web service implementation. This information is then displayed in the Grafana[60] web application. Apodini offers web service developers automatic techniques to record commonly used metrics. In addition, Apodini provides a logger using the dependency injection mechanism provided by the Apodini environment that can be used to log messages and a metrics API that both provide information to the ApodiniObserve component.

The water quality measurement system demonstrates the deployment evolution-related functionality of the Apodini Deployer subsystem using the IoT Deployment Provider. The deployment mechanism is validated using a Raspberry Pi-based setup. The ApodiniObserve extensions demonstrate the automatic integration of observability-related functionality in the Swift-based Apodini DSL and how observability information is provided to multiple off-the-shelf components.

[58]Open-sourced at https://github.com/Apodini/ApodiniObservePrometheus.
[59]Prometheus (https://github.com/prometheus/prometheus) is a time-series database-based collection, storage, and querying component.
[60]Grafana (https://github.com/grafana/grafana) is an observability and data analysis platform that can query data from Prometheus instances or other storage components.

9.6 Summary

This chapter presents five web service instantiations that demonstrate several subsystems of the Apodini ecosystem. The single-case mechanism experiments empirically validate aspects associated with all technical research goals.

> **Technical Research Goal 1:**
> Design artifacts supporting web service API type agnostic development to enable web service interface evolution.

The projects in all sections use the Apodini DSL and the Interface Exporter subsystem to provide an API to web service clients. Section 9.1 highlights the extensibility of the Apodini DSL and the Interface Exporter Mechanisms by demonstrating several functionalities developed based on feedback provided by the project's web service developers. The section demonstrates the extensibility and applicability of the Apodini ecosystem to the application domain and the usage of the OpenAPI Interface Exporter in a project involving a mobile application.

> **Technical Research Goal 2:**
> Design artifacts enabling web service client compatibility while supporting web service API evolution.

The event management platform in Section 9.2 demonstrates the usage of the Apodini Migrator subsystem. The instantiations of the concepts described in Chapter 6 and Section 9.2 migrate 78 model and endpoint changes. The changes are automatically classified into breaking and non-breaking changes that are partially solvable by migrations provided by the Apodini Migrator subsystem.

> **Knowledge Question 4:**
> How do web API type-independent migration guides translate to client-side web API-specific migration mechanisms?

The Apodini gRPC Migrator and Apodini REST Migrator use the migration guide and web API documents to automatically generate a stable client library. The examples demonstrated in Section 9.2 demonstrate the mapping of abstract web API evolution patterns manifested in the migration guide to concrete web API migrations on the client-side. The results of the empirical validation are available as an open-source project on GitHub.

Technical Research Goal 3:
Design artifacts supporting web service deployment evolution by supplying relevant context for generating static and dynamic deployment structures.

Sections 9.3, 9.4 and 9.5 demonstrate the extensibility and functionality of the Apodini Deployer subsystem. The web services demonstrate the applicability of the annotation-based deployment and partitioning for local subprocess partitioning, FaaS-based partitioning, and WoT-based deployments. The WoT experiment setups showcase the creation of static and dynamic web service deployment structures.

Knowledge Question 6:
How do Deployment Providers enable deployment evolution in different deployment environments?

The demonstrated Deployment Providers highlight the applicability of the Deployer subsystem for different deployment-related domains. The mechanisms demonstrated in the sections show the mechanism incorporated in the subsystems, which are empirically validated by deploying the web services to different domains and observing and interpreting the result.

Part V
Epilog

THE epilog concludes the dissertation. Chapter 10 summarizes the contributions and findings and puts them in relation to the research questions, design problems, and knowledge questions defined in Part I. We provide an overview of all designed artifacts, the Apodini instantiations, and how we validated our work in Part IV. The conclusion also details future work and possible improvements further to address web service evolution during web service development. The part also contains lists of figures, tables, listings, and the bibliography.

Chapter 10

Conclusion and Future Work

The main contribution is the Apodini ecosystem, which introduces the Apodini internal domain-specific language. The Apodini DSL enables web service development independent of a web service interface or web API type. The Apodini DSL was instantiated for the Swift and Kotlin programming languages and provides several extension mechanisms to address web service evolution-related challenges.

A second contribution is a metamodel for web service interfaces, instantiated in the Apodini Interface Exporter subsystem. The metamodel describes associations between the web service, Handler, request, response, and serialization stereotypes found in all conforming web API types. The metamodel presents a UML profile containing several stereotypes augmenting UML class diagrams representing web service API types. We have identified four communication patterns, specifying the multiplicities between the request and response stereotypes. The conformance of RPC-based, message-based, and resource-based web service interface types was demonstrated by applying the metamodel and the communication patterns to UML class diagrams modeling the gRPC, GraphQL, and RESTful API types. The Interface Exporter subsystem provides a foundation and extension points to support several web service interface types and web API types. Interface Exporters retrieve a parsed structure of the web service to export Handlers to the web API type-specific concepts as demonstrated in their conformance to the metamodel. The Interface Exporter uses a shared repository pattern with several knowledge sources to collaboratively create web API representations and enable knowledge sharing between Interface Exporters. The subsystem also provides a common infrastructure for handling network requests and instantiating different communication patterns.

A third contribution is a collection of web service API evolution patterns that classify web API changes based on stereotypes defined in the web service interface metamodel. These patterns were instantiated in the Apodini Migrator subsystem. The patterns were grouped into three contexts: affecting a Handler, the request serialization, and the response serialization. We distinguished four change types

across these contexts: additions, removals, identifier changes, and context-specific changes. All changes are classified into breaking and non-breaking changes associated with migration strategies. The Apodini Migrator subsystem addresses web service API evolution with a web API type-independent migration guide that stores web API changes and migration steps. The migration guide uses an algorithm that classifies the changes according to the change patterns and provides a best-effort approach of automatically migrating changes. The best-effort migrations can be customized and extended using a JavaScript-based migration script functionality. A shared client library generation enables different Migrators to generate web API type-specific stable client libraries mitigating web API evolution.

The fourth contribution deals with web service metadata annotation. Annotations can be made on a web service, Handler, and content serialization level of the web service metamodel. The annotations allow expressing deployment-related issues such as WoT deployments, observability challenges, FaaS-based cloud deployments, and application domain-specific constraints. Based on the annotations, the Apodini Deployer subsystem enables Deployment Providers to extract metadata annotations based on the Apodini DSL. The annotations are incorporated in a web service structure that is combined with deployment environment-related constraints to generate an application domain and deployment environment-specific deployment structure. The Apodini Deployer subsystem enables the partitioning of web services into smaller components that can be deployed to IoT and FaaS environments. The cross-deployment node communication component enables deployment-agnostic Handler communication.

The dissertation demonstrated a complete design cycle. Several single-case mechanism experiments were performed to validate the Apodini ecosystem and demonstrate the web service interface evolvability-related functionality. We developed five Interface Exporter to demonstrate the applicability of the Apodini DSL and Interface Exporter mechanisms. Furthermore, we developed five systems to demonstrate the applicability of Apodini across a wide range of application domains. A future challenge is to expand the design science research project beyond the design cycle towards completing an entire engineering cycle incorporating a treatment implementation and implementation evaluation as defined by Wieringa [309].

The artifacts described in the dissertation focus on the evolution of web services. The next step would be to investigate Apodini concepts beyond web services, for example, in heterogeneous IoT systems using decentralized decision-making and communication protocols. Another challenge is to explore how the Apodini Migrator can be extended to improve the best-effort automatic migration of web API changes. Another research opportunity is to further explore the runtime reconfiguration of systems deployed using Apodini Deployer based on changes in the deployment environment with differing real-time requirements.

List of Figures

1.1 Design Cycles According to Wieringa . . . 9
1.2 Research Goal Structure According to Wieringa . . . 10
1.3 Organization of the Dissertation . . . 19

3.1 Web Service Interface Type Metamodel . . . 48
3.2 RESTful Web API Model . . . 51
3.3 GraphQL-Based Web API Model . . . 52
3.4 gRPC-Based Web API Model . . . 53

5.1 Web Service Metadata Annotation Model . . . 84

6.1 Top Level Design . . . 93
6.2 Apodini Interface Exporter Control Flow . . . 99
6.3 Apodini Migrator Control Flow . . . 101
6.4 Apodini Deployer Control Flow . . . 102
6.5 Apodini Subsystem Decomposition . . . 104
6.6 Interface Exporter Subsystem Decomposition . . . 105
6.7 Migrator Subsystem Decomposition . . . 106
6.8 Deployer Subsystem Decomposition . . . 108

7.1 Apodini DSL Components . . . 112
7.2 Semantic Model . . . 116
7.3 Apodini Migration Guide . . . 118
7.4 Deployment Structure . . . 122

8.1 GraphiQL Web IDE Offered by the GraphQL Interface Exporter . . . 142
8.2 Swagger UI Interface Offered by the OpenAPI Exporter . . . 154

9.1 Apodini Subsystems to Web Service Instantiation Mapping . . . 158
9.2 TrainLens iOS Application User Interface . . . 160
9.3 Basketball Player Health Monitoring System Swagger UI Interface . 161
9.4 Screens of the Xpense iOS Application . . . 169
9.5 AWS Lambda Dashboard . . . 173
9.6 AWS API Gateway Dashboard . . . 174

9.7 Smart City IoT System EE/S/P Mapping 176
9.8 IoT Deployment Provider Deployment Process 177
9.9 Water Quality Measurement System EE/S/P Mapping 181
9.10 Obervability EE/S/P Mapping 183
9.11 Water Quality Measurement System Dashboard 184

List of Tables

1.1 Overview of Web API Types 7

2.1 Overview of RPC API Types 27

3.1 Handler Communication Patterns 47

4.1 Web Service API Evolution Patterns 67

Listings

7.1 Swift-Based Apodini DSL: Web Service 113
7.2 Swift-Based Apodini DSL: Handler 113
7.3 Kotlin-Based Apodini DSL: Web Service 114
7.4 Kotlin-Based Apodini DSL: Handler 115
7.5 JSON Representation of a Migration Guide 119
7.6 Apodini Migrator Configuration 120
7.7 Apodini Deployer Invokable Handler Conformance 123
7.8 Apodini Deployer Remote Handler Invocation 124

8.1 gRPC Interface Exporter Configuration 130
8.2 Protocol Buffer Specification for Single Handler 131
8.3 Exporter-Specific Modifier: gRPC Method Name 132
8.4 Apodini Handler Supporting Bidirectional Streams 132
8.5 Protocol Buffer Specification for Bidirectional Streaming Handler . . 133
8.6 Open Context Message of the Web Socket Interface Exporter 135
8.7 Parameter Mutability and Apodini Errors 136
8.8 WebSocket Message Exchange Demonstrating Parameter Mutability 137
8.9 WebSocket Message Exchange Demonstrating Apodini Errors 138
8.10 GraphQL Interface Exporter Example Web Service 139
8.11 Dependency Injection in the Kotlin-Based Apodini DSL 140
8.12 Mutating Handler Using the Apodini Environment 140
8.13 HTTP Parameter Options and Bindings in Handler 143
8.14 Client-Side Streaming Handler Using the Connection State 144
8.15 Groups and Path Parameters in the Swift-Based Apodini DSL 145
8.16 Groups and Path Parameters in the Kotlin-Based Apodini DSL 146
8.17 Curl Request to Swift-Based Apodini Web Service 146
8.18 Kotlin-based Apodini Web Service Using a REST Exporter 149
8.19 JSON Response of the Root of the Web Service 150
8.20 JSON Response to /movies of the Web Service 150
8.21 JSON Response to /movies/1 of the Web Service 150
8.22 JSON Response to /movies/1/cast of the Web Service 151
8.23 Swift-based Apodini Web Service Using a REST Exporter 152

9.1 Event Management Platform Web Service 163
9.2 Migrator Validation API Document Export Command 163
9.3 Migrator Validation Migration Guide Generation Command 164
9.4 Model-Related Web API Change Migration 164
9.5 Endpoint-Related Web API Change Migration 165
9.6 Stable Client Library Generation Command 166
9.7 Client Library Rename Model Field Migration 167
9.8 Client Library Rename Endpoint Migration 167
9.9 Swift-Based Apodini Xpense Web Service 170
9.10 Localhost Deployment Provider Execution 171
9.11 AWS Lambda Deployment Provider Execution 172
9.12 Swift-Based Apodini Smart City IoT Web Service 179

Bibliography

[1] DoD standard internet protocol. RFC 760, Information Sciences Institute, University of Southern California, January 1980.

[2] XDR: External data representation standard. RFC 1014, Sun Microsystems Inc., June 1987.

[3] RPC: Remote procedure call protocol specification. RFC 1050, Sun Microsystems Inc., April 1988.

[4] RPC: Remote procedure call protocol specification: Version 2. RFC 1057, Sun Microsystems Inc., June 1988.

[5] NFS: Network file system protocol specification. RFC 1094, Sun Microsystems Inc., March 1989.

[6] UML profile for schedulability, performance, and time specification. Standard Version 1.1, Object Management Group, January 2005.

[7] IEEE standard for developing a software project life cycle process. *IEEE Std 1074-2006 (Revision of IEEE Std 1074-1997)*, pages 1–110, July 2006.

[8] UML profile for modeling quality of service and fault tolerance characteristics and mechanisms specification. Standard Version 1.1, Object Management Group, April 2008.

[9] Common event expression: Architecture overview. Technical report, The CEE Editorial Board, May 2010.

[10] OWL 2 web ontology language document overview. W3C recommendation, W3C, December 2012.

[11] Service oriented architecture Modeling Language (SoaML) Specification. Standard Version 1.0.1, Object Management Group, May 2012.

[12] ISO/IEC/IEEE international standard - Software and systems engineering – Software testing – Part 1: Concepts and definitions. *ISO/IEC/IEEE 29119-1:2013(E)*, pages 1–64, Sep. 2013.

[13] OMG Unified Modeling Language (OMG UML). Standard Version 2.5, Object Management Group, March 2015.

[14] ISO/IEC international standard - Systems and software engineering systems and software quality requirements and evaluation (SQuaRE) — System and software quality models. *ISO/IEC 25010:2011(E) First edition 2011-03*, 2017.

[15] ISO/IEC/IEEE international standard - Systems and software engineering – Software life cycle processes. *ISO/IEC/IEEE 12207:2017(E) First edition 2017-11*, pages 1–157, 2017.

[16] OMG Meta Object Facility (MOF) core specification. Standard Version 2.5.1, Object Management Group, October 2019.

[17] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45(3):515–526, 2006.

[18] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, 2015.

[19] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Data-Centric Systems and Applications. Springer Berlin Heidelberg, 2013.

[20] M. Amundsen. *Design and Build Great Web APIs*. Pragmatic Bookshelf, 2020.

[21] V. Andrikopoulos, S. Benbernou, and M. P. Papazoglou. Managing the evolution of service specifications. In Z. Bellahsène and M. Léonard, editors, *Advanced Information Systems Engineering*, pages 359–374, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[22] The Apache Software Foundation. *Apache Avro™ 1.11.0 Specification*, October 2021.

[23] R. Arnold and S. Bohner. Impact analysis - Towards a framework for comparison. In *1993 Conference on Software Maintenance*, pages 292–301, Sep. 1993.

[24] J. Arundel and J. Domingus. *Cloud Native DevOps with Kubernetes: Building, Deploying, and Scaling Modern Applications in the Cloud*. O'Reilly Media, 2019.

[25] W. C. Athas and C. L. Seitz. Multicomputers: Message-passing concurrent computers. *Computer*, 21(8):9–24, Aug 1988.

[26] L. Aversano, G. Canfora, A. Cimitile, and A. De Lucia. Migrating legacy systems to the web: An experience report. In *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, pages 148–157, March 2001.

[27] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA ’05, pages 265–279, New York, NY, USA, 2005. Association for Computing Machinery.

[28] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, K. Govindarajan, A. Karp, H. Kuno, M. Lemon, G. Pogossiants, S. Sharma, and S. Williams. Web services conversation language (WSCL) 1.0. W3C note, W3C, March 2002.

[29] P. Baran. On distributed communications networks. *IEEE Transactions on Communications Systems*, 12(1):1–9, 1964.

[30] P. Bartalos and M. B. Blake. Engineering energy-aware web services toward dynamically-green computing. In G. Pallis, M. Jmaiel, A. Charfi, S. Graupner, Y. Karabulut, S. Guinea, F. Rosenberg, Q. Z. Sheng, C. Pautasso, and S. Ben Mokhtar, editors, *Service-Oriented Computing - ICSOC 2011 Workshops*, pages 87–96, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[31] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley Professional, 3rd edition, 2012.

[32] P. Bates. Distributed debugging tools for heterogeneous distributed systems. In *The 8th International Conference on Distributed*, pages 308–315, Los Alamitos, CA, USA, jun 1988. IEEE Computer Society.

[33] S. Battle. Boxes: black, white, grey and glass box views of web-services. Technical report, HP Laboratories Bristol, 2003.

[34] A. Bauer. Requirements traceability for web services. Bachelor’s thesis, Technical University of Munich, 2021.

[35] A. Bauer. Change impact analysis of web API evolution. Guided research, Technical University of Munich, 2022.

[36] J. Bean. *SOA and Web Services Interface Design: Principles, Techniques, and Standards*. Morgan Kaufmann, 2009.

[37] T. Bellwood, S. Capell, L. Clement, J. Colgrave, M. J. Dovey, D. Feygin, A. Hately, R. Kochman, P. Macias, M. Novotny, M. Paolucci, C. von Riegen, T. Rogers, K. Sycara, P. Wenzel, and Z. Wu. UDDI version 3.0.2. OASIS standard, OASIS, October 2004.

[38] M. Belshe, R. Peon, and M. Thomson. Hypertext transfer protocol version 2 (HTTP/2). RFC 7540, May 2015.

[39] B. Benatallah, F. Casati, D. Grigori, H. R. M. Nezhad, and F. Toumani. Developing adapters for web services integration. In O. Pastor and J. Falcão e Cunha, editors, *Advanced Information Systems Engineering*, pages 415–429, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[40] S. Bernardi and D. C. Petriu. Comparing two UML profiles for non-functional requirement annotations: the SPT and QoS profiles. *SVERTS – Specification and Validation of Real-Time and Embedded Systems*, 2004.

[41] T. J. Berners-Lee. Information management: A proposal. 1989.

[42] T. J. Berners-Lee, R. Cailliau, J.-F. Groff, and B. Pollermann. World-wide web: The information universe. *Electronic Networking*, 2(1):52–58, 1992.

[43] T. J. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[44] M. Biehl. *API Architecture: The Big Picture for Building APIs*. API-University Series. CreateSpace Independent Publishing Platform, 2015.

[45] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984.

[46] J. Bogner. *On the evolvability assurance of microservices: Metrics, scenarios, and patterns*. PhD thesis, University of Stuttgard, 2020.

[47] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. Assuring the evolvability of microservices: Insights into industry practices and challenges. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 546–556, Sep. 2019.

[48] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. Microservices in industry: Insights into technologies, characteristics, and software quality. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 187–195, March 2019.

[49] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. Industry practices and challenges for the evolvability assurance of microservices. *Empirical Software Engineering*, 26(5):104, 2021.

[50] J. Bogner, S. Wagner, and A. Zimmermann. Using architectural modifiability tactics to examine evolution qualities of service- and microservice-based systems. *SICS Software-Intensive Cyber-Physical Systems*, 34(2):141–149, 2019.

[51] G. Bondel, A. Landgraf, and F. Matthes. API management patterns for public, partner, and group web API initiatives with a focus on collaboration. In *26th European Conference on Pattern Languages of Programs*, EuroPLoP'21, New York, NY, USA, 2021. Association for Computing Machinery.

[52] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu. *Fog Computing: A Platform for Internet of Things and Analytics*, pages 169–186. Springer International Publishing, Cham, 2014.

[53] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. Association for Computing Machinery.

[54] J. Bosch. *Continuous Software Engineering*. Springer, 2014.

[55] T. Boubez, Ü. Yalçinalp, M. Hondo, A. Vedamuthu, D. Orchard, F. Hirsch, and P. Yendluri. Web services policy 1.5 - Primer. W3C note, W3C, November 2007.

[56] A. Bouguettaya, M. Singh, M. Huhns, Q. Z. Sheng, H. Dong, Q. Yu, A. G. Neiat, S. Mistry, B. Benatallah, B. Medjahed, M. Ouzzani, F. Casati, X. Liu, H. Wang, D. Georgakopoulos, L. Chen, S. Nepal, Z. Malik, A. Erradi, Y. Wang, B. Blake, S. Dustdar, F. Leymann, and M. Papazoglou. A service computing manifesto: The next 10 years. *Commun. ACM*, 60(4):64–72, March 2017.

[57] A. Brito, L. Xavier, A. Hora, and M. T. Valente. APIDiff: Detecting API breaking changes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 507–511, March 2018.

[58] G. Brito, T. Mombach, and M. T. Valente. Migrating to GraphQL: A practical assessment. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 140–150, 2019.

[59] F. P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Publishing Company, 1975.

[60] R. Brooks. Using a behavioral theory of program comprehension in software engineering. In *Proceedings of the 3rd International Conference on Software Engineering*, ICSE '78, pages 196–201. IEEE Press, 1978.

[61] W. J. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. ITPro collection. Wiley, 1998.

[62] B. Bruegge and A. H. Dutoit. *Object Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall, 3rd edition, 2010.

[63] B. Bruegge, T. Gottschalk, and B. Luo. A framework for dynamic program analyzers. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 65–82, New York, NY, USA, 1993. Association for Computing Machinery.

[64] B. Bruegge, S. Krusche, and L. Alperowitz. Software engineering project courses with industrial clients. *ACM Trans. Comput. Educ.*, 15(4), dec 2015.

[65] B. Burns, J. Beda, and K. Hightower. *Kubernetes: Up and Running: Dive into the Future of Infrastructure*. O'Reilly Media, 2nd edition, 2019.

[66] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, 14(1):70–93, jan 2016.

[67] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. Number Volume 4. John Wiley & Sons, 2007.

[68] L. M. Cameron. What to know about the scientist who invented the term "software engineering". *Software Magazine*, June 2018.

[69] E. Cano. Automated generation of machine-readable migration guides for web services. Master's thesis, Technical University of Munich, 2021.

[70] C. S. Carr, S. D. Crocker, and V. G. Cerf. New host-host protocol. RFC 33, February 1970.

[71] V. Cerf, Y. Dalal, and C. Sunshine. Specification of internet transmission control program. RFC 675, December 1974.

[72] H.-M. Chen and R. Kazman. Architecting ultra-large-scale green information systems. In *2012 First International Workshop on Green and Sustainable Software (GREENS)*, pages 69–75, 2012.

[73] S. Cheshire and M. Krochmal. DNS-based service discovery. RFC 6763, February 2013.

[74] S. Cheshire and M. Krochmal. Multicast DNS. RFC 6762, February 2013.

[75] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. W3C note, W3C, March 2001.

[76] A. A. Chuvakin, K. J. Schmidt, and C. Phillips. *Logging and Log Management: The Authoritative Guide to Understanding the Concepts Surrounding Logging and Log Management*. Elsevier Science, 2012.

[77] G. Cormode and B. Krishnamurthy. Key differences between web 1.0 and web 2.0. *First Monday*, 13(6), Apr. 2008.

[78] R. Daigneau. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley signature series Service design patterns. Addison-Wesley, 2012.

[79] D. W. Davies, K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A digital communication network for computers giving rapid response at remote terminals. In *Proceedings of the First ACM Symposium on Operating System Principles*, SOSP '67, pages 2.1–2.17, New York, NY, USA, 1967. Association for Computing Machinery.

[80] A. De Lucia, R. Oliveto, F. Zurolo, and M. Di Penta. Improving comprehensibility of source code via traceability information: A controlled experiment. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 317–326, 2006.

[81] M. De Sanctis, K. Geihs, A. Bucchiarone, G. Valetto, A. Marconi, and M. Pistore. Distributed service co-evolution based on domain objects. In A. Norta, W. Gaaloul, G. R. Gangadharan, and H. K. Dam, editors, *Service-Oriented Computing – ICSOC 2015 Workshops*, pages 48–63, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[82] S. Deering and R. Hinden. Internet protocol, version 6 (IPv6) specification. RFC 8200, July 2017.

[83] F. Desiderato. Automatic deployment and dynamic reconfiguration of web services in heterogeneous IoT environments. Master's thesis, Technical University of Munich, 2021.

[84] D. Dig and R. Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, 2006.

[85] E. W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, October 1972.

[86] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle. The web of things: Interconnecting devices with high usability and performance. In *2009 International Conference on Embedded Software and Systems*, pages 323–330, 2009.

[87] D. Dzvonyar, L. Alperowitz, D. Henze, and B. Bruegge. Team composition in software engineering project courses. In *2018 IEEE/ACM International Workshop on Software Engineering Education for Millennials (SEEM)*, pages 16–23, 2018.

[88] N. E. Elyacoubi, F.-Z. Belouadha, and O. Roudies. A metamodel of WSDL web services using SAWSDL semantic annotations. In *2009 IEEE/ACS International Conference on Computer Systems and Applications*, pages 653–659, May 2009.

[89] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. The Pearson Service Technology Series from Thomas Erl. Pearson Education, 2005.

[90] T. Erl. *Service-Oriented Architecture: Analysis and Design for Services and Microservices*. The Prentice Hall Service Technology Series from Thomas Erl. Prentice Hall, 2nd edition, 2017.

[91] T. Espinha, A. Zaidman, and H.-G. Gross. Web API growing pains: Loosely coupled yet strongly tied. Technical report, Delft University of Technology - Software Engineering Research Group, 2014.

[92] T. Espinha, A. Zaidman, and H.-G. Gross. Web API growing pains: Stories from client developers and their code. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 84–93, Feb 2014.

[93] Facebook. *GraphQL*, June 2018.

[94] J. Farrell and H. Lausen. Semantic annotations for WSDL and XML schema. W3C recommendation, W3C, August 2007.

[95] R. Fatoohi, V. Gunwani, Q. Wang, and C. Zheng. Performance evaluation of middleware bridging technologies. In *2000 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS (Cat. No.00EX422)*, pages 34–39, 2000.

[96] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[97] B. Fitzgerald and K.-J. Stol. Continuous software engineering and beyond: Trends and challenges. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, RCoSE 2014, pages 1–9, New York, NY, USA, 2014. Association for Computing Machinery.

[98] B. Fitzgerald and K.-J. Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176–189, 2017.

[99] M. Fokaefs. WSDarwin: A framework for the support of web service evolution. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 668–668, Sep. 2014.

[100] M. Fokaefs, R. Mikhaiel, N. Tsantalis, E. Stroulia, and A. Lau. An empirical study on web service evolution. In *2011 IEEE International Conference on Web Services*, pages 49–56, 2011.

[101] M. Fokaefs, M. Oprescu, and E. Stroulia. WSDarwin: A web application for the support of REST service evolution. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 336–338, Sep. 2015.

[102] M. Fokaefs and E. Stroulia. WSDarwin: Automatic web service client adaptation. In *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '12, pages 176–191, USA, 2012. IBM Corp.

[103] M. Fokaefs and E. Stroulia. WSMeta: A meta-model for web services to compare service interfaces. In *Proceedings of the 17th Panhellenic Conference on Informatics*, PCI '13, pages 1–8, New York, NY, USA, 2013. Association for Computing Machinery.

[104] M. Fokaefs and E. Stroulia. *WSDarwin: Studying the Evolution of Web Service Systems*, pages 199–223. Springer New York, New York, NY, 2014.

[105] M. Fokaefs and E. Stroulia. Using WADL specifications to develop and maintain REST client applications. In *2015 IEEE International Conference on Web Services*, pages 81–88, June 2015.

[106] M.-E. Fokaefs. *WSDarwin: A Comprehensive Framework for Supporting Service-Oriented Systems Evolution*. PhD thesis, University of Alberta, 2015.

[107] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Signature Series. Pearson Education, 2010.

[108] A. C. Franco da Silva, U. Breitenbücher, P. Hirmer, K. Képes, O. Kopp, F. Leymann, B. Mitschang, and R. Steinke. Internet of things out of the box: Using TOSCA for automating the deployment of IoT environments. In *Proceedings of the 7th International Conference on Cloud Computing and Services Science*, CLOSER 2017, pages 358–367, Setubal, PRT, 2017. SCITEPRESS - Science and Technology Publications, Lda.

[109] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994.

[110] M. Garriga and A. Flores. Standards-driven metamodel to increase retrievability of heterogeneous services. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 2507–2514, New York, NY, USA, 2019. Association for Computing Machinery.

[111] D. Ghosh. *DSLs in Action*. Manning, 2011.

[112] J. Gilbert. *Software Architecture Patterns for Serverless Systems: Architecting for innovation with events, autonomous services, and micro frontends*. Packt Publishing, 2021.

[113] T. Gîrba and S. Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(3):207–236, 2006.

[114] Google. *Protocol Buffers Version 3 Language Specification*, December 2020.

[115] W. Goralski. *The Illustrated Network: How TCP/IP Works in a Modern Network*. Elsevier Science, 2nd edition, 2017.

[116] O. Gotel and C. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of IEEE International Conference on Requirements Engineering*, pages 94–101, 1994.

[117] R. B. Grady. *Practical software metrics for project management and process improvement*. Prentice Hall, 1992.

[118] R. B. Grady and D. L. Caswell. *Software Metrics: Establishing a Company-wide Program*. Prentice-Hall, 1987.

[119] J. Gregorio, R. Fielding, M. Hadley, M. Nottingham, and D. Orchard. URI template. RFC 6570, March 2012.

[120] W. Grosso. *Java RMI*. O'Reilly Media, Sebastopol, CA, October 2001.

[121] D. Guinard. *A Web of Things Application Architecture. Integrating the Real-World into the Web*. PhD thesis, ETH Zurich, Zürich, 2011.

[122] D. Guinard and V. Trifa. Towards the web of things: Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences)*, Madrid, Spain, April 2009.

[123] D. Guinard, V. Trifa, F. Mattern, and E. Wilde. *From the Internet of Things to the Web of Things: Resource-oriented Architecture and Best Practices*, pages 97–129. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[124] D. Guinard, V. Trifa, and E. Wilde. A resource oriented architecture for the web of things. In *2010 Internet of Things (IOT)*, pages 1–8, 2010.

[125] M. Hadley. Web application description language. W3C member submission, W3C, August 2009.

[126] V. Hartig. Semi-automated system decomposition using static and dynamic code analysis. Bachelor's thesis, Technical University of Munich, 2020.

[127] J. Henkel and A. Diwan. CatchUp! capturing and replaying refactorings to support API evolution. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 274–283, New York, NY, USA, 2005. Association for Computing Machinery.

[128] D. Henze. *Dynamically Scalable Fog Architectures*. PhD thesis, Technical University of Munich, Munich, 2020.

[129] D. Henze, P. Schmiedmayer, and B. Bruegge. Fog horizons – A theoretical concept to enable dynamic fog architectures. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, UCC'19, pages 41–50, New York, NY, USA, 2019. Association for Computing Machinery.

[130] A. Hernandez-Mendez, N. Scholz, and F. Matthes. A model-driven approach for generating RESTful web services in single-page applications. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*, MODELSWARD 2018, pages 480–487, Setubal, PRT, 2018. SCITEPRESS - Science and Technology Publications, Lda.

[131] J. Higginbotham. *Designing Great Web APIs: Creating Business Value Through Developer Experience*. O'Reilly Media, Inc., 2015.

[132] J. Higginbotham. *Principles of Web API Design: Delivering Value with APIs and Microservices*. Addison-Wesley Professional, 2021.

[133] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley Object Technology Series. Addison-Wesley, 2000.

[134] E. Hull, K. Jackson, and J. Dick. *Requirements Engineering*. Springer, third edition edition, 2011.

[135] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010.

[136] K. Hur, S. Chun, X. Jin, and K.-H. Lee. Towards a semantic model for automated deployment of IoT services across platforms. In *2015 IEEE World Congress on Services*, pages 17–20, 2015.

[137] K. Hur, X. Jin, and K.-H. Lee. Automated deployment of iot services based on semantic description. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 40–45, 2015.

[138] K. Indrasiri and D. Kuruppu. *gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes*. O'Reilly Media, 2020.

[139] M. Iorga, N. G. Goren, L. Feldman, R. Barton, M. Martin, and C. Mahmoudi. Fog computing conceptual model. Special Publication 500-325, National Institute of Standards and Technology, Gaithersburg, MD, March 2018.

[140] J. Islam, E. Harjula, T. Kumar, P. Karhula, and M. Ylianttila. Docker enabled virtualized nanoservices for local IoT edge networks. In *2019 IEEE Conference on Standards for Communications and Networking (CSCN)*, pages 1–7, 2019.

[141] J. Islam, T. Kumar, I. Kovacevic, and E. Harjula. Resource-aware dynamic service deployment for local IoT edge computing: Healthcare use case. *IEEE Access*, 9:115868–115884, 2021.

[142] Information technology - Open systems interconnection - Basic reference model: The basic model. Standard, International Organization for Standardization, Geneva, CH, 1994.

[143] A. Javaloyes, J. M. Sarabia, R. P. Lamberts, and M. Moya-Ramon. Training prescription guided by heart-rate variability in cycling. *International Journal of Sports Physiology and Performance*, 14(1):23 – 32, 2019.

[144] J. O. Johanßen. *Continuous User Understanding in Software Evolution*. PhD thesis, Technical University of Munich, 2019.

[145] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, 2019.

[146] M. Jones and D. Hardt. The OAuth 2.0 authorization framework: Bearer token usage. RFC 6750, October 2012.

[147] D. Jordan and J. Evdemon. Web services business process execution language version 2.0. OASIS standard, OASIS, April 2007.

[148] R. E. Kalman. On the general theory of control systems. *IFAC Proceedings Volumes*, 1(1):491–502, 1960.

[149] P. Kaminski, M. Litoiu, and H. Müller. A design technique for evolving web services. In *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '06, pages 23–es, USA, October 2006. IBM Corp.

[150] S. Karumuri, F. Solleza, S. Zdonik, and N. Tatbul. Towards observability data management at scale. *SIGMOD Rec.*, 49(4):18–23, March 2021.

[151] D. Kchaou, N. Bouassida, and H. Ben-Abdallah. WS-UML: A UML profile for web service applications. In *Proceedings of the Third International Conference on Innovation and Information and Communication Technology*, ISIICT'09, page 7, Swindon, GBR, 2009. BCS Learning & Development Ltd.

[152] M. Kelly. JSON hypertext application language. Internet-Draft draft-kelly-json-hal-08, IETF Secretariat, May 2016.

[153] S. Kennedy, O. Molloy, R. Stewart, P. Jacob, M. Maleshkova, and F. Doheny. A semantically automated protocol adapter for mapping SOAP web services to RESTful HTTP format to enable the web infrastructure, enhance web service interoperability and ease web service migration. *Future Internet*, 4(2):372–395, 2012.

[154] S. Kennedy, R. Stewart, P. Jacob, and O. Molloy. StoRHm: a protocol adapter for mapping SOAP based web services to RESTful HTTP format. *Electronic Commerce Research*, 11(3):245–269, 2011.

[155] G. Kim, J. Humble, P. Debois, and J. Willis. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. ITpro collection. IT Revolution Press, 2016.

[156] M. Kleehaus, Ö. Uludag, and F. Matthes. Towards a continuous feedback loop for service-oriented environments. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 126–134, 2018.

[157] A. Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 1 edition, 2008.

[158] R. Koçi, X. Franch, P. Jovanovic, and A. Abelló. Classification of changes in API evolution. In *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 243–249, Oct 2019.

[159] L. Kollmer. Automated and user-configurable deployment of web services. Bachelor's thesis, Technical University of Munich, 2021.

[160] L. Kollmer. Declarative development of interface-type-agnostic web services. Master's thesis, Technical University of Munich, Unpublished.

[161] S. Kotstein and J. Bogner. Which RESTful API design rules are important and how do they improve software quality? A delphi study with industry experts. In J. Barzen, editor, *Service-Oriented Computing*, pages 154–173, Cham, 2021. Springer International Publishing.

[162] P. J. Kraft. Decentralized observability using distributed user interface generation. Guided research, Technical University of Munich, 2020.

[163] N. Kratzke. A lightweight virtualization cluster reference architecture derived from open source PaaS platforms. *Open Journal of Mobile Computing and Cloud Computing (MCCC)*, 1(2):17–30, 2014.

[164] S. Krusche, B. Bruegge, I. Camilleri, K. Krinkin, A. Seitz, and C. Wöbker. Chaordic learning: A case study. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET)*, pages 87–96, May 2017.

[165] J. Kunze, V. Mayer, L. Thiergart, S. Javed, P. Scheppe, T. Tran, M. Haug, M. Avezum, B. Bruegge, and E. C. Ezin. Towards SWARM: A smart water monitoring system. In *2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)*, volume 1, pages 332–337, 2020.

[166] M. Lanthaler, D. Wood, and R. Cyganiak. RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C, February 2014.

[167] K. Lawrence and C. Kaler. Web services security: SOAP message security 1.1. OASIS standard, OASIS, February 2004.

[168] K. Lawrence and C. Kaler. WS-SecureConversation 1.4. OASIS standard, OASIS, February 2009.

[169] K. Lawrence and C. Kaler. WS-Trust 1.4. OASIS standard, OASIS, April 2012.

[170] P. J. Leach, M. Mealling, and R. Salz. A universally unique identifier (UUID) URN namespace. RFC 4122, July 2005.

[171] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[172] M. M. Lehman and J. F. Ramil. Software evolution and software evolution processes. *Annals of Software Engineering*, 14(1):275–309, 2002.

[173] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution-the nineties view. In *Proceedings Fourth International Software Metrics Symposium*, pages 20–32, 1997.

[174] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar. End-to-end versioning support for web services. In *2008 IEEE International Conference on Services Computing*, volume 1, pages 59–66, July 2008.

[175] F. Leymann. Web services flow language (WSFL 1.0). Technical report, IBM Software Group, May 2001.

[176] F. Li, M. Vögler, M. Claeßens, and S. Dustdar. Towards automated IoT application deployment by a cloud-based approach. In *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*, pages 61–68, 2013.

[177] J. Li, Y. Xiong, X. Liu, and L. Zhang. How does web service API evolution affect clients? In *2013 IEEE 20th International Conference on Web Services*, pages 300–307, June 2013.

[178] J. C. R. Licklider. Man-computer symbiosis. *IRE Transactions on Human Factors in Electronics*, HFE-1:4–11, March 1960.

[179] J. C. R. Licklider and R. W. Taylor. The computer as a communication device. *Science and Technology*, 1968.

[180] P. Lipton and S. Moser. Topology and orchestration specification for cloud applications version 1.0. OASIS standard, OASIS, November 2013.

[181] K. Lougheed and J. Rekhter. Border gateway protocol (BGP). RFC 1105, June 1989.

[182] D. Lübke, O. Zimmermann, C. Pautasso, U. Zdun, and M. Stocker. Interface evolution patterns: Balancing compatibility and extensibility across service life cycles. In *Proceedings of the 24th European Conference on Pattern Languages of Programs*, EuroPLop '19, New York, NY, USA, 2019. Association for Computing Machinery.

[183] P. Mäder and J. Cleland-Huang. A visual language for modeling and executing traceability queries. *Software & Systems Modeling*, 12(3):537–553, 2013.

[184] P. Mäder and A. Egyed. Do developers benefit from requirements traceability when evolving and maintaining a software system? *Empirical Software Engineering*, 20(2):413–441, 2015.

[185] P. Mäder and O. Gotel. Towards automated traceability maintenance. *Journal of Systems and Software*, 85(10):2205–2227, 2012.

[186] P. Mäder, O. Gotel, and I. Philippow. Getting back to basics: Promoting the use of a traceability information model in practice. In *2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, pages 21–25, 2009.

[187] M. Massé. *REST API Design Rulebook*. Oreilly and Associate Series. O'Reilly Media, 2011.

[188] P. Mell and T. Grance. The NIST definition of cloud computing. Special Publication 800-145, National Institute of Standards and Technology, Gaithersburg, MD, September 2011.

[189] A. Melnikov and I. Fette. The WebSocket protocol. RFC 6455, December 2011.

[190] R. Mikhaiel, G. Lin, and E. Stroulia. Simplicity in RNA secondary structure alignment: Towards biologically plausible alignments. In *Sixth IEEE Symposium on BioInformatics and BioEngineering (BIBE'06)*, pages 149–158, Oct 2006.

[191] J.-J. Moreau, H. F. Nielsen, M. Gudgin, M. Hadley, N. Mendelsohn, A. Karmarkar, and Y. Lafon. SOAP version 1.2 part 1: Messaging framework (second edition). W3C recommendation, W3C, April 2007.

[192] J.-J. Moreau, S. Weerawarana, R. Chinnici, and A. Ryman. Web services description language (WSDL) version 2.0 part 1: Core language. W3C recommendation, W3C, June 2007.

[193] K. Morris. *Infrastructure as Code: Managing Servers in the Cloud*. Safari Books Online. O'Reilly Media, 2016.

[194] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen. *Microservice Architecture*. O'Reilly Media, Inc., 1st edition edition, 2016.

[195] NASA Inventions and Contributions Board. The NASA heritage of creativity. *Annual Report of the NASA Inventions & Contributions Board*, 2003.

[196] P. Naur and B. Randell, editors. *Software engineering: Report of a conference sponsored by the NATO Science Committee*. Scientific Affairs Division, NATO, Brussels, Jan 1969.

[197] B. J. Nelson. *Remote Procedure Call*. PhD thesis, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304, May 1981.

[198] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., 2nd edition edition, August 2021.

[199] S. Niedermaier, F. Koetter, A. Freymann, and S. Wagner. On observability and monitoring of distributed systems – An industry interview study. In S. Yangui, I. Bouassida Rodriguez, K. Drira, and Z. Tari, editors, *Service-Oriented Computing*, pages 36–52, Cham, 2019. Springer International Publishing.

[200] J. Nielsen. *Usability Engineering*. Interactive Technologies. Elsevier Science, 1994.

[201] M. Nygard. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2nd edition, 2018.

[202] M. Obermeier. Improving runtime performance and maintainability of the apodini server-side swift framework. Bachelor's thesis, Technical University of Munich, 2021.

[203] H. H. Olsson, H. Alahyari, and J. Bosch. Climbing the "stairway to heaven": A mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 392–399, 2012.

[204] D. Orchard, H. Haas, J.-J. Moreau, S. Weerawarana, A. Lewis, and R. Chinnici. Web services description language (WSDL) version 2.0 part 2: Adjuncts. W3C recommendation, W3C, June 2007.

[205] R. Orfali and D. Harkey. *Client/Server Programming with Java and CORBA*. Wiley, 1997.

[206] E. G. Ormeño, M. I. Lund, L. N. Aballay, and S. Aciar. An UML profile for modeling RESTful services. In *13th Argentine Symposium on Software Engineering, ASSE 2012*, 2012.

[207] G. Ortiz and J. Hernandez. A case study on integrating extra-functional properties in web service model-driven development. In *Second International Conference on Internet and Web Applications and Services (ICIW'07)*, pages 35–35, May 2007.

[208] G. Ortiz and J. Hernandez. Toward UML profiles for web services and their extra-functional properties. In *2006 IEEE International Conference on Web Services (ICWS'06)*, pages 889–892, 2006.

[209] C. Pahl. Containerization and the PaaS cloud. *IEEE Cloud Computing*, 2(3):24–31, May 2015.

[210] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi. Cloud container technologies: A state-of-the-art review. *IEEE Transactions on Cloud Computing*, 7(3):677–692, 2019.

[211] M. P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering, 2003. WISE 2003.*, pages 3–12, Dec 2003.

[212] A. Parker, D. Spoonhower, J. Mace, R. Isaacs, and B. Sigelman. *Distributed Tracing in Practice*. O'Reilly Media, Inc., April 2020.

[213] D. L. Parnas. Information distribution aspects of design methodology. February 1971.

[214] A. A. Patil, S. A. Oundhakar, A. P. Sheth, and K. Verma. METEOR-S web service annotation framework. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 553–562, New York, NY, USA, 2004. Association for Computing Machinery.

[215] C. Peng and G. Bai. Using tag based semantic annotation to empower client and REST service interaction. In *Proceedings of the 3rd International Conference on Complexity, Future Information Systems and Risk - COMPLEXIS,*, pages 64–71. INSTICC, SciTePress, 2018.

[216] F. Petrillo, P. Merle, N. Moha, and Y.-G. Guéhéneuc. Are REST APIs for cloud computing well-designed? An exploratory study. In Q. Z. Sheng, E. Stroulia, S. Tata, and S. Bhiri, editors, *Service-Oriented Computing*, pages 157–170, Cham, 2016. Springer International Publishing.

[217] D. J. Plews, P. B. Laursen, A. E. Kilding, and M. Buchheit. Evaluating training adaptation with heart-rate measures: A methodological comparison. *International Journal of Sports Physiology and Performance*, 8(6):688 – 691, 2013.

[218] D. J. Plews, P. B. Laursen, J. Stanley, A. E. Kilding, and M. Buchheit. Training adaptation and heart rate variability in elite endurance athletes: Opening the door to effective monitoring. *Sports Medicine*, 43(9):773–781, 2013.

[219] E. Porcello and A. Banks. *Learning GraphQL: Declarative Data Fetching for Modern Web Apps*. O'Reilly Media, August 2018.

[220] J. Postel. File transfer protocol specification. RFC 765, June 1980.

[221] J. Postel. Telnet protocol specification. RFC 764, June 1980.

[222] J. Postel. User datagram protocol. RFC 768, August 1980.

[223] J. Postel. Internet protocol. RFC 791, September 1981.

[224] J. Postel. NCP/TCP transition plan. RFC 801, November 1981.

[225] J. Postel. Simple mail transfer protocol. RFC 788, November 1981.

[226] J. Postel. Transmission control protocol. RFC 793, September 1981.

[227] J. Postel and J. White. Procedure call documents: Version 2. RFC 674, December 1974.

[228] T. Preston-Werner. *Semantic Versioning 2.0.0*, June 2013.

[229] M. Quintero Szillat. A framework to build and maintain GraphQL client-server architectures in Swift. Master's thesis, Technical University of Munich, 2020.

[230] M. Quintero Szillat. Building protocol-agnostic server-side applications using domain-specific languages. Guided research, Technical University of Munich, 2021.

[231] F. Rademacher, M. Peters, and S. Sachweh. Design of a domain-specific language based on a technology-independent web service framework. In D. Weyns, R. Mirandola, and I. Crnkovic, editors, *Software Architecture*, pages 357–371, Cham, 2015. Springer International Publishing.

[232] V. T. Rajlich and K. H. Bennett. A staged model for the software life cycle. *Computer*, 33(7):66–71, 2000.

[233] B. Ramesh, T. Powers, C. Stubbs, and M. Edwards. Implementing requirements traceability: A case study. In *Proceedings of 1995 IEEE International Symposium on Requirements Engineering (RE'95)*, pages 89–95, 1995.

[234] P. Ravi, P. Chinnaiah, and S. A. Abbas. *Cloud Computing Technologies for Green Enterprises: Fundamentals of Cloud Computing for Green Enterprises*, pages 1–27. IGI Global, Hershey, PA, USA, 2018.

[235] E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly Open Source Series. O'Reilly, 2001.

[236] A. Rensink and A. Kleppe. On a graph-based semantics for UML class and object diagrams. *Electronic Communications of the EASST*, 10, 2008.

[237] J. Reschke. The 'Basic' HTTP authentication scheme. RFC 7617, September 2015.

[238] L. Roberts. The arpanet and computer networks. In *Proceedings of the ACM Conference on The History of Personal Workstations*, HPW '86, pages 51–58, New York, NY, USA, 1986. Association for Computing Machinery.

[239] J. Robie, R. Cavicchio, R. Sinnema, and E. Wilde. RESTful service description language (RSDL): Describing RESTful services without tight coupling. In *Proceedings of Balisage: The Markup Conference 2013*, Balisage Series on Markup Technologies. Balisage, 2013.

[240] C. Rodríguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali, and G. Percannella. REST APIs: A large-scale analysis of compliance with principles and best practices. In A. Bozzon, P. Cudre-Maroux, and C. Pautasso, editors, *Web Engineering*, pages 21–39, Cham, 2016. Springer International Publishing.

[241] D. Romano and M. Pinzger. Analyzing the evolution of web services using fine-grained changes. In *2012 IEEE 19th International Conference on Web Services*, pages 392–399, June 2012.

[242] N. Rosa, P. Cunha, and G. Justo. ProcessNFL: A language for describing non-functional properties. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, pages 3676–3685, 2002.

[243] D. Rossi. UML-based model-driven REST API development. In *Proceedings of the 12th International Conference on Web Information Systems and Technologies - Volume 1: WEBIST,*, pages 194–201. INSTICC, SciTePress, 2016.

[244] R. Schantz. Commentary on procedure calling as a network protocol. RFC 684, April 1975.

[245] B. Schmeling. *Composing Non-Functional Concerns in Web Services*. PhD thesis, Technische Universität Darmstadt, July 2013.

[246] B. Schmeling, A. Charfi, S. Heinzl, and M. Mezini. A survey on non-functional concerns in web services. *International Journal of Web Information Systems*, 8(1):5–31, Jan 2012.

[247] B. Schmeling, A. Charfi, and M. Mezini. Non-functional concerns in web services: Requirements and state of the art analysis. In *Proceedings of the 12th International Conference on Information Integration and Web-Based Applications and Services*, iiWAS '10, pages 67–74, New York, NY, USA, 2010. Association for Computing Machinery.

[248] B. Schmeling, A. Charfi, R. Thome, and M. Mezini. Composing non-functional concerns in web services. In *Proceedings of the 2011 IEEE Ninth European Con-*

ference on Web Services, ECOWS '11, pages 73–80, USA, 2011. IEEE Computer Society.

[249] P. Schmiedmayer. Apodini: An internal domain specific language to design web services. *21st International Middleware Conference Doctoral Symposium (Middleware '20 Doctoral Symposium)*, December 2020.

[250] P. Schmiedmayer, R. Chatley, J. P. Bernius, S. Krusche, K. Chaika, K. Krinkin, and B. Bruegge. Global software engineering in a global classroom. In *2022 IEEE/ACM 44rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, 2022.

[251] P. Schmiedmayer, L. M. Reimer, M. Jovanović, D. Henze, and S. Jonas. Transitioning to a large-scale distributed programming course. In *2020 IEEE 32nd Conference on Software Engineering Education and Training (CSEE&T)*, pages 1–6, 2020.

[252] V. Schreibmann and P. Braun. Model-driven development of RESTful APIs. In *Proceedings of the 11th International Conference on Web Information Systems and Technologies - Volume 1: WEBIST,*, pages 5–14. INSTICC, SciTePress, 2015.

[253] S. Schreier. Modeling RESTful applications. In *Proceedings of the Second International Workshop on RESTful Design*, WS-REST '11, pages 15–21, New York, NY, USA, 2011. Association for Computing Machinery.

[254] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, USA, 1st edition, 2001.

[255] A. H. N. Seitz. *An Architectural Style for Fog Computing: Formalization and Application*. PhD thesis, Technical University of Munich, 2019.

[256] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[257] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[258] Y. Shkuro. *Mastering Distributed Tracing*. Packt Publishing, February 2019.

[259] B. Simon, B. Goldschmidt, and K. Kondorosi. A metamodel for the web services standards. *Journal of Grid Computing*, 11(4):735–752, 2013.

[260] M. P. Singh and M. N. Huhns. *Service-Oriented Computing: Semantics, Processes, Agents*. Wiley, 2005.

[261] Z. U. Singhera and A. A. Shah. Extended web services framework to meet non-functional requirements. In *Workshop Proceedings of the Sixth International Conference on Web Engineering*, ICWE '06, pages 21–es, New York, NY, USA, 2006. Association for Computing Machinery.

[262] E. Sisinni, A. Saifullah, S. Han, U. Jennehag, and M. Gidlund. Industrial internet of things: Challenges, opportunities, and directions. *IEEE Transactions on Industrial Informatics*, 14(11):4724–4734, Nov 2018.

[263] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 156 University Ave, Palo Alto, CA, April 2007.

[264] R. Snodgrass. *Monitoring Distributed Systems: A Relational Approach*. PhD thesis, Carnegie-Mellon University, December 1982.

[265] L. Snyder and R. L. Henry. *Fluency With Information Technology*. Pearson, 7th edition edition, 2017.

[266] S. M. Sohan, C. Anslow, and F. Maurer. A case study of web API evolution. In *2015 IEEE World Congress on Services*, pages 245–252, June 2015.

[267] I. Sommerville. *Software Engineering*. Pearson, 2016.

[268] C. Sridharan. *Distributed Systems Observability*. O'Reilly Media, Inc., 2018.

[269] R. Srinivasan. RPC: Remote procedure call protocol specification version 2. RFC 1831, August 1995.

[270] M. Sternemann. Reliability and observability of declarative web services. Bachelor's thesis, Technical University of Munich, 2022.

[271] J. Strauch and S. Schreier. RESTify: From RPCs to RESTful HTTP design. In *Proceedings of the Third International Workshop on RESTful Design*, WS-REST '12, pages 11–18, New York, NY, USA, 2012. Association for Computing Machinery.

[272] A. S. Tanenbaum and M. van Steen. *Distributed systems*. Pearson, Upper Saddle River, NJ, 2 edition, October 2006.

[273] Technical Oversight Committee (TOC). *Cloud Native Definition v1.0*. Cloud Native Computing Foundation (CNCF), June 2018.

[274] Technical University of Munich. *Code of Conduct for Safeguarding Good Academic Practice and Procedures in Cases of Academic Misconduct at Technische Universität München*, July 2015.

[275] Technical University of Munich. *TUM Citation Guide*, June 2020.

[276] R. Thurlow. RPC: Remote procedure call protocol specification version 2. RFC 5531, May 2009.

[277] B. Traversat, M. Abdelaziz, D. Doolin, M. Duigou, J.-C. Hugly, and E. Pouyoul. Project JXTA-C: Enabling a web of things. In *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*, pages 9 pp.–, 2003.

[278] M. Treiber, L. Juszczyk, D. Schall, and S. Dustdar. Programming evolvable web services. In *Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems*, PESOS '10, pages 43–49, New York, NY, USA, 2010. Association for Computing Machinery.

[279] M. Treiber, H.-L. Truong, and S. Dustdar. SEMF - service evolution management framework. In *2008 34th Euromicro Conference Software Engineering and Advanced Applications*, pages 329–336, Sep. 2008.

[280] M. Treiber, H.-L. Truong, and S. Dustdar. On analyzing evolutionary changes of web services. In G. Feuerlicht and W. Lamersdorf, editors, *Service-Oriented Computing – ICSOC 2008 Workshops*, pages 284–297, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[281] M. V. Trifa. *Building Blocks for a Participatory Web of Things. Devices, Infrastructures, and Programming Frameworks*. PhD thesis, ETH Zurich, Zürich, 2011.

[282] N. Tsantalis, N. Negara, and E. Stroulia. Webdiff: A generic differencing service for software artifacts. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 586–589, Sep. 2011.

[283] UDDI Consortium. *UDDI Executive White Paper*, November 2001.

[284] United Nations. Transforming our world: The 2030 agenda for sustainable development, September 2015.

[285] B. Upadhyaya, Y. Zou, H. Xiao, J. Ng, and A. Lau. Migration of SOAP-based services to RESTful services. In *2011 13th IEEE International Symposium on Web Systems Evolution (WSE)*, pages 105–114, Sep. 2011.

[286] S. van der Burg and E. Dolstra. Automated deployment of a heterogeneous service-oriented system. In *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 183–190, 2010.

[287] S. van der Burg and E. Dolstra. Disnix: A toolset for distributed deployment. *Science of Computer Programming*, 79:52–69, 2014.

[288] H. van Vliet. *Software Engineering: Principles and Practice*. Wiley, 2000.

[289] J. M. Vara, J. Verde, V. Andrikopoulos, V. Bollati, and E. Marcos. An EMF-based toolkit for reasoning on web services evolution. In *Proceedings of the Workshop on ACadeMics Tooling with Eclipse*, ACME '13, New York, NY, USA, 2013. Association for Computing Machinery.

[290] D. Verna. *Extensible Languages: Blurring the Distinction between DSL and GPL*, pages 1–31. IGI Global, Hershey, PA, USA, 2013.

[291] W. G. Vincenti. *What Engineers Know and How They Know It. Analytical Studies from Aeronautical History*. The Johns Hopkins University Press, 1990.

[292] M. Vitali. Towards greener applications: Enabling sustainable cloud native applications design. Submitted for CAiSE 2022, LNCS, Springer, 2022.

[293] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering: Designing, Implementing and Using Domain-specific Languages*. CreateSpace Independent Publishing Platform, 2013.

[294] M. Vögler, J. Schleicher, C. Inzinger, S. Nastic, S. Sehic, and S. Dustdar. LEONORE – Large-scale provisioning of resource-constrained iot deployments. In *2015 IEEE Symposium on Service-Oriented System Engineering*, pages 78–87, 2015.

[295] M. Vögler, J. M. Schleicher, C. Inzinger, and S. Dustdar. DIANE - dynamic iot application deployment. In *2015 IEEE International Conference on Mobile Services*, pages 298–305, 2015.

[296] M. Vögler, J. M. Schleicher, C. Inzinger, and S. Dustdar. Optimizing elastic IoT application deployments. *IEEE Transactions on Services Computing*, 11(5):879–892, 2018.

[297] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[298] H. Wada, J. Suzuki, and K. Oba. A model-driven development framework for non-functional aspects in service oriented architecture. *International Journal of Web Services Research (IJWSR)*, 5(4):1–31, 2008.

[299] S. Wagner. *Software Product Quality Control*. Springer Berlin Heidelberg, 2013.

[300] S. Wagner, K. Lochmann, L. Heinemann, M. Kläs, A. Trendowicz, R. Plösch, A. Seidl, A. Goeb, and J. Streit. The Quamoco product quality modelling and

assessment approach. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1133–1142. IEEE Press, 2012.

[301] P. Walmsley and D. Fallside. XML schema part 0: Primer second edition. W3C recommendation, W3C, October 2004.

[302] S. Wang, I. Keivanloo, and Y. Zou. How do developers react to RESTful API evolution? In X. Franch, A. K. Ghose, G. A. Lewis, and S. Bhiri, editors, *Service-Oriented Computing*, pages 245–259, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[303] K. Warr, D. Davis, A. Malhotra, and W. Chou. Web services metadata exchange (WS-MetadataExchange). W3C recommendation, W3C, December 2011.

[304] A. Weinkötz. A process model for client migration after service changes in distributed systems. Master's thesis, Technical University of Munich, 2020.

[305] J. E. White. High-level framework for network-based resource sharing. RFC 707, December 1975.

[306] J. E. White. Elements of a distributed programming system. RFC 708, January 1976.

[307] J. E. White. A high-level framework for network-based resource sharing. In *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*, AFIPS '76, pages 561–570, New York, NY, USA, 1976. Association for Computing Machinery.

[308] A. Wiedemann, N. Forsgren, M. Wiesche, H. Gewald, and H. Krcmar. Research for practice: The DevOps phenomenon. *Commun. ACM*, 62(8):44–49, jul 2019.

[309] R. J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, 2014.

[310] L. Williams and A. Cockburn. Agile software development: It's about feedback and change. *Computer*, 36(6):39–43, 2003.

[311] D. Winer, S. Thatte, D. Box, G. Kakivaya, and A. Layman. SOAP: Simple object access protocol. Internet-Draft draft-box-http-soap-01, Internet Engineering Task Force, December 1999. Work in Progress.

[312] E. Wittern, A. Cha, and J. A. Laredo. Generating GraphQL-wrappers for REST(-like) APIs. In T. Mikkonen, R. Klamma, and J. Hernández, editors, *Web Engineering*, pages 65–83, Cham, 2018. Springer International Publishing.

[313] E. Wittern, A. Ying, Y. Zheng, J. A. Laredo, J. Dolby, C. C. Young, and A. A. Slominski. Opportunities in software engineering research for web API consumption. In *Proceedings of the 1st International Workshop on API Usage and Evolution*, WAPI '17, pages 7–10. IEEE Press, 2017.

[314] C. Wohlin. Case study research in software engineering—It is a case, and it is a study, but is it a case study? *Information and Software Technology*, 133:106514, 2021.

[315] Z. Xing and E. Stroulia. UMLDiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 54–65, New York, NY, USA, 2005. Association for Computing Machinery.

[316] J. Yasmin, Y. Tian, and J. Yang. A first look at the deprecation of RESTful APIs: An empirical study. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 151–161, Sep. 2020.

[317] B. Yildiz. Corvus - A declarative server-side swift framework. Bachelor's thesis, Technical University of Munich, 2020.

[318] P. Zagar. Observability of distributed web services. Bachelor's thesis, Technical University of Munich, 2021.

[319] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989.

[320] H. Zimmermann. OSI reference model - the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.

[321] W. Zuo. *Managing and modeling web service evolution in SOA architecture*. PhD thesis, Université de Lyon, July 2016.

[322] W. Zuo, Y. Amghar, and A.-N. Benharkat. The impact analysis model for web service evolution. In *2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, volume 1, pages 457–460, Dec 2015.

[323] W. Zuo, A. N. Benharkat, and Y. Amghar. Holistic and change-centric model for web service evolution. In *2014 IEEE World Congress on Services*, pages 250–253, June 2014.