NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

RODRIGO JORGE RIBEIRO

Bachelor Degree in Computer Science

SAFE API EVOLUTION IN A MICROSERVICE ARCHITECTURE WITH A PLUGGABLE AND TRANSACTIONLESS SOLUTION

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2022

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

SAFE API EVOLUTION IN A MICROSERVICE ARCHITECTURE WITH A PLUGGABLE AND TRANSACTIONLESS SOLUTION

RODRIGO JORGE RIBEIRO

Bachelor Degree in Computer Science

Adviser: Carla Ferreira
Associate Professor, FCT NOVA University Lisbon

Co-adviser: João Costa Seco
Associate Professor, FCT NOVA University Lisbon

Examination Committee

Chair: Henrique João Lopes Domingos
Associate Professor, FCT NOVA University Lisbon

Rapporteur: Alcides Miguel C. Aguiar Fonseca
Assistant Professor, Faculty of Sciences of the University of Lisbon

Member: Carla Ferreira
Associate Professor, FCT NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2022

Safe API Evolution in a Microservice Architecture with a Pluggable and Transactionless Solution

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.9.4) [30].

Abstract

In contrast to monolithic system designs, microservice architectures provide greater scalability, availability, and delivery by separating the elements of a large project into independent entities linked through a network of services. Because services are tied to one another via their interfaces, they can only evolve separately if their contracts remain consistent. There is a scarcity of mechanisms for safely evolving and discontinuing functionalities of services.

In monolithic system design's, changing the definition of an element can be accomplished quickly with the aid of developer tools (such as IDE refactoring toolkits).

In distributed systems there is a lack of comparable tools, developers are left with the burden of manually tracking down and resolving problems caused by uncontrolled updates. To ensure that microservices are working properly the general approach is to validate their behaviour through empirical tests.

This thesis aims to supplement the conventional approach by providing mechanisms that support the automatic validation of deployment operations, and the evolution of microservice interfaces. It´s presented a microservice management system that verifies the safety of modifications to service interfaces and that enables the evolution of service contracts without impacting consumer services. The system use runtime-generated proxies, that dynamically convert the data sent between services to the format expected by static code, thereby relieving the developer of the need to manually adapt existing services.

Keywords: microservices, software evolution, service compatibility, API, interface adaptation

Resumo

Em contraste com sistemas tradicionais monoliticos, as arquiteturas de microsserviços permitem grande escalabilidade, disponibilidade e capacidade de entrega, separando os elementos de um grande projeto em entidades independentes ligadas através de uma rede serviços. Como os serviços estão ligados uns aos outros através das suas interfaces, só podem evoluir separadamente se os seus contratos se mantiverem consistentes. Existe uma escassez de mecanismos para evoluir e descontinuar as funcionalidades dos serviços em segurança.

Nos sistemas tradicionais monoliticos, a alteração da definição de um elemento pode ser realizada rapidamente com a ajuda de ferramentas automatizadas (tais como kits de ferramentas de refactoring IDE). Em sistemas distribuídos, existe falta de ferramentas comparáveis, os programadores ficam normalmente sobrecarregados com a resolução manual de problemas causados por atualizações e pela validação do correcto funcionamento do sistema através de testes empíricos.

O trabalho desenvolvido nesta tese procura complementar a abordagem convencional, fornecendo mecanismos que suportam a validação das operações de deployment. É apresentado um sistema de gestão de microsserviços que verifica a segurança das modificações das interfaces de serviço e a evolução dos contratos. A abordagem utiliza proxies, que convertem dinamicamente os dados enviados entre serviços ao formato esperado pelo código de serviço estático, minimizando a intervenção manual do programador.

Palavras-chave: microsserviços, evolução de software, compatibilidade de serviços

Contents

1 Introduction

Software architecture is one of the most important factor's that influences the capabilities and constraints of a system's evolution throughout its lifecycle [9]. The subject of architecture is primarily concerned with the composition and design of the fundamental structures of a software system, with the goal of meeting functional requirements, while maximizing non-functional criteria, such as scalability, maintainability, extensibility, availability, and so on.

A software architectural model may be viewed as a macroscopic blueprint for the composition of a system elements. Element's typically fall into one of three overlapping categories: data elements, processing elements, and connecting elements. [36]. The data elements contain the state of system (eg. database, variable, file); The processing elements transform the data elements (eg. method, function, service); The connecting elements serve as the bridge that permits different processing elements to collaborate (e.g. procedure call, shared memory, network message). Software architectural models are primarily distinguished by the capabilities of the connecting elements and by the layers used to segregate elements into different subsystems. This thesis aims to enhance the connecting elements used in microservice architectures in order to fulfill stricter extensibility and availability requirements.

The field of software architecture traces back to 1960s, but it wasn't until 1992 that Perry and Wolf [36] created a firm basis on the subject. Changing the architecture of a system after its initial design is typically very costly. Software engineers have developed novel methods for composing systems that offer broad functionality and fit a diverse set of criteria, Bosch work [8, 7] offers an overview of software architectural research.

A typical example of an architectural design is the Model-View-controller (MVC) [12, 31]. The MVC pattern encourages three separate layers: The (M)odel layer manages the data of an application; The (V)iew layer handles the display of information to an end user; The (C)ontroller layer processes user input and converts it to commands for the model or view layers. These layers improve software maintainability by separating distinct concerns, however this architectural model is notorious for its lack of extensibility and scalability due to monolithic nature of each layer. Creating and modifying instances of

massive monoliths is time-consuming and error-prone; minor system updates require the complete redeployment of the application, severely limiting a system's capacity to evolve while preserving its availability.

Service Oriented Architecture (SOA) [35] is an alternative model that aims to improve the re-usability and maintainability of software by minimizing the coupling between independent system components. In SOA, components provide functionality to other components via network message passing. A component is exposed via the service abstraction, which defines a standardized contract for the message exchange and format. Microservices [31] are a modern iteration of the SOA model, that reduces the excessive and imposing architectural layers introduced in earlier generations of the SOA model.

Some advantages of service orientation over monoliths are as follows:

- Functional independence - Each component is operationally independent, components only communicate through their standardized interfaces.
- Component scalability - When a single component becomes overloaded, additional instances of the component can be deployed separately.
- Separate development - Different teams may work on different modules of the system with minimal coordination if the component interfaces are defined ahead of time.
- Flexibility - Each component can be implemented in the programming language and platform that best suits its requirements.

These advantages boost the system's ability to evolve by allowing orthogonal domain's to be disassembled into disjoint subsystems. In a microservice architecture, subsystems can fully evolve independently of one another, as long as they don't share services. Because services are tied to one another via their interfaces, they can only evolve separately if their contracts remain consistent. It is often advantageous to have the capacity to change service contracts independently, particularly in agile workflows and the early phases of development, where contracts are prone to frequent changes.

1.1 Problem Statement

The main challenge in developing applications based on microservices-based architectures is the lack of mechanisms for evaluating the safety of service contract updates.

In a monolithic system, interactions between different components are conducted through procedure calls. In a distributed system, however, each individual component has to communicate with other components across the network, without the same guarantees. Modifying a method definition and all references in a monolithic application may be done quickly with the aid of IDE refactoring tools. Equivalent tools for detecting

incompatibilities between endpoints and their consumers are scarce in distributed systems. Developers are left with the burden of manually tracking down and refactoring the system's dependencies across all consumers and producers services.

With a rising number of separate services and their interactions, contract administration and service integration become progressively more challenging. Software engineering guidelines assume a scenario of frequent service implementation changes and few contract modifications; in the context of iterative approaches to software development, contracts are typically modified as frequently as implementation details. Preserving the integrity of microservice architectures is a daunting task that can only be accomplished by the most diligent teams, for the following reasons:

The ramifications of changing a contract are discovered after the fact: Services evolve independently, and contracts are rarely formally specified or documented. The only symptoms of a broken system are runtime errors or unexpected behaviors.

The safety of a deployment operation is unknown: Established orchestration platforms can't guarantee the safety of deployments because they rely only on software packaging information (like version IDs) to manage deployments. A new model for such environments is required to ensure the safety of deployment tasks, one that uses information from contracts of previously deployed services, and employs a compatibility relation between them.

A contract change, entails the redeployment and downtime of its consumers: Modifications to a service's definition have an immediate impact on the entire system, by requiring the downtime of all consumer services. Deployments of "compatible" services should not disrupt the system soundness, one service should be able to be replaced while the remainder of the system remains operational.

1.2 Contributions

The current work aims to promote contract evolution safety and to ensure that changes in producers contracts do not require the consumers upgrade and redeployment.

The challenge of insuring the safety of contract evolutions will be handled by defining and adopting an API description language, as well as implementing a pre-flight safety check procedure that verifies whether service contracts are compatible before deployment. The compatibility mechanism is akin to type-checking procedures in compiled languages. The aforementioned mechanism preserves a distributed system's correct behavior by rejecting additions or modifications that would threaten it and by informing the programmer in advance of all the repercussions of the deployment.

To ensure that changes in producers contracts do not require the consumers upgrade and redeployment, contract evolution will be supported at runtime by a generated proxy component that dynamically adapts the data exchanged between services. The proxy

adapter is capable of evaluating all the outgoing and incoming requests and determine whether they should be left unchanged or how they should be transformed.

The addition, deletion, rename and migration of fields will be supported under a proposed evolution description language and system. Modifications on the signatures of WebAPIs, such as changing a path parameter to a query parameter, will also be supported. More complex changes, such as altering a field type or format, will be supported through the use of user-defined functions.

In summary, this work will provide:

- A compatibility verification on contract evolutions that determines whether or not messages may be exchanged without data loss;
- Definition of a description language for the specification of contract evolutions;
- Implementation of a robust type-safe adaptation protocol for service contracts;
- Implementation of a management system tasked with holding service contracts, tracking service dependencies, and providing a lightweight preflight safety check procedure for deployment/un-deployment operations;
- A benchmark platform to evaluate the solution.

1.3 Document Scructure

The document is organized into seven sections: Chapter 1-Introduciton; Chapter 2- Background; Chapter 3- Related work; Chapter 4- System design; Chapter 5- Implementation; Chapter 6- Evaluation and Chapter; 7- Conclusions and future work.

The first chapter, discusses the source of the problem as well as its challenges, and the contributions of the present study. The second chapter, offers a review of the main concepts, techniques, and applications underlying the problem, and the proposed approach. The ideas are expanded upon in greater detail in the third chapter, which evaluates the relevant work on topics such as Web APIs, schema representation languages, evolution approaches, and API Management Tools.

In the fourth chapter, it is discussed the design of the proposed system, as well as alternatives to the decisions made. The first section is devoted to the criteria that guided the system's design, while the remainder of the chapter is devoted to a discussion of the considered relevant aspects. The implementation of the proposed system is presented in the fifth chapter. It begins with an overview of the system's architecture before delving into the implementation of each module.

The results of the implementation are assessed in the sixth chapter. The benchmark platform is also presented in this chapter, as well as the methodology and experiments carried out. Finally, the last chapter is dedicated to the main conclusions and recommendations for future work.

2 Background

2.1 Microservices

Microservices [31, 53, 15] are an architectural style in which software is developed using self-contained components that communicate with one another via standardized network interfaces. These services segregate fine-grained business functionalities and can be independently tested, deployed, and scaled by automated mechanisms. There is typically no centralized management, each service may be written in a different programming language and employ a different data storage technology.

To understand the microservice architectural style it's useful to compare it to the monolithic architectural style (Figure 2.1).

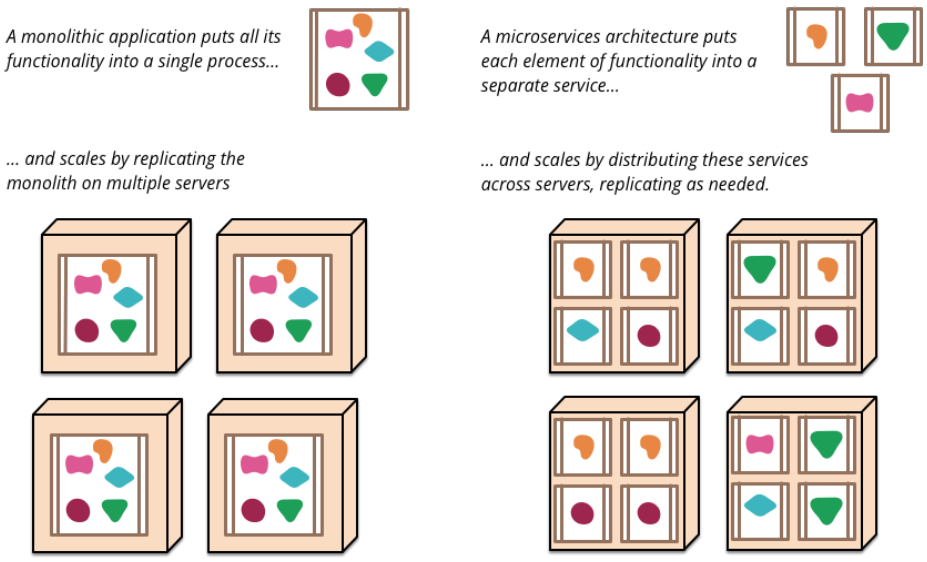

Figure 2.1: Monoliths and microservices [31]

Monolithic application's are often built using the Model-View-Controller (MVC) pattern [12], which is composed by three parts: The view, a client-side user interface composed of HTML pages and Javascript that runs in the user's browser; The model, a relational database management system; The controller, a server-side application that handles requests, retrieves and updates data from the database, executes domain logic, and populates the views that are sent to the browser.

The server-side application is a monolith, a single logical executable, in which all logic for handling requests runs as a single process, different domains of the application are divided into classes, functions, and namespaces by utilizing the basic features of a programming language. Large monolithic software becomes a barrier when the underlying application must scale while maintaining a high level of availability. Scaling the server-side application involves scaling all the application functionalities, rather than the functionalities that require greater resources. Any small change involves building and re-deploying the entire monolith.

In contrast to monolithic applications, several techniques are used in the microservice architecture to meet the requirements of system scalability and high availability, namely:

Componentization - In contrast to monoliths, which achieve componentization solely via the use of libraries and programming language capabilities, microservices achieve componentization primarily through the division of different business domains and functionalities into distinct executables that are exposed as services.

While this componentization approach helps to enforce component's encapsulation via more explicit component interfaces, its primary benefit is that services become independently deployable.

This feature helps to fulfill the requirements of system scalability and high availability by making different components scalable across multiple nodes and limiting cascading errors via the replication and isolation of components.

Decentralized Data Management - While monolithic applications typically use a single logical database for persistent data, microservices provide greater flexibility, fault tolerance, and scalability by allowing each service to manage its own database, which can be a different instance of the same database technology or a completely different database system.

Decentralizing data responsibility across services has implications for the implementation of cross-domain operations affecting multiple resources. The common approach for dealing with the problem of consistency when updating multiple resources in a database system is to use transactions. A transaction must preserve Atomicity, Consistency, Isolation, and Durability, commonly known as ACID properties. These properties are more difficult to fulfill in a distributed context than in a traditional database setting, hence developing and maintaining applications that use distributed transactions is notoriously complex. Microservice architectures

advocate for transaction-less service coordination with eventual consistency. The divergence between different services is addressed with the use of compensating operations instead of transactions.

Communication Channel - Martin Fowler, a well-known author in the context of microservices, promotes the use of ”smart endpoints” and ”dumb pipes” for microservices communication. Enterprise Service Buses (ESB) [33] were previously commonly used in service-oriented architecture (SOA) systems, and it was common to incorporate orchestration and transformation logic into the communication infrastructure, making the ”pipe smart”. This approach had several flaws, the tooling was complex and expensive, and it was difficult to solve problems in production environments.

The reverse approach has been adopted with microservices, where services own their domain-centric logic ”smart endpoints” and transport messages via ”dumb pipes”.

Communication Strategies - The majority of communication between microservices is done via request/response-based communication or event-driven messaging.

Request/response-based communication protocols are typically suited for synchronous settings, where the client contacts one receiver at time and needs the response before it can continue. In this approach there is a clear control of the flow, there is a service that plays the role of orchestrator and determines the sequence of operations to be performed in other services. The HTTP [20] and RPC [11] protocols are the most widely adopted protocols under this approach.

Event-driven messaging communication protocols are suited for asynchronous settings, where a client publishes a message to multiple receivers and can process the responses at a later time. In this approach there is no orchestrator, each service knows their role and what to do based on events that occurred. The main disadvantage of this strategy is that consistency is not guaranteed when multiple services consume events and one of them fails. Kafka [29] is the most widely adopted protocol under this approach.

Decomposition - When designing a software system, it’s important to decide how the system is divided into distinct components. Independent replacement and upgradeability are key features to consider when designing a component. The problems caused by poor division of concerns are alleviated in the context of microservices, because a system’s are typically divided into small components with a single responsibility, whereas in traditional architectures components have wider responsibilities.

To reduce the risk of structural refactors, techniques such as Domain-Driven Design (DDD) [17] are often used. DDD is a software design approach that decomposes

a complex system into multiple autonomous bounded contexts, where all the software structure (e.g methods, classes, variables) match the business domain. If the business domain is well-defined, the requirements of independent replacement and upgrade-ability can be met by reflecting business correlations in the software structure.

2.2 Software Provisioning

Two key processes are essential to provision distributed systems: a method for packaging and isolating services; a system capable of managing physical hardware to support services.

Services can be isolated using one of two methods: containers or virtual machines (VM). Containers offer the most efficient solution because, unlike virtual machines, they share the host system's kernel with other containers. VMs virtualize an entire machine down to the hardware layers, while containers only virtualize the software layers above the operating system level.

Docker [2] is the most popular container technology. It is built on top of the following kernel systems:

- Namespaces: Isolate the kernel resources (e.g. processes, filesystem, users, network stacks) used by each container.
- Cgroups: Isolate the hardware resources used by each container.
- Copy-On-Write File system: Allows several containers to share common data.

Additionally, Docker provides a mechanism for packaging code and its dependencies, referred to as container images. A container image is an executable package of software that contains all the components necessary to run an application: code, libraries, runtime and settings.

The management of services can be accomplished with the use container orchestration technologies. Container orchestration eliminates many of the manual processes involved in the management of distributed systems. Some popular options used for the lifecycle management of services are Docker Swarm [46] and Kubernetes [42].

Kubernetes (K8s) is an open-source container orchestration system that evolved from Google's Borg and Omega projects [10]. It manages the deployment, management, scaling, and networking of containers of distributed systems across a wide range of environments and cloud providers. This means ensuring that all containers used to execute various workloads are scheduled to run on physical or virtual machines, while adhering to the deployment environment's and cluster configuration's constraints. Any containers that are dead, unresponsive, or otherwise unhealthy are automatically replaced. Additionally, Kubernetes also provides a control plane to monitor all running containers. Kubernetes

accomplishes this through a well-defined, high-level architecture that encourages extensibility:

- Pod: Encapsulates an application's container (or multiple containers), storage resources, has a unique network IP address, and provides configuration options for the container(s).
- Service: Is an abstraction that defines a logical set of Pods as well as a policy for accessing them.
- Volume: Is a directory which is accessible to the Containers in a Pod.
- Namespace: Defines the scope of resource names, which must be distinct within the same namespace but not across namespaces.
- Deployment: Describes the desired state of the system.
- ReplicaSet: Ensures that a certain number of pod replicas are active at all times.
- DaemonSet: Ensures that all, or some Nodes are running a replica of a Pod.
- StatefulSet: Is used to manage stateful applications.

Kubernetes is a more sophisticated container management system than Docker Swarm. Docker Swarm is only compatible with Docker, whereas Kubernetes is compatible with other container services. In comparison to Swarm, Kubernetes is more difficult to deploy and manage, however it is more scalable.

2.3 DevOps

DevOps [16] can be viewed as a collection of practices designed to accelerate the software development cycle and shorten the time between developing new features and deploying them to production. The term DevOps is derived from the words development and IT operations, and it refers to the combination and automation of the traditional responsibilities of both departments.

Traditionally the tasks of development, testing, and deployment were divided among different engineering teams. Different teams had different goals, and in the typical scenario, communication between teams was slow and inefficient. By automating repetitive operations, DevOps minimizes the need for team coordination and speeds up manual processes that would have taken days or at least several hours. Building automated pipelines that are executed after the developer changes the codebase is a common DevOps practice.

Pipelines are typically in charge of packaging, testing, and deploying code to testing or production environments. If any phase of the automated pipeline fails, the developer receives immediate feedback on which task failed, and the pipeline's execution is halted. An example of a continuous deployment (CD) pipeline is shown in Figure 2.2.

The pipeline is divided into two sections the build and release pipelines. To reduce the iteration time of software development, the test steps of the build pipeline are conducted

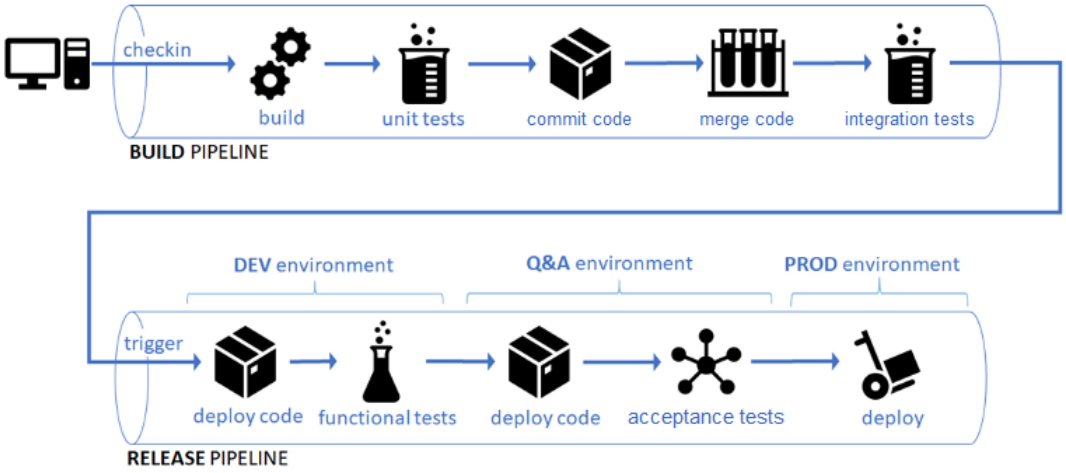

Figure 2.2: CD pipeline example

first on the developer computer and then repeated and verified on the external pipeline. The release pipeline entails the deployment of code into production. The code is initially deployed in staging environments where it is tested and monitored. After passing all staging phases, the code is considered as part of the system and is deployed into the production environment.

There are less complex pipeline architectures that automate fewer tasks. A continuous integration (CI) pipeline contains a subset of the tasks of CD pipeline. It only defines a strategy for continuously integrating and merging code from multiple sources. Continuous delivery pipeline is an extension of CI, that enables the automation of deployments to staging environments where the software is tested in sequential phases. A continuous deployment (CD) pipeline goes a step further by defining a strategy for automating the deployment and scaling of software in a production environment.

3

Related Work

3.1 Web API Description Languages

Web service development has risen dramatically in recent years, unlike statically linked library APIs, where developers may choose to stick with an earlier version that met their needs, with web APIs, the provider can discontinue a specific version and capability at any moment, changes are immediate and irreversible. This represents a heavy burden for developers of client modules as it causes an endless struggle to keep up with changes pushed by the web API providers. This load is exacerbated if the web API is inadequately documented.

Web API Description Languages (WADL) [21] are domain-specific languages used to describe web service contracts in a standardized structure; also sometimes more generally referred to as interface description languages (IDLs). These structured descriptions may be used to produce documentation for human programmers, that is easier to read than free-form documentation, because all the generated documentation adheres the formatting norms set by the used tool. Furthermore, description languages are often accurate enough to allow for the automatic generation of various software artifacts such as mock servers, client code generation in different programming languages, load test scripts, and so on.

The OpenAPI Specification (OAS) [38] is the most widely adopted Http Description Language; it defines a standard language-agnostic interface that allows both computers and humans to comprehend the capabilities of a service. The OpenAPI initiative is on a continuous effort to develop API tooling on top of their specification, some of the supported tools of version 3.0 range from:

- Auto Generators: Tools that parse code and turn it into an OpenAPI Specification document
- Mock Servers: Fake servers that take the description document as input, then route incoming HTTP requests to example responses or dynamically generates examples.
- Data Validators: Verify if API requests and responses are lining up with the API

description.

- Security: Tools that can look out for attack vectors by inspecting OpenAPI descriptions.
- Converters: Various tools to convert to and from OpenAPI and other API description formats.

OpenApi allows a detailed description of the syntactic aspects of the data transferred, however it ignores semantic aspects, such as the ability to relate different parts of the same data, to relate the input against the state of the service, and the output against the input. For instance, OpenAPI does not allow developers to indicate that in the creation of a new user, the nickname must be shorter than the full name.

Vasconcelos et al. [50] proposes an alternative WADL that supports the expression of semantic aspects in data, the HeadRest specification language. Two ideas are embodied in the proposed language:

Refinement Types [18] can be used to express the properties of data exchanged. A refinement type x can be defined as

$$x : T \rightarrow e$$

Where x is an object of the primitive type T, and e is a predicate which returns true or false depending on whether the value conforms to the boolean expression (e.g. x > 10).

Pre- and post-conditions can be used to express relationships between data sent in requests, and the data returned in responses. These conditions can be expressed as a collection of Hoare triple assertions

$$\{\phi\}(a : t)\{\psi\}$$

Where a is HTTP operation type (GET, POST, PUT, or DELETE), t is an URI template (e.g. /users/), and Φ and Ψ are boolean expressions. Formula Φ, called the pre-condition, addresses the state in which the action is performed as well as the data transmitted in the request, whereas Ψ, the post-condition, addresses the state resulting from the execution of the operation together with the values transmitted in the response.

Web API description languages provide a foundation for defining compatibility relations between service contracts in the evolution description language, presented further ahead in Section 4.2.4.

3.2 Schema Representation Languages

Schema representation languages (SL) [34] are used to specify and document the structure and semantics of data. A schema language is typically backed by a serialization protocol,

which is used when data must be transported across a network or stored with durability. Some serialization protocols do not have a well-defined schema language. This is the case in the serialization libraries provided by programming languages, which serialize data in a native binary format that is not human-readable. In the setting of microservices, serialization libraries supplied by programming languages are strongly discouraged, because each service may be written in a different programming language. Data consumers will be unable to comprehend producers if they use mismatching data representations.

Cross-language serialization libraries, such as JSON [5], can solve this problem. However, cross-language formats typically don't have the same level of expressiveness as the programming languages that employ's them, making data consumption more difficult due to poor type-safety guarantees. For example, whereas Java offers distinct data types that differentiate integers from floating point values, JSON regards them as the same data type. In the above case, if no further documentation is supplied, consumers will be constrained to adopt the more embracing data type "float"in order to avoid runtime errors.

JSON also doesn't have a well-defined schema language. In JSON schemas are typically inferred from the structure of objects in a programming language that represent the received and sent messages. Without an explicit schema definition and versioning strategy, fields can be unilaterally added or withdrawn at any moment without the consumers' knowledge.

There are cross-language serialization protocols that require the data schemas to be explicitly defined under schema language. Avro [51], Thrift [13], and Protocol Buffers [44] are among them. The benefit of having an explicit and versioned schema is that it makes the system more robust to schema evolutions, and enables messages to be sent with minimal size. The encoding of a message only requires the inclusion of the message data, and the schema version. A service can be made aware of a message complete schema prior to receiving it via a schema registry. The inclusion of information about each field's data type and name is redundant, and it is only useful if the message is meant to be human-readable in an environment where the message schema is inaccessible.

Schema representation languages are a part of the specification of service contracts presented further ahead in Section 4.2.2.

3.3 Schema Registry

A schema registry, as the name implies, is a repository for schemas. It stores a versioned history of schemas and provides an interface for retrieving, registering, as well as checking the compliance of schemas. It is essentially a CRUD (Create, Read, Update, Delete) application with persistent storage for schema definitions, where each schema is given a unique ID.

One framework that employs schema registries is Avro [51], a data serialization framework developed within Apache's Hadoop project. Avro requires a schema registry because

the encoding of data records does not include data about the record structure such as field names or data types. Only the data values of a record are appended in the serialized byte sequence, as seen in Figure 3.1. The deserializer knows which bytes belong to which field by comparing both the consumer and producer schemas. This allows for a more compact serialization method, but it requires data consumers to understand the structure of the data written by producers.

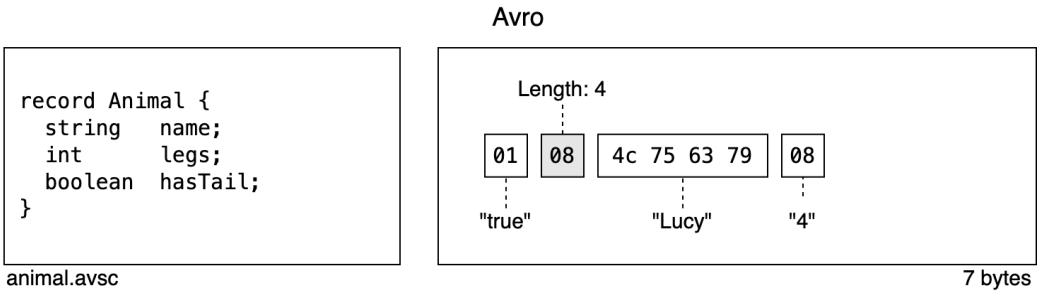

Figure 3.1: A Avro schema and its associated serialized record

Another system that makes use of schema registries, is the Java RMI [14], a Java API that supports remote procedure calls (RPC) with distributed garbage-collection. The Java RMI makes use of a schema registry to enable clients to obtain a references (stubs) to a remote objects. In the RMI the registry is only used for locating the first object that a client needs, each reply from the registry provides support for finding other objects.

Registries can be used to store not only service schemas but also service-to-service dependencies, and other meta-information about services, the benefits and drawbacks of their use will be assessed in Section 4.2.10.

3.4 Service Evolution Approaches

Controlling and supporting the evolution of services is a significant technological research endeavor. Meng and He [32] propose five standards for evaluating existing methods for evolving Web services. These standards and some methods are also applicable in a microservice setting:

- **Granularity of evolution** - Granularity refers to the unit on which the evolution is based. Coarse-grained evolution strategies support the evolution of general interface properties, whereas fine-grained evolution strategies support the evolution individual basic properties, such as the type of parameter in a method.
- **Terminal of evolution** - The term refers to whether the end result of a changed service affects the producer, the consumer, or both.
- **Type of evolution** - Evolutionary methods can be divided into two broad categories: semantic and architectural. Architectural methods typically support changes in the composition of components, whereas semantic methods are primarily based on the semantic extension of contracts.

- **Scalability** - This standard assesses an evolutionary method's applicability and generality in different contexts.
- **Maintainability** - This standard assesses the difficulty of maintaining system's that employ the relevant evolutionary method.

Below, it is presented a summary of the common approaches used to support the evolution of microservices, along with their strengths and shortcomings in regard to the standards outlined above.

3.4.1 Hand-off Period

The conventional approach for evolving microservices is to keep deployments of service versions that are still being consumed available with a hand-off period, and to route requests to the service deployment that has the appropriate version. This approach has been proven to be pragmatic, but exceedingly expensive, because it necessitates high maintainability and partitions available resources among continually shifting subsets of consumers.

3.4.2 Deprecated Endpoints

Alternatively it's possible to support backwards compatibility without distinct deployments, by including in the new service contract the endpoints of the prior service versions that are still in use. When a breaking change occurs in a contract evolution, we preserve the old endpoint and implement the new endpoint separately. The old endpoint is not modified and is still accessible via a similar but distinct path.

Contract changes that only affect HTTP parameters can easily be accommodated under this approach, however, contract changes that affect request body schemas are problematic because it would be necessary to write distinct serialization and de-serialization rules for each version of the schema. This approach can add significant amount of effort to the development team because each breaking change requires all the serialization rules to be rebuilt, in order to support the format of the new revised schemas. Schemas with multiple serialization rules also tend to become unreadable. Furthermore, this approach doesn't offer any safety guarantees, a developer might easily make the error of deleting a serialization rule that is still in use.

3.4.3 Schema Resolution Rules

The previous approach can be complemented with the use of sophisticated serialization protocols such as Avro, Protocol Buffers or Thrift. These protocols support the evolution of schemas via resolution rules and declarative semantics, where the old version of the software can deal with the new version of the service's syntax. However, the schema evolution support in these protocols is limited [28]. The common limitations of the aforementioned protocols are presented bellow:

- To ensure backwards compatibility, newly added fields must be optional.
- The only fields that can be removed to maintain forward compatibility are optional fields.
- It's not possible to change a field type or its format freely, field types can only be promoted to more embracing types (e.g int→float).
- Renaming fields is supported through the use of aliases that cannot be re-used by other fields in the same scope.

In the context of microservices, backward compatibility is only required in request schemas, while forward compatibility is only required in response schemas. In other words the current producer implementation must be able to interpret request messages from outdated consumers, and outdated consumers must be able to interpret response messages from the current producer implementation. Schemas that are used both in request and response messages most support forward and backward compatibility.

Optional fields are imposed in the above cases due to missing information in messages. If a field must be required, its value can be supplied through static default values. All fields that are optional exclusively because of limitations in resolution rules should be treated as required when a consumer and producer agree on the same schema version. Since there is no absence of information, the default value should not be used in this case. To simply put it, the system should enforce required fields when the consumer and producer agree on the same schema, and only use static default values when the consumer and producer disagree.

The aforementioned protocols do not differentiate between the two cases; instead, their resolution rules solely operate over the syntax of schemas and are agnostic to the version of the consumer and producer's schema. In order to enforce required fields in the former case, validation logic would need to be written repeatedly by programmers (in the same layer as the business logic). This validation should be in a layer above because, the scale of the validation logic is proportional to the complexity of messages being validated; for messages with more properties, particularly those with nested objects, the validation footprint can rise dramatically in terms of both line-count and logical complexity.

There are serialization protocols with built-in support for the evolution of contracts based on Event-driven and RPC communication methods. However, if the HTTP communication protocol is adopted, serialization protocols will be unable to manage changes in the signature of endpoints (e.g. changing the method of HTTP endpoint's from GET to POST), only changes in schemas of a message body will be managed in this case. To support the evolution of endpoint signatures, this approach will need to be supplemented the deprecation of endpoints discussed above. To decrease the complexity of contract evolutions, it is beneficial to handle both the evolution of schemas, and the evolution of endpoint signatures in a single integrated approach.

3.4.4 Chain of adapters

The chain of adapters is a design strategy for supporting web services in the face of independently developed and unsupervised external consumers while preserving strict backwards compatibility. This approach decomposes long update/rollback transactions into smaller, independent transactions. This technique is discussed in depth by Kaminski, Müller, and Litoiu [27]. The key concepts of the approach are presented bellow.

The service's backwards compatibility is ensured by the composition of adapters in a chain, as seen in Figure 3.2. When a service API is changed, the prior implementation is decommissioned, however the affected endpoints of prior API are made available in a distinct path's. The previous implementations are replaced by adapters that translate the requests data and format as necessary, and that redirect requests to the current implementation.

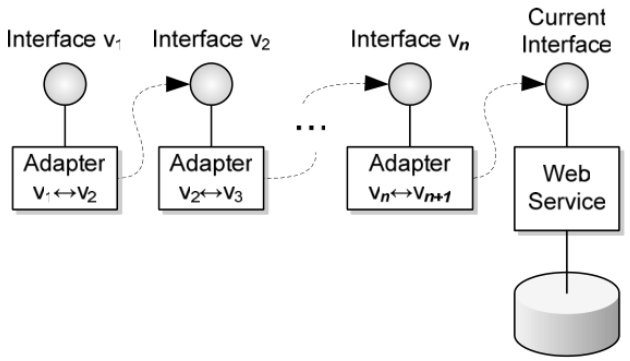

Figure 3.2: Chain of adapters structure after n versions have been published [27]

Updating a service's API in this manner requires the developer to do two additional tasks:

- Duplicate all modified endpoints in service contract into a different path;
- Manually implement an adapter that translates the endpoints in version $v_i \rightarrow v_{i+1}$.

The adapters code is included in the web service implementation that supports the revised service endpoints. The adapters are not deployed separately, only the most up to date implementation of a service is deployed.

For instance, when an outdated consumer in version v_i communicates with a producer in version v_{i+2}, the endpoints that provide the service in version v_i use the adapters $v_i \rightarrow v_{i+1}$, and $v_{i+1} \rightarrow v_{i+2}$ to translate the operation before forwarding it to the endpoint in the current version.

This approach offers a high degree of granularity because all types of contract evolutions can be supported under it. The adapter's implementation are tailored to solve specific contract evolutions, allowing the developer to select the most appropriate evolution strategy and method. However, it is difficult to maintain a system using this approach due to burden imposed by the additional tasks described above.

3.4.5 Lazy Proxy Adapter

In this strategy the knowledge of a microservice's contract is used to automatically create a lightweight proxy capable of dynamically adapting the messages exchanged between services to match them with the static service code. The proxy intercepts messages from consumers and adapts the message to the new specification before they reach the producer. Consumer services are initially deployed without the adapter proxy. Proxies are installed and updated on demand using a lazy instantiation method, that builds and deploys the adapter proxy when the first communication between corresponding services occurs. Seco et al. and Santos et al. investigate this approach [43, 41].

The proxy creation mechanism makes use of the available meta-information to automatically update the proxy components as needed. In essence, every communication contains information about the agreed-upon version, the handshake protocol uses this information to determine if a proxy update is required or not. The lazy instantiation method improves the maintainability of the system by simplifying deployment operations.

This approach improves the system maintainability, by sacrificing some granularity on the supported evolutions due to the automatic generation of the proxy components. However, with this approach external consumers cannot be served, since changes to a service contract affect its consumers. The proxy adapter components cannot be easily installed in unsupervised consumers.

3.5 Contract Evolution in MicroService Architectures

The evolution of a microservice contracts while ensuring their soundness and avoiding heavy adaptation processes due to service redeploying is demonstrated to be possible by Seco et al. [43]. The proposed approach to microservice contract evolution [43] makes use of a global deployment manager component that is responsible for managing module references, as well, as dynamically generated proxies that are capable of adapting messages exchanged between modules when contracts mismatch. The following example demonstrates the mechanism.

Consider the marketing system shown in Figure 3.3 that comprises three distinct modules: Catalog, Marketing, and BackOffice. These three simple modules combined, form a triangle of dependencies, complex enough to demonstrate the proposed mechanism. Please note that the Catalog module is a producer, whereas the Marketing module is a producer and a consumer.

Each module has a unique name, a collection of type and function definitions, and a set of type and function references. If a producer module exposes definitions, they can be used by other consumer modules via an explicit reference. Each programming element is accompanied by an immutable and unique key; for instance – the key of type "Product" is k1. These keys enable elements to be identified even after they have been renamed or

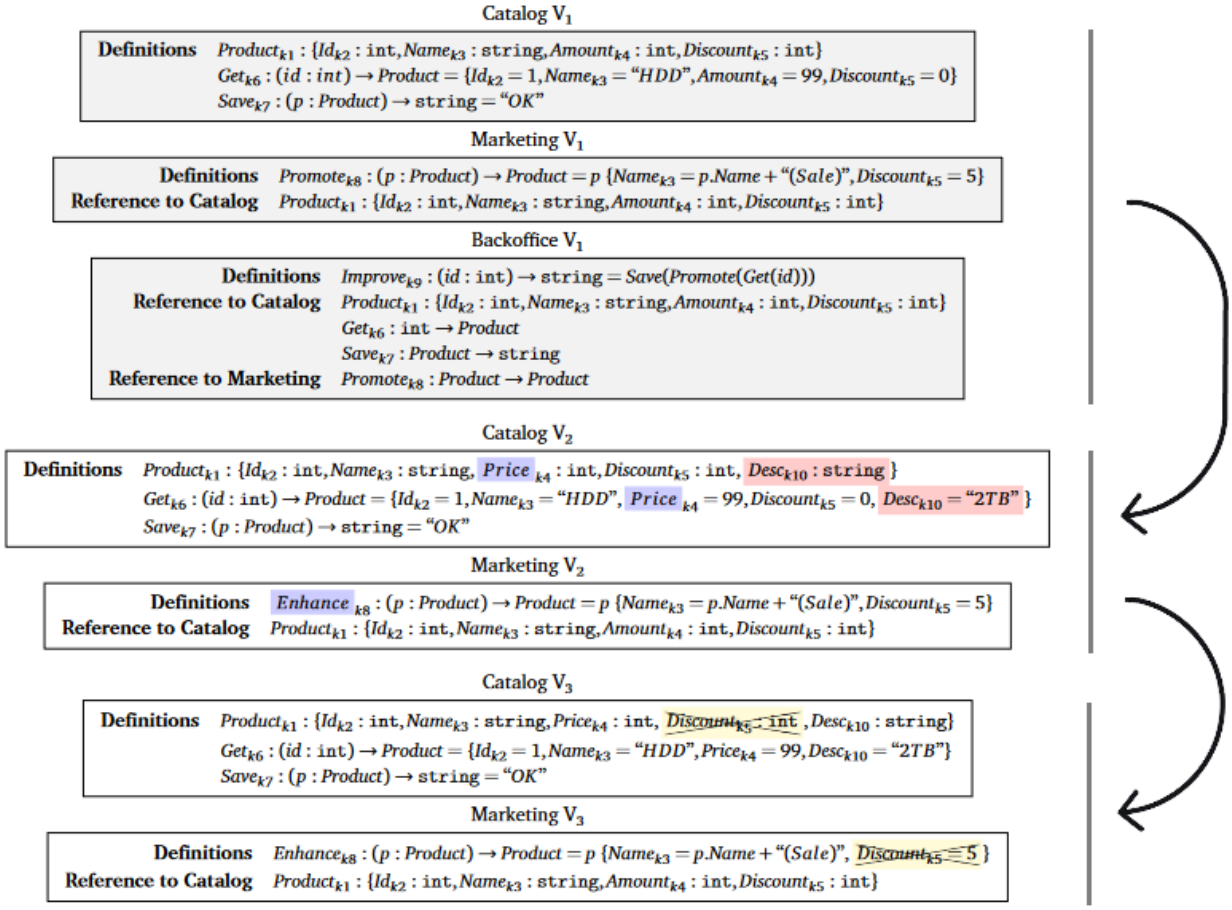

Figure 3.3: Evolution of modules Catalog, Marketing, Backoffice [43]

reinstated.

Initially, the system is composed by a collection of services, represented in grey in the first instance of the example. The reference definitions indicate their interactions.

In a second phase, it is developed version 2 of the Catalog and Marketing modules independently. In the Catalog module, the element ”Amount” is renamed to ”Price”, and the attribute ”Desc” is added, while in the Marketing module, the attribute ”Promote” is renamed to ”Enhance”. The module BackOffice has not been modified at this point and will become out of sync with the other modules. In the figure, the newly added and modified elements are highlighted in red and blue, respectively.

In a third phase, the Catalog module is upgraded to version 3, where the attribute ”Discount” of the element ”Product” is removed. The elements that have been removed are highlighted in yellow. This version cannot be deployed in conjunction with version 2 of Marketing, as the attribute is used by the function ”Enhance”.

Both modules can be deployed together if the attribute ”Discount” is removed in the definition of ”Enhance”, resulting in version 3 of Marketing module. Additionally, it

can be deployed version 3 of Marketing first and then version 3 of Catalog, but not vice versa. Furthermore, it is permitted to retain the attribute "Discount"in the specification of the type "Product"in the reference to catalog. Correctness is guaranteed as long as the property is not utilized.

As illustrated, the system is capable of overcoming differences in contract definitions with the following evolutions:

- **Adding new attributes to a type** - As a result of the addition of the "Desc"attribute to Catalog, the data returned by this module will include values for this attribute. Other modules will keep this data (as unknown attributes) to ensure that no data is lost;

- **Renaming functions** - The ”Promote” function in Marketing module has been renamed to ”Enhance”, which will affect the endpoints exposed by the service. The proxy is dynamically built at runtime to use the actual endpoint name while issuing calls, eliminating the need to update and redeploy the service Backoffice;

- **Eliminating unused attributes and functions** - Unused elements do not require explicit adaptation.

Traditional approaches, are insufficiently robust to handle these types of changes, effectively rendering them breaking changes. For instance, renaming functions results in the modification of remote URIs, and changing the name of an attribute may result in data loss. By contrast, with this approach, such changes are permitted without compromising module compatibility. This adaptive approach enables gradual module deployment of the referenced modifications without halting the entire system, avoiding data loss and misinterpretations.

3.6 Service Integration Adapters

There has been substantial investigation on the adaptability of services in the context of service-oriented architectures [6]. Although SOA and Microservices are similar in that they both use a separation approach based on services, these two designs differ in several important ways, most notably in that microservices are loosely coupled whereas SOA services are tightly tied due to common data storage between services. As a result, adaptability in SOA is focused on service integration and replace-ability rather than evolution.

In the setting of SOA, adapters are used to wrap the heterogeneous services that communicate with distinct protocols and data formats, in order to make them homogeneous and integrated them more easily. These adapters presume that data adaptation has already been established by developers at an upper layer, their primary concern is

integrating services with distinct communication protocols and mismatches between operations that have the same functionality but different signatures (e.g parameters name, order, type).

Benatallah et al. [6] present a taxonomy of all conceivable mismatches as well as remedies to each type in the context of SOA. Some of the defined mismatches remain important in the context of microservice evolution:

- **Message Split Mismatch:** This type occurs when the protocol P requires a single message to achieve certain functionality, while in protocol PR the same behavior is achieved by receiving several messages.
- **Message Merge Mismatch:** This type occurs when protocol P needs to receive several messages for achieving certain functionality while protocol PR requires one message to achieve the same functionality.
- **Differences at the Operation Level:** This type occurs when the operation O of service S imposes constraints on input parameters, which are less restrictive than those of operation O input parameters in service SR (e.g., differences in value ranges).

The ”Message Split Mismatch” can be handled through an adapter, however doing so would require the adapter to use distributed transactions in order to handle independent failures in each contacted service.

The ”Message Merge Mismatch” can also be solved with an adapter, but the adapter must be stateful because it must store request messages before they can be merged and sent through the new endpoint.

The first two types of mismatches can be handled in the context of microservice evolution without the aforementioned complications, by essentially keeping the old endpoints operational but marked as deprecated until there are no more consumers using them, and by providing the new endpoints in parallel.

The latter form of mismatch ”Differences at the Operation Level” has an impact on microservice evolution if service contracts allow refined types. In other words, when constraints in input parameter are validated in a layer above the application layer. If validation is conducted in a lower layer, the mismatch can be easily resolved by modifying the application’s implementation while keeping the service contract untouched.

3.7 Deployment Strategies

Deployment strategies [22] are techniques that aim to provide means to upgrade a service, while discontinuing the previous implementation without incurring downtime.

These strategies will be beneficial in the context of the proposed solution since all service updates will evolve the immediate termination of the prior implementation, as it is no longer required to keep several versions available to offer backwards compatibility in service contracts.

A blue-green deployment approach is the most often used deployment strategy. The new version ”blue” is made available for testing and review, while users continue to use the stable version ”green”. Users are migrated to the new version after it has passed all compliance tests.

A common alternative strategy is to have two A/B versions active at the same time, with some users using one and others using the other. This strategy can be used to test and gather feedback on modifications to user interfaces. It can also be used to troubleshoot problems that affect only a subset of users, in a production setting.

A canary deployment can be used to evaluate a new version, if a problem is discovered, the deployment is quickly reverted. This technique can be performed in either of the above approaches.

A rolling deployment is an alternative strategy that gradually replaces instances of a service previous version with instances of the new version. Before scaling down the old components, a rolling deployment normally waits for additional pods to become available through a readiness check. If a significant problem arises, the rolling deployment is halted. This strategy ensures the highest availability while using the fewest resources. In comparison to other strategies, the additional resources allocated during the deployment process are minimal.

3.8 API Management Tools

API management is the process of overseeing functions like API creation, description, testing, analysis, publication, securing, and monitoring. All of these functions can only be met with the assistance of tools.

Microsoft Azure's API Management Platform (AAMP) [3] is an example of a system that provides tool's with the functionalities mentioned above. For example, in this platform, programmers are allowed to specify transformations in policy statements, that are performed on all responses from a microservice. The main use case of these transformations in AAMP is the integration of legacy backends by modernizing their APIs and making them accessible from cloud services, without the risk of migration. This tool is one of the inspirations for the proposed approach, which aims to allow similar transformations to be performed in both request and response messages.

In AAMP all requests from client applications are routed through the API gateway, which then forwards them to the appropriate backend services. The API gateway, acts as a facade to the backend services, allowing API providers to abstract API implementations and evolve backend architecture without impacting API consumers. The AAMP's other responsibilities and functionalities include:

- Accepting API calls and routing them to configured backends;
- Verifying API keys, JWT tokens, certificates, and other credentials;
- Enforcing usage quotas and rate limits;

- Caching responses to improve response latency and minimize the load on backend services;
- Emitting traces for monitoring, reporting, and troubleshooting;
- Collecting API usage metrics.

Most of the aforementioned functionalities are also supported in the Kubernetes ecosystem and its extensions. The proposed approach can be viewed as API management tool that solely focus on the evolution of API's. We aim to make it compatible with existing API management tools in the Kubernetes ecosystem that cover the other functionalities.

4

System Design

4.1 Discussion

Recall that the problem at hand is how to make the process of updating contracts in microservice-based systems more robust, in the sense that it should be harder to deploy a service that could potentially break the system soundness. There are many approaches to solve this problem, each with its own set of compromises. The requirements that guided the design of the system, are outlined below by their priority:

- No downtime when upgrading service contracts;
- Automatic validation of the safety of deployment operations;
- Supporting all types of contract changes;
- Integration with existing tools and workflows;
- Minimize conception and maintenance effort;
- Consumer and producer services ignorant to the presence of proxy adapters;
- Transactionless deployment of upgrades to service contracts;
- No overhead when services communicate using the same contract version.

Four criteria were considered when evaluating each approach: **Flexibility** the applicability of the approach under diverse scenarios; **Effectiveness** the efficiency of an approach at solving a problem; **Utility** the effort required to adopt and maintain the approach; **Performance** the overhead associated with approach.

In the prototype's design, it was prioritized utility above performance because: in the common case, it is expected for most service's to communicate via up-to-date contracts, messages will only need adaptation during a transitory period; the performance overhead remains roughly static under diverse contract changes, as it is largely attributable to communication costs. It was also prioritized flexibility and effectiveness above other criteria because if an approach isn't applicable or effective under a scenario, then it will be necessary to adopt hybrid approaches, which will have a detrimental effect on the utility of the entire solution.

4.2 Design Aspects

4.2.1 Communication Protocol

The communication protocol defines the syntax, semantics and synchronization of communication between microservices. The choice in the communication protocol is bounded by the adopted communication strategy between microservices, orchestration or choreography. Orchestration relies on request-response protocols while choreography relies on event-driven protocols.

Approach: It was chosen to design the system around the orchestration pattern and the HTTP protocol, because most applications require a web presence and microservice systems that rely on choreography still require the HTTP request-response protocol to support external consumers.

Alternatives: If choreography was adopted, the approach would need to support an additional communication protocol with different a contract structure and syntax. It is complex to implement a generic approach that can handle multiple types of communication protocol's. Other event-driven and request-response protocols, such as RPC, have contracts with a simplified syntax and structure. There are already a number of tools that support the evolution of contracts under these protocols, albeit with some limitations.

4.2.2 Contract Specification

It is required that each microservice's interface be described in a high-level language which abstracts away implementation details, because it is common for microservices to be implemented under different frameworks and programming languages. Additionally the contract specification of a service also includes information about its dependencies, and the necessary computational resources to support its operation.

Approach: The contract is composed by two distinct manifests. In the first manifest the capabilities of a service are specified using the conventional approach of a Web API description language. In the second manifest the service dependencies and hardware requirements are documented with a distinct description language. The contract is divided into separate manifests, because external consumers are only interested on the documentation of the capabilities of a service, and because it may be problematic to disclose information about hardware requirements.

To reduce the documentation effort only the dependencies between services are document, the dependencies between service procedures are not specified (eg. Service A in version 1.0 depends on service B in version 2.0). One drawback of not documenting dependencies at the procedure level is that it is not possible to detect procedures that are unused by other services. The installation of adapters is unnecessary in cases where

only unused procedures are updated in a contract. This case is expected to be rare, and having unused procedures in a microservice is undesirable due to API bloat. The specification of inter-service dependencies is sufficient to ensure the safety of deployment operations. The documentation of inter-procedure dependencies can be useful in API management tools, these dependencies can be discovered indirectly via the alternative approach presented below.

Alternatives: Dependencies can be determined automatically by inspecting request logs between services. The advantage of this approach is the reduced documentation effort, but it comes at a cost because it is no longer possible to validate the safety of service deployments, since the dependencies of a service are unknown when it is deployed. With this approach it is only possible to validate the safety of the removal of services that have been running over a long enough period of time.

4.2.3 Supported Contract Evolutions

An HTTP contract is composed by procedures, each of which contains one request message and up to N response messages.

Response messages are composed by a status code and parameters. The same status code cannot be shared across multiple response messages. A response message's parameters can be placed in one of three places: headers, cookies, or body.

Request messages, on the other hand, are composed by an endpoint and parameters. The endpoint is comprised by an HTTP method, a domain, and a sequence of path segments. The parameters of a request message can be placed in two additional locations: path or query. Only body parameters can have object or collection types. Other parameter groups are limited to using primitive types (string, number, boolean).

All the contract elements mentioned above can be subject to change's as the contract evolves. Multiple types of changes can also occur simultaneously in the same contract evolution.

A taxonomy of all conceivable HTTP contract evolutions [52] is presented bellow:

Functional evolutions:

- Adding/removing a parameter.
- Adding/removing a response message from a procedure.
- Adding/removing a procedure.

Semantic evolutions:

- Changing the type of parameter.
  (eg. turning a integer field into a string)
- Changing the HTTP method of a procedure.
- Changing the status code of a response.

Syntactic evolutions:

- Renaming a parameter.
- Migrating a parameter into a different location.
  (eg. converting a path parameter into a field of the request's body schema)
- Changing the format of a parameter.
  (eg. modifying the format of a date)
- Splitting a parameter into N different parameter.
  (eg. dividing a name field into first and last name)
- Merging parameters into a single parameter.
- Changing the domain or resource URL

Architectural evolutions:

- Splitting a procedure into N different procedures.
- Merging N procedures into a single procedure.

Approach: Of all the stated contract evolutions, the only changes that aren't directly supported are architectural evolutions.

Supporting the division of a procedure into multiple procedures is complex because the adapter would have to use distributed transactions to handle the independent failures of each new procedure. Combining multiple procedures into a single procedure is also costly because the adapter must be stateful and session based to support this change. The requests of the affected procedures would need to be stored in the adapter before they could be merged and sent through the new endpoint. Furthermore, these two types of evolution are cumbersome to be documented in a description language, because the language would need to relate procedures in an $N \rightarrow 1$ and $1 \rightarrow N$ relationships.

Alternatively, architectural evolutions can be converted into equivalent functional evolution's that keep the deprecated procedures in the contract alongside the new procedures.

4.2.4 Specification of Contract Evolutions

It is not always possible to implicitly deduce the evolutions in a contract when comparing a new version to a prior version (for example, renaming field A to B is indistinguishable, from inserting a new field B and removing a field A of the same type). The developer must declare explicitly which evolution occurred.

Approach: Contract evolutions are specified in a manifest file, complete with its own syntax and rules, which outlines how to adapt calls between two versions of a contract. In this file all mappings between elements in each version are explicitly defined. A mapping for one element can fall into one of three types:

- **Default value**: The developer provides a default value for the element. This mapping is only valid if the default value has the same type and format as the element.
- **Link**: The developer indicates that element A on the previous contract is equivalent to element B in the new contract. This mapping is only valid if the two elements have the same type and format.
- **Function**: The developer applies a function over X_n elements of previous contract and indicates that the result of the function is equivalent to element Y in the new contract. This mapping is only valid if the function's output type and format are equal to element Y and if the function's arguments match the type and format of the provided X_n elements.

Depending on the API evolution a different mapping type is used: renamed fields in contracts are supported with the usage of links; the addition of new fields in contracts is supported with the use of default values; complex contract changes, such as changing the format of a date are supported with the use of functions.

Additionally, procedures of a previous contract that are unused can be declared as obsolete in cases where there functionality is no longer needed. Procedures that are obsolete don't need to be adapted by the proxy component.

Alternatives: The evolution specification is defined alongside the contract specification in the same manifest file. Most web API specification languages already support default values and name aliases. With these two features it would be straightforward to document the following contract evolutions: renamed/addition/removal of fields. This alternative may seem appealing for simpler contracts, but it becomes impractical as the number of revisions and consumers in prior versions rises. This approach clutters service contract's with information that is irrelevant to consumers, and makes the specification of complex contract evolutions more cumbersome.

4.2.5 Compatibility Verification

Intuitively, compatibility verification determines whether all the edited elements in a producer contract are compatible with the ones effectively used by the consumer services, and whether the new contract requirements are met by the system's existing resources.

Approach: The verification procedure examines if:

- There are enough computational resources to accommodate the service;
- All the service dependencies are available and reachable in the system;
- A new contract is compatible with the prior contract versions that are still being consumed.

Edited elements are considered compatible if a resolution rule is present that employs earlier elements to satisfy the edited elements' requirements. This approach makes the assumption that all procedures from previous contract versions are being used. If a new contract is missing a procedure supported in an earlier version, the developers must verify if the procedure is not being used and mark it as obsolete in the evolution manifest before the contract can be deemed safe.

Alternatives: To assess the safety of a deployment operation, the verification process alternatively could examine only the most recent producer contract version and all consumer references. This approach has the benefit of removing human error from the verification of unused service procedures, however, it necessitates the documentation of all references between consumers and service procedures.

4.2.6 Adapter Location

The adaptation of messages is supported at runtime by a generated proxy component that dynamically adapts the data exchanged between services. The adapter can be deployed and intercept messages in different points through the exchange.

Approach: The adapters are installed as side-car container on the same node as the producer service container. Consumers direct their requests based on the expected version of the service using a routing rule. If the expected version matches the current service version, the request is answered directly by the service; otherwise, the request is redirected through the adapter. Since the adapter and the service are in the same node, the communication cost is expected to be comparable to the cost of inter-process message passing. The main benefit of this approach is that the adapter can be deployed alongside the service, which circumvents distributed transactions in deployments, because container orchestrations tools such as Kubernetes allow the deployment of closely related containers as atomic units known as "pods".

Alternatives: An option, is to have the generated adapter code embedded in the service implementation. This approach is expected to have higher performance than the previous approach since the adapter would use function calls instead of inter-process calls, and because message serialization and de-serialization would be unnecessary. The downside of this approach is that it would entail the modification of the service's code, which

is problematic because each service can be implemented in a different framework and programming language.

In another option, the adapter can be installed on the nodes of all the consumers of a producer service. This approach is problematic because a service upgrade would entail the re-deployment of all consumers. If one of the consumer re-deployments fails, the entire service upgrade would have to be cancelled; this is only achievable with the use of distributed transactions, which are inherently complex. The re-deployment of consumers pods due to the installation of adapters can be avoided with the use of mobile code that is provided by the producer in a dedicated endpoint (for example, this could be accomplished in Java with the use of network class loaders). This approach is ineffective because it provides no benefits and requires more resources than the alternatives, since instead of one adapter, N adapters would be needed, one in each consumer.

4.2.7 Adapter Coverage

The adapter can have a generic implementation that is able to adapt messages from any service, or it can have a direct implementation that can only manage the adaptation of messages for a single service.

Approach: The adapter implementation is derived from a template implementation, that is completed with code generated from evolution specification manifest file. The advantage of this approach is that the implementation will be more performant than generic approaches.

Alternatives: The adapter has a generic implementation that is initialized with the data parsed from evolution specification files. The correct adaptation procedure for a message is determined by inspecting specialized headers in the message that indicate the message type, service and version. The advantage of this approach is that a single adapter can convert messages from any service as long as the evolution manifests are supplied to the adapter beforehand. This approach will consume fewer computational resources, in scenarios where the number of messages that require adaptation is low and uniformly distributed across all services, because a single adapter can handle the load of all requests, whereas the alternative requires one adapter per service. The disadvantage of this approach is that it requires modifications on the consumer's implementation to include message headers indicating the expected version of the producer service.

4.2.8 Message Routing

Messages will need to reach either the application or the adapter, depending on the consumer expected contract, and the current producer contract. Only messages that require adaptation should be forwarded through the adapter to reduce the adapter's load and resource consumption.

Approach: The application and adapter servers of a microservice are accessible internally inside a cluster via distinct Kubernetes (K8s) services. A K8s service, is an abstraction that defines a policy for accessing a logical set of containers under a single network address. For each version of a service contract a distinct K8s service is created with the corresponding service name and contract version. The address of each K8s service is discoverable via an internal DNS server.

When a microservice is upgraded, it's K8s services are refreshed with the new addresses of the application and adapter containers. The K8s service that previously pointed to the application containers, will now point to the adapter containers, and a new K8s service is created to support the new contract. All consumers of the previous contract will begin forwarding their requests to the adapter container's rather than the application containers.

Consumers that only use unaffected procedures in the contract evolution don't need to direct their calls through the adapter. If optimal performance and resource consumption is desired, such consumer services will have to be updated and redeployed with the new producer DNS name, in order for their calls to be redirected to the application servers.

The redeployment of consumers in the former case can be avoided, if consumers participate in the contract evolution process. The adapter server includes a header in responses that contains the address of the application server, and that indicates if a request required adaptation. If the request didn't require modifications, consumer services begin to forward the request to the application server instead of the adapter. With this approach consumer services are no longer unaware of the existence of proxy adapters, and will need to use different addresses when calling different procedures of the same producer.

Alternatives: A reverse proxy like Nginx [39] could be used to route only the messages that require adaptation through the adapter. With this approach all requests are forwarded through reverse proxy and redirected based on custom routing rules. The routing rules are installed and updated when contract implementations are deployed. This approach is counter-productive since it necessitates an additional reverse proxy server, which will use roughly the same computational resources and add the same latency as the adapter server.

4.2.9 Adapter Management

The management of adapters should be automated similarly to the management of services. In this section, it is discussed how to conciliate the operations related to the management of adapters with the more common operations of service deployment and upgrade.

Approach: During normal operation the adapters don't require any type of management other than monitoring, adapters components only need to be provisioned during the

upgrade of service contracts. A contract upgrade is supported by the re-deployment of the impacted microservice, which is often managed via a continuous deployment pipeline.

The operations of contract compatibility verification, adapter generation, deployment, and removal are introduced as additional stages in typical continuous deployment pipelines.

4.2.10 API Management Tools

In distributed systems there is a lack of tools comparable to those used in centralized systems, for visualizing the impact of a change and effort required to implement it. In this section, it is discussed how such tools could be implemented.

Approach: Since all service contracts declare their dependencies on other services, the impact of change can be evaluated at a macroscopic level by iterating thorough all contracts of the active services and verifying whether the changed service is being consumed. Service contracts are stored in a version control system, with semantic versioning, where each major version represents incompatible contract changes and minor versions represent backwards compatible contract changes. The active services are discovered by querying the deployed services in the container orchestration tool.

Alternatives: Information collected about dependencies between procedures can be translated into an ontology language [19] and stored in a registry. The main advantage of this approach is that the registry can be integrated with a reasoning system capable of performing deductive inferences, such as seeing the impact of a service failure across the system, and the resulting cascading failures. This approach delivers fine-grained data about impact of contract changes and reduces the burden in the development of other API management tools. This approach has not been explored because it falls outside the scope of this thesis, and can be used as complementary solution to the previous approach.

5 Implementation

5.1 Architecture

The system was built using a modular approach, each module is entirely self-contained, making it simple to expand or replace system functionality. The system components are split into two distinct areas: runtime, and pipeline components. The runtime comprises all essential components to the system liveliness, whereas the pipeline covers all components needed during the system's provisioning and development.

5.1.1 Runtime Architecture

The only novel component introduced in a typical microservices runtime environment is the proxy adapter. An example with three microservices is depicted in Figure 5.1. In this example, the Account's microservice has undergone three major API changes. The Inventory microservice, which is dependent on the previous service, has been updated to reflect the changes, but the Shipping microservice has not been updated and is two versions behind.

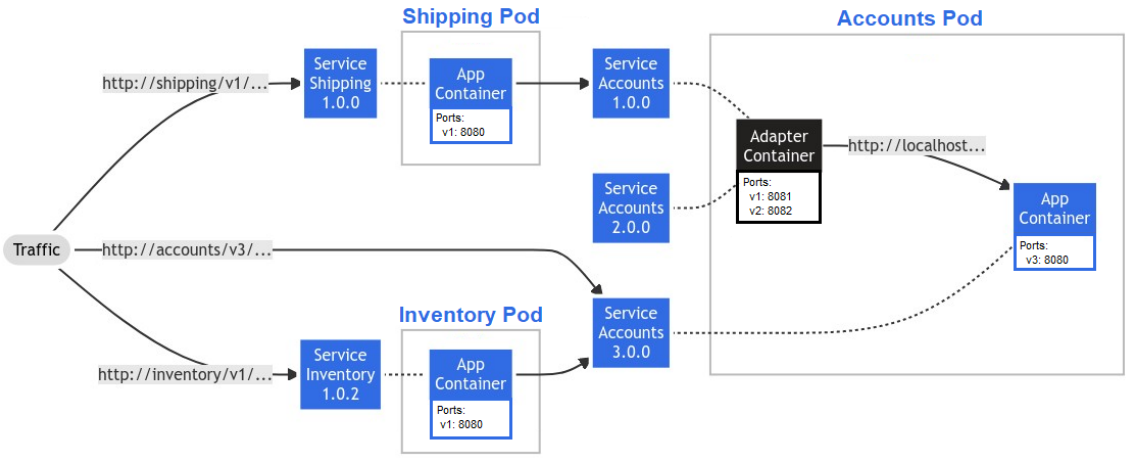

Figure 5.1: Example of a runtime configuration

Every contract version that is still in use has its own unique K8s service, as can be seen for the three versions of the Accounts microservice. Older versions are supported by the adapter, while the most up to date version is directly supported by the microservice app. A consumer accesses a producer app via the service that has the same version as the one specified in its code.

Containers are selected in services policies by attributing names to their exposed ports and assigning the same name on the target port of the service policy. In adapter containers, one port is exposed for each of the supported contract versions and named with the version number.

When a microservice app is upgraded via a rolling update, the services never become unavailable: they point either to the adapter containers or the application containers, depending on whether the affected pods finished the upgrade process. In the traditional approach all consumers would need to be upgraded in tandem with their consumed services in order to avoid downtime.

5.1.2 DevOps Pipeline

A proof-of-concept pipeline was developed, its architecture is depicted in Figure 5.2. The nodes depicted in blue represent the introduced novel stages. If a service only underwent changes that did not affect its API or dependencies, the introduced stages are skipped.

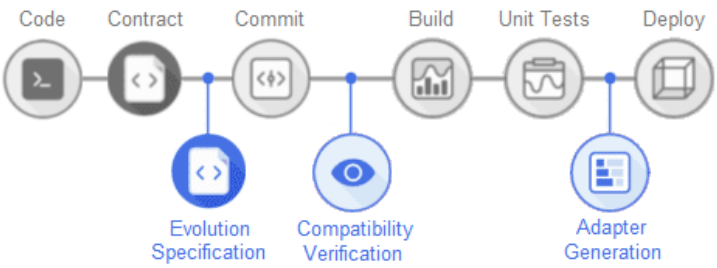

Figure 5.2: DevOps Pipeline

The pipeline was developed using the Jenkins [45] pipeline suite, each of the novel stages is supported by a distinct Java application that is pulled from version control and executed directly on the pipeline, whereas the other stages are supported natively by the suite. The modules that enable each novel stage will be discussed in detail in the following sections.

5.2 Contract Representation

The OpenAPI specification is the most widely adopted WADL for HTTP services, instead of designing yet another WADL, it was adopted the OpenAPI specification to support the needs of the implementation. Designing a WADL tailored for this problem would be counterproductive because teams rely on other tools, and supporting an additional

WADL would require the development of an extensive converter tool that would need to be constantly maintained in order to support changes in new versions of the OpenApi and the new WADL specification's.

To represent records in messages, it was chosen the JSON format, because of its simplicity and widespread adoption. There are more performant formats that have a binary representation, however such formats are typically not supported by front-end frameworks, and for the purpose of this prototype it is not important to measure the performance of different object serialization protocols, since there are already sources that provide comparisons in this regard between XML, JSON, Protobuf, Thrift and other popular formats [47].

The schema of JSON records must be described. The OpenAPI standard includes a schema description language in its Web API description language, that supports JSON records. There are dedicated JSON schema description languages (such as the JSON-schema project [37]), however, using a language other than OpenApi, would necessitate decoupling the specification of schemas from the specification of HTTP endpoints. This would require additional effort on the part of the developing team to maintain the cross-references between each specification manifest, and their versions.

A snippet of a *Pet Store* OpenApi contract can be seen in Figure 5.3. In this example we can view the specification of a JSON schema for a Pet data object, as well the specification for an HTTP endpoint. OpenApi allows properties to be defined robustly with specification of not only their type but also their format, as shown in line 31.

The types and formats of properties are standardized between object schemas and HTTP parameters. Properties have the following data types: string, number, integer, boolean, array, and object. Property formats are free-form, yet there are established conventions for the most commonly used formats.

The current implementation only supports services with a RESTFul API. Advanced OpenAPI specification features such as parameters with alternative types are not supported (e.g Type:=OneOf(String, Integer)).

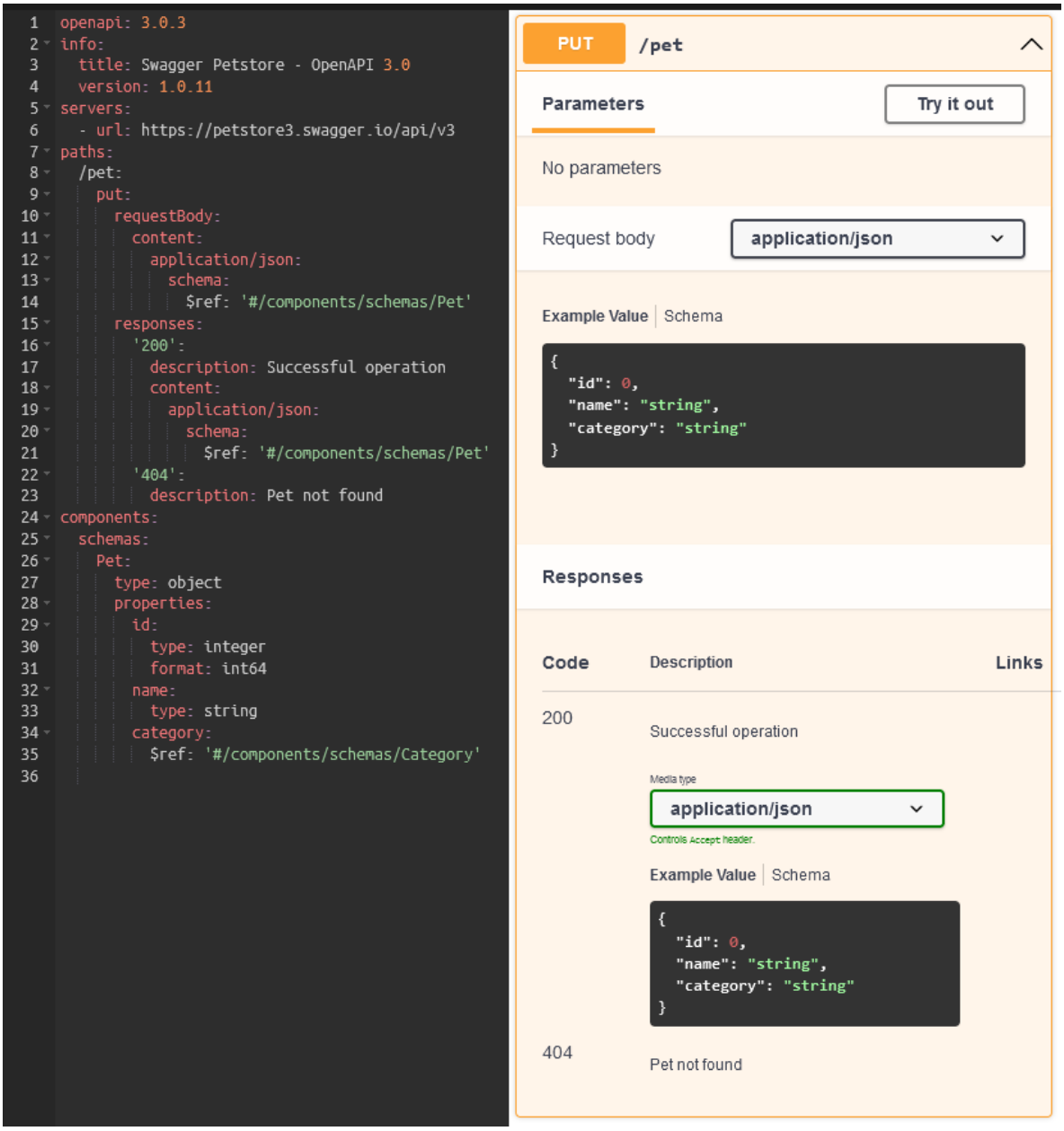

Figure 5.3: OpenAPI contract example

The specification of the dependencies of a service was done using Helm [23]. Helm is package manager for Kubernetes that allows developers to more easily package, configure, and deploy applications and services onto Kubernetes clusters. A chart is a Helm package, composed of a collection of files that describe a related set of Kubernetes resources. The most significant components of a Chart are the:

- The Chart.yaml, that allows the definition of the Chart dependencies and other meta-information about the application such as its name and version;
- The Template's directory, that contains incomplete Kubernetes manifests *Templates* such as Services, Deployments, DaemonSets, Namespaces and so on;
- The Values.yaml, that contains key-value pairs used to complete the definition of each template. Values can be overridden when the chart is deployed, the file is used to define the default values.

A single chart might be used to deploy something simple, like a single Kubernetes pod, or something complex, like a complete web app stack with HTTP servers, databases, caches, and so on. The typical approach is to define an individual chart for each distinct microservice, so that they can be upgraded individually. If a single chart is used to deploy multiple services it is no longer possible to represent accurately the dependencies of each service in the chart definition, because the chart syntax doesn't allow the individual definition of the dependencies of each service, it only allows the declaration of the dependencies of the deployment as a whole.

An example of the definition of a service/chart dependencies can be seen in Listing 5.1. The dependencies of a service are mapped indirectly, each dependency point to a Helm chart that is implicitly associated with one service.

```
dependencies:
    - name: nginx
      version: "1.2.3"
      repositiry: https://example.com/charts
    - name: memcached
      version: "3.2.1"
      repositiry: https://another.example.com/charts
```

Listing 5.1: Chart.yaml example

The declaration of resources consumed by a service is defined in chart template files. For each deployment template it was defined the resources consumed by its containers by setting the minimum amount of CPU and Memory resources (represented on the requests key) and by setting a limit on these resources (represented on the limits key). An example of the definition of a service resources can be seen in Listing 5.2.

```
apiVersion: v1
kind: Pod
sepc:
    containers:
    - name: app
      image: images.my-company.example/app:v4
      resouces:
        request:
          memory: "64Mi"
          cpu: "250m"
        limits:
          memory: "128Mi"
          cpu: "500m"
```

Listing 5.2: Helm template resources example

5.3 Contract Interpretation

The interpretation of a contract specification is done in a dedicated module named ”Parser”. This module is meant to be used as Java library by other modules, it interprets contracts written in OpenApi format and provides a read-only Java interface that is agnostic to the original specification format.

The implementation of the parser module uses the Swagger-parser project [48] to convert OpenApi contracts into a Java POJO representation. The OpenApi POJO representation is accessed through a custom read-only wrapper class and interface. All the forthcoming modules operate under this interface instead of the OpenApi POJO format.

Although it was adopted the OpenApi format the solution is not dependent on it, as the parser module provides a decoupling point. This enables the specification of different contract to be written in different or even mixed WADL, as long as the parser module is extended to support the new WADL's. The interpretation of service resources and dependencies written in Helm charts is also supported by the parser module under a separate interface.

5.4 Evolution Representation

The evolution of contracts is represented in a custom description language. The description only maps the details about the changed procedures of a service, it is not necessary to map the changes in service dependencies and resource requirements because they can be inferred by comparing the last two contract versions. The description language haves the following format:

- **Procedures:** List of the procedures that are adapted between contract versions.
  - **Endpoint:** The endpoint of the procedure and its HTTP method in the new contract.
  - **Prior Endpoint:** The endpoint of the procedure and its HTTP method in the previous contract.
  - **Messages:** List of the messages exchanged by the procedure in the new contract version (the request message and response messages).
    - **Type:** The response or request type in the new contract.
      (e.g. REQUEST, 200, 400, etc)
    - **Prior Type:** The response or request type in the previous contract.
    - **Parameters:** List of the parameters of the message (Path, Query, Header, Cookie and Body parameters).
      - **Key:** The identifier of the parameter.
        (e.g path|quantity)
      - **Resolution:** The specification of a resolution for a parameter.
        (eg. link=query|limit).
      - **Type:** The type of the parameter.
        (eg. String)
      - **Format:** The format of the parameter.
        (eg. yyyy-mm-dd hh:mm:ss)

The first segment of parameter keys represents the parameter location (e.g.Path, Query, Header, Cookie and Body parameters), the second segment represents the ”name” of the parameter.

$$\langle location \rangle | \langle name \rangle \quad (e.g\ path|quantity)$$

Schemas used in a message body are flattened into multiple parameters:

$$\textbf{Item}\{id, price\} \rightarrow json|item.id,\ json|item.price$$

The schema Item is flattened into two distinct parameters, which are mapped by specifying the hierarchy path in the parameter name.

As previously mentioned in Section 4.2.4, resolutions can have one of three distinct types DefaultValue, Link or Function. The type of resolution is represented in the first segment of a resolution rule, while the second segment represents the resolution itself:

- **Default value resolution:**

$$default = \langle value \rangle \quad (e.g\ default = true)$$

- **Link resolution:**

$$link = \langle location \rangle | \langle name \rangle \quad (e.g\ link = query|limit)$$

- **Function resolution:**

$$function = \langle expression \rangle \quad (e.g\ function = json|name + json|surname)$$

Function resolutions are written as one line Java expressions that can use parameters of the previous contract. The parameters are referenced by their location and name.

The description language supports arrays in request and response schemas. Arrays are defined by bracketing the array elements of a parameter.

$$\textbf{Item}\{tags : Tag[]\}\ \ \textbf{Tag}\{id : String\} \rightarrow json|item.[tags].id$$

$$\textbf{Item}\{tags : Tag[]\}\ \ \textbf{Tag}\{keywords : String[]\} \rightarrow json|item.[tags].[keywords]$$

Link resolution's between two parameters with the same number of array elements are resolved by matching the members of the different array elements. If a Link resolution uses parameters with a different number of array elements, the first members of the unpaired array elements are selected, while remaining array members are ignored.

The Listing 5.3 depicts an example of a evolution description in this language.

```
procedures:
- endpoint: /pets GET
  endpoitPrior: /pets GET
  messages:
  - type: Request
    typePrior: Request
    parameters:
    - key: query|limit
      resolution: link=path|limit
      type: String
      format: none
  - type: '200'
    typePrior: '200'
    parameters:
    - key: json|[pet].name
      resolution: function=json|[pet].first_name + [pet].surname
      type: String
      format: none
    - key: json|[pet].birth
      resolution: function=LocalDate.now().getYear() - json|[pet].age
      type: String
      format: yyyy
```

Listing 5.3: Evolution specification example

To minimize the documentation effort it was developed a GUI editor that facilitates the specification of evolutions. Figures 5.4 and 5.5 depict the editor interfaces. The editor was implemented as an IDE Plugin, the current implementation supports the IntelliJ integrated development environment.

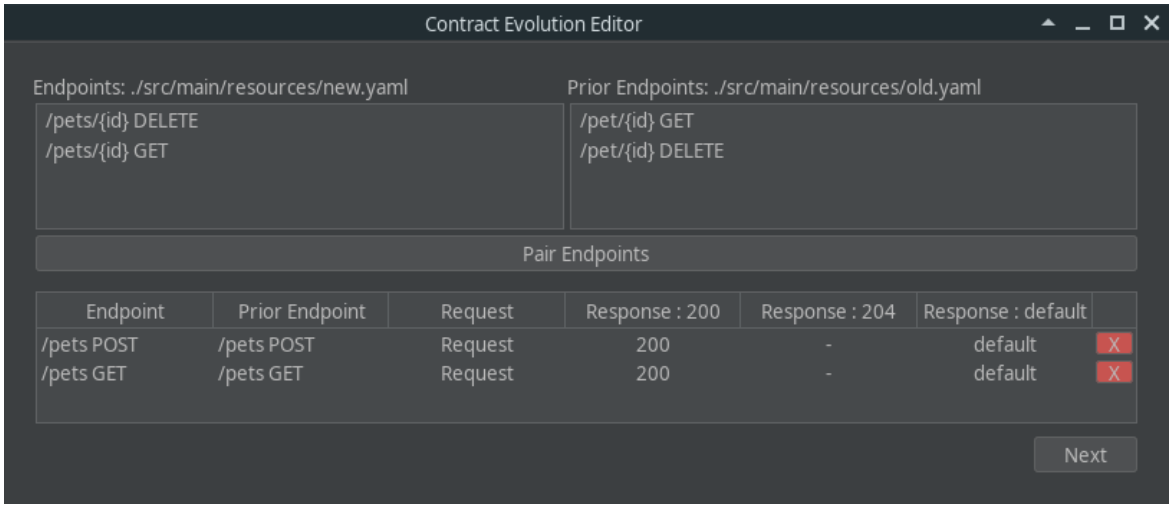

Figure 5.4: Evolution editor, procedure mappings

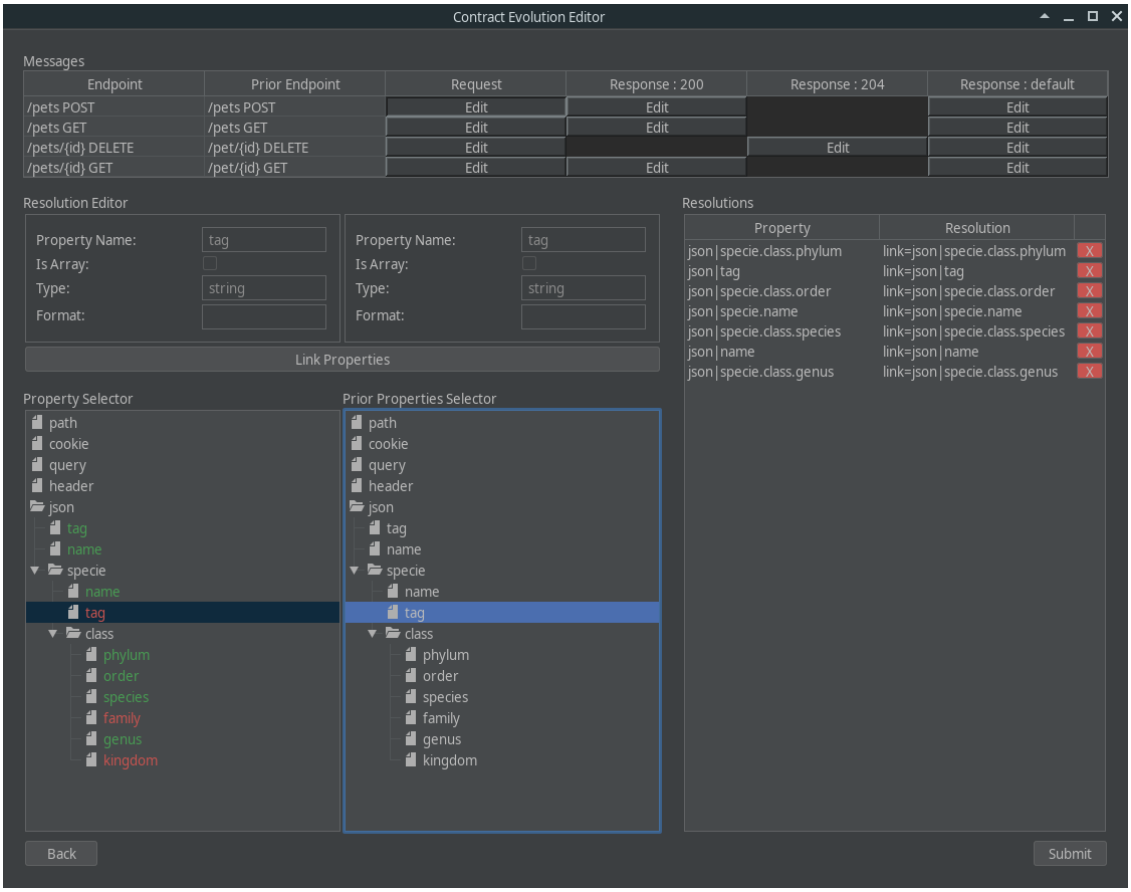

Figure 5.5: Evolution editor, parameter mappings

The editor compares two contract versions, and automatically maps all the elements that haven't changed from one version to the other, the developer is left with the task of explicitly mapping the remaining elements.

The editor starts by requesting the developer to map endpoints in the new contract to endpoints in the old contract, afterwards the developer must match each of the endpoint messages and specify a resolution for all the parameters that changed in a contract, unresolved parameters are highlighted in red.

The right column of the editor 5.5 holds the resolutions that were already mapped. The uppermost left column allows the developer to write rules for each unresolved parameter. The developer can select leaf parameters in the hierarchy, or choose intermediate parameters. If they choose an intermediate parameter and input a resolution rule, then all child parameters are also resolved.

After all parameters are resolved the developer submits the resolutions, and the editor outputs an evolution specification file with the same description language presented in the beginning of this section.

5.5 Compatibility Verification

The compatibility verification process is done by a dedicated module named "Verifier". This module was implemented in Java as a command-line application. The module verifies if the provided evolution specification is valid. In Algorithm 1 it is presented the pseudo-code of the compatibility verification process. It starts by checking if all the procedures and messages of prior contract are present in the specification, then it verifies the validity of the resolutions for each parameter of the new contract messages.

Resolutions can be invalid due to four reasons:

- Invalid syntax in resolution expressions;
- References parameters that don't exist in the previous contract;
- Uses functions that are not supported;
- Uses resolutions that have a different type and format then the parameter.

The present implementation of the verifier module does not check whether the cluster has adequate computational resources to provision the service or whether all service dependencies are alive and reachable. These functionalities were not implemented since the Helm deployment process already performs these verifications prior to the deployment of a service. Nonetheless, including these verifications in the verifier module could be useful in order for the solution to be decoupled from Helm.

The module also does not verify if the type and format of a function resolution matches the type and format of a parameter. To support this validations in verifier module with would be necessary to use a dedicated language for specifying function resolutions. The verification of correct function resolution types is done alternatively by the generator

module that verifies compile-time errors when parameter references are substituted by real values.

Algorithm 1 Evolution validation algorithm

```
 1: procedure ValidateEvolution(Evolution, PriorContract, NewContract)
 2:     assert PriorContract.procedures ⊆ Evolution.procedures → priorEndpoint
 3:     for all pd ∈ Evolution.procedures do
 4:         assert pd ∈ NewContract.procedures
 5:         PriorMessages ← PriorContract.getMessages(pd.priorEndpoint)
 6:         NewMessages ← NewContract.getMessages(pd.endpoint)
 7:         assert PriorMessages ⊆ pd.messages
 8:         for all msg ∈ pd.messages do
 9:             assert msg ∈ NewMessages
10:             assert msg.parameters = NewMessages.get(msg).parameters
11:             for all prm ∈ msg.parameters do
12:                 OldMessage ← PriorMessages.get(msg.priorType)
13:                 assert VerifyResolution(prm, OldMessage)
14:             end for
15:         end for
16:     end for
17: end procedure

18: procedure VerifyResolution(Parameter, OldMessage)
19:     Resolution ← Parameter.resolution
20:     if Resolution.type = Value then
21:         assert Resolution.value instanceof Parameter.type
22:         assert format(Resolution.value) = Parameter.format
23:     else if Resolution.type = Link then
24:         assert Resolution.parameter ∈ OldMessage.parameters
25:         assert Resolution.parameter.type = Parameter.type
26:         assert Resolution.parameter.format = Parameter.format
27:     else if Resolution.type = Function then
28:         assert ~~Resolution.function.returnType = Parameter.type~~
29:         assert ~~Resolution.function.returnFormat = Parameter.format~~
30:         for all prm ∈ Resolution.function.parameters do
31:             if prm.key == null && prm.resolution! = null then
32:                 assert VerifyResolution(prm, OldMessage)
33:             else
34:                 assert prm ∈ OldMessage.parameters
35:             end if
36:         end for
37:     end if
38: end procedure
```

5.6 Adapter Generation

The generation of the proxy adapter code is done in a dedicated module named ”Generator”. The adapter is generated by populating a template implementation with the information provided in evolution specifications. The adapter must be able to support all prior versions that are still in use, and adapt them to the last contract version.

Evolution specifications only provide information on how to adapt messages from contiguous versions (eg. v1→v2). The information necessary to adapt messages from discontinuous versions is obtained by chaining the evolution specifications and merging their resolutions. Figure 5.6 exemplifies the merge process.

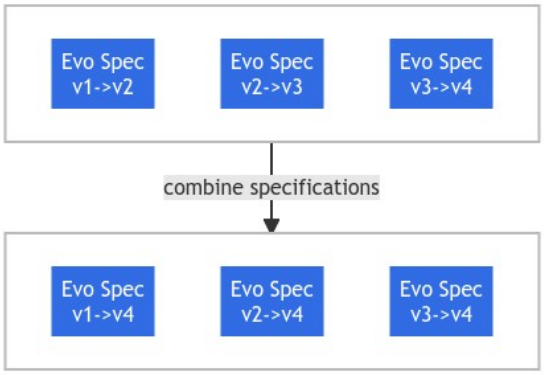

Figure 5.6: Evolution specification merge process example

This module begins by querying the active versions of the target service with the Kubectl command-line tool. Kubectl [4] is a tool that interfaces with Kubernetes clusters and can be used to inspect and manage cluster resources.

After determining the active service versions, this module retrieves all evolution specifications up to the oldest version that is still active, and then combines them so that all specifications link to the current version.

Link resolution rules can be chained together in a straightforward manner. Function resolutions are chained together by recursively replacing missing parameter references with their resolutions . Below in Table 5.1 it is presented an example, where resolution rules are chained incrementally from the two last versions to the oldest version:

$Transition$	$Missing^*$	$Resolution for parameter :== path\|seven_7$
$v_3 \rightarrow v_4$	7	$function = path\|five_5 + path\|two_2$
$v_2 \rightarrow v_4$	7,2	$function = path\|five_5 + (default = 2)_2$
$v_1 \rightarrow v_4$	7,5,2	$function = (function = path\|three_3 + (default = 2)_2)_5 + (default = 2)_2$

*Indicates the missing parameters in the prior contract (e.g 7⟨=⟩path|seven)

Table 5.1: Chained resolution rules example

5.7 Adapter

The adaption approach, for each endpoint, re-constructs the entire message from zero and populates the parameter resolutions with the information of the original message.

In Algorithm 2, it is presented the pseudo-code of the template implementation of the adapter proxy. The terms highlighted in blue are replaced by a template engine. The sections highlighted in orange represent code fragments that are repeated for each of the mentioned terms in the section comment.

The adapter current template implementation does not support HTTPS, however this could be added in future work. Because TLS is mandatory in HTTP 1.2, the implementation only supports HTTP 1.1.

The adapter was built with Spring Boot and an Apache Tomcat/9.0.65 HTTP server, the base image of the adapter docker container is an "openjdk:11-nanoserver". In order to prevent the JVM from calling garbage collector frequently and reallocating the heap memory while Tomcat is trying to serve requests, the JVM has started with a higher maximum heap memory, and the initial heap memory size was set to the same value as its maximum memory size. The minimum and maximum number of threads for Tomcat was set to 25 and 2000 respectively, in order to support a higher load of requests.

Algorithm 2 Proxy adapter template algorithm

```
#For each endpoint {
procedure AdaptEndpointRequest(Path, Query, Headers, Cookies, Body)
    Body ← FlattenBody(Body)
    new_method ← NewMethod
    new_url ← NewURL
    new_params ← {}
    #For each new request parameter {
        id ← param.key.id
        location ← param.key.location
        resolution ← Resolution(Path, Query, Headers, Cookies, Body))
        new_params.add(location, id, resolution)
    }
    new_body ← ComposeBody(new_params.get(body))
    new_path ← ComposePath(new_url, new_params.get(path||query))
    new_headers ← ComposeHeaders(new_params.get(header||cookie))
    response ← ForwardRequest(new_path, new_method, new_body, new_headers)
    return AdaptEndpointResponse(response)
end procedure

procedure AdaptEndpointResponse(Response)
    Status ← Response.status
    Headers ← Response.headers
    Cookies ← Response.cookies
    Body ← FlattenBody(Response.body)
    #For each prior response status{
        if Status = PriorStatus then
            new_status_code ← NewStatusCode
            new_params ← {}
            #For each prior response parameter{
                id ← param.key.id
                location ← param.key.location
                resolution ← Resolution(Headers, Cookies, Body))
                new_params.add(location, id, resolution)
            }
            new_body ← ComposeBody(new_params.get(body))
            new_headers ← ComposeHeaders(new_params.get(header|cookie))
            return ComposeResponse(new_body, new_headers, new_status_code)
        end if
    }
end procedure
}
```

6 Evaluation

6.1 Benchmark platform

A benchmark platform was developed to evaluate our solution to existing ones. This section discusses the proposed benchmark platform's requirements and design. Aderaldo et al. [1] propose an initial set of requirements to support repeatable microservices research (Table 6.1).

Requirement	Assessment Rationale
R1: Explicit Topological View	The benchmark should provide an explicit description of its main service elements and their possible runtime topologies.
R2: Pattern-based Architecture	The benchmark should be designed based on well-known microservices architectural patterns.
R3: Easy Access from a Version Control Repository	The benchmark's software repository should be easily accessible from a public version control system.
R4: Support for Continuous Integration	The benchmark should provide support for at least one continuous integration tool.
R5: Support for Automated Testing	The benchmark should provide support for at least one automated test tool.
R6: Support for Dependency Management	The benchmark should provide support for at least one dependency management tool.
R7: Support for Reusable Container Images	The benchmark should provide reusable container images for at least one container technology.
R8: Support for Automated Deployment	The benchmark should provide support for at least one automated deployment tool.
R9: Support for Container Orchestration	The benchmark should provide support for at least one container orchestration tool.
R10: Independence of Automation Technology	The benchmark should provide support for multiple technological alternatives at each automation level of the DevOps pipeline.
R11: Alternate Versions	The benchmark should provide alternate implementations in terms of programming languages and/or architectural decisions.
R12: Community Usage & Interest	The benchmark should be easy to use and attract the interest of its target research community.

Table 6.1: Benchmark requirements [1]

In addition to the requirements listed above, it was considered that the following

requirements should also be included:

- The capacity to evaluate different solutions in comparable scenarios while using the same evaluation criteria;
- The capacity to evaluate diverse solutions without changing their implementation;
- Experiments must be easy to share and replicate;
- It should be possible to aggregate reported metrics for a specific time period between two events, such as the start and end of a service's evolution;
- Users must be able to specify how and when each service should evolve via a configuration file or directly through a terminal.

The architecture of the benchmark platform can be seen in Figure 6.1, and its components are described subsequently.

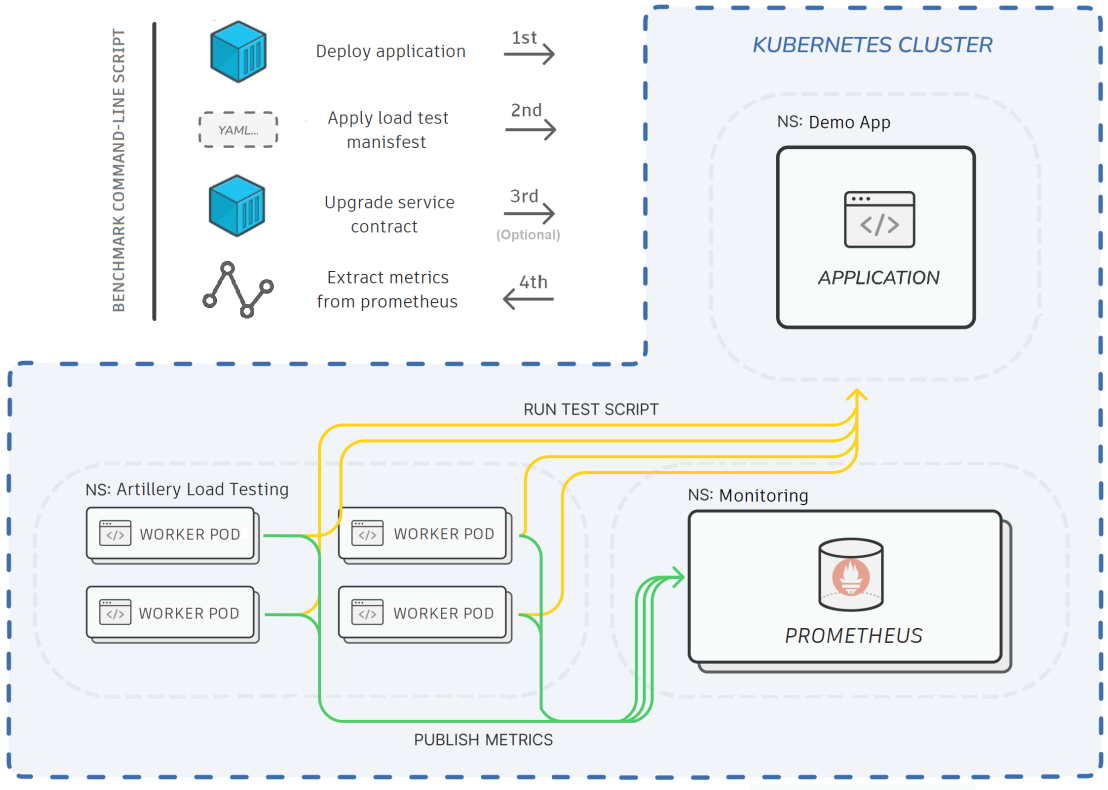

Figure 6.1: Benchmark platform architecture

Kubernetes [42] is the test environment. It hosts the applications under test, the monitoring system, and the load test workers.

Promotheus [49] is a pull-based monitoring system. It periodically sends HTTP scrape requests, the response to these requests is parsed in storage along with the metrics for the scrape itself. Prometheus provides a query language that allows the metrics

to be aggregated by events, components or metadata. The gathered metrics can be consulted by external tools or directly visualized in this platform via dashboards.

Artillery [24] is an open-source load testing suite for developers and system administrators.

Artillery Operator [25] enables the creation and execution of distributed Artillery load tests from Kubernetes schedule jobs. The architecture of this tool is depicted in Figure 6.2.

Helm [23] is a package manager for Kubernetes that allows developers to more easily deploy, package, and configure applications into Kubernetes clusters.

Helmfile [40] extends Helm's capabilities by enclosing it in a declarative specification that allows the composition of multiple Helm charts as a single deployment artifact.

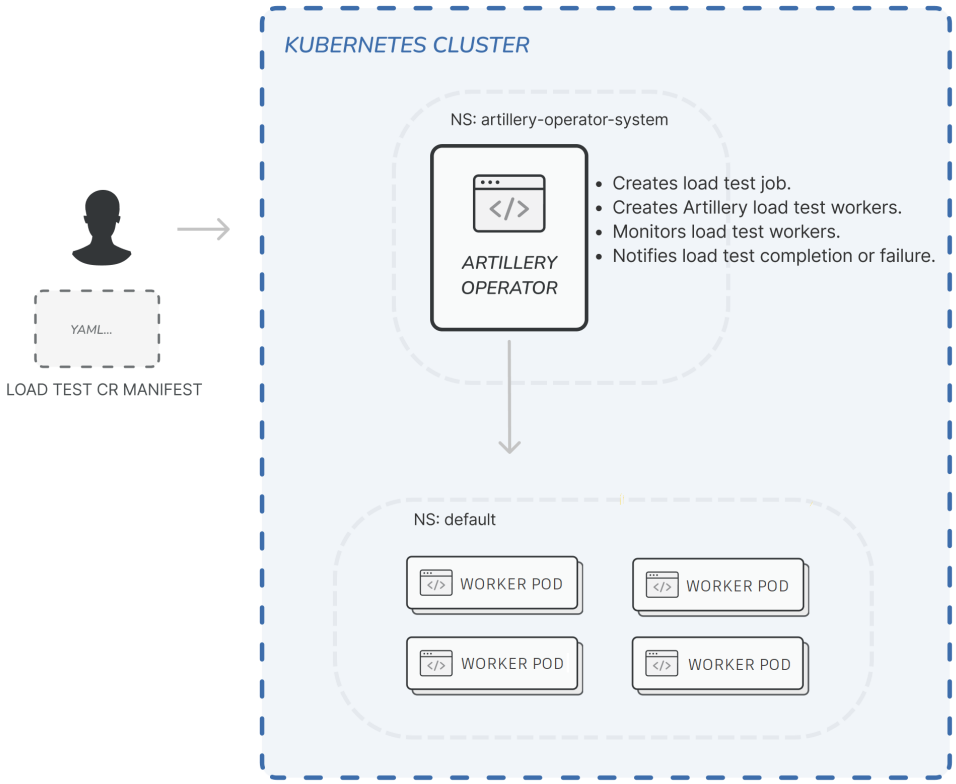

Figure 6.2: Artillery operator architecture

Experiments are launched using a custom command-line tool. The command-line tool can also be used to install all the benchmark platform's required components in a Kubernetes cluster. The tool allows for the execution of multiple experiments in sequence without the need for developer intervention. The components and resources of an experiment are automatically deleted, and experiment metrics are automatically gathered at the end of the experiment. The experiment metrics are obtained by querying the prometheus

monitoring system, which returns series of observations recorded throughout the experiment.

An experiment is composed by four manifests:

- **Helmfile** specifies the application under test;
- **Artillery workload job** defines the test scenarios of the experiment;
- **Metrics script** holds the queries that should be performed after the experiment ends;
- **(Optional) Evolution script** outlines the deployment actions required to upgrade services during the experiment execution.

Each of the four manifests can include template variables that are changed using a configuration file. An experiment is started by passing the configuration file to a run script.

The demo application and load test scripts support the following configurations:

- Warm up phase - duration, arrival rate;
- Ramp up phase - duration, min/max arrival rate;
- Sustained load phase - duration, arrival rate;
- Number of distributed artillery workers;
- Number of application replicas;
- Payload size of the sent messages;
- Fanout - Number of subsequent parallel calls for a single request.
- Depth - Number of subsequent blocking calls after an initial request;

6.2 Evaluation Methodology

The setup and the methodology of the conducted experiments is described below:

Environment - The experiments were conducted in a cloud environment with 10 virtual machines. All 10 nodes were hosted in the same datacenter in the region of Strasbourg, France. Each virtual machines was hosted in different physical machines, no two VMs shared the same hardware. Each node is interconnected with a bandwidth of 2 Gb/s. Each virtual machine is equipped with 4GB of RAM and 2 virtual cores of an Intel Core Haswell (no TSX) CPU.

Tested microservice system - To test the cost of the adapter system a demo application was developed. The demo application holds a configurable amount of services that run an identical image. The image is a simple HTTP Spring server that forwards and returns received requests unchanged. Environment variables allow for the

configuration of the interconnections between services. The number of parallel calls (fanout) and recursive calls (depth) between services after an request are both configurable. The demo application was developed in two separate versions with unique contracts. In the first version, one of the procedures of the application was altered. All the parameters in the body's schema of this procedure were renamed.

Experiment Setup - To inject load in the experiments, 10 workers of the artillery load testing tool were used, the workers are executed remotely in the cluster and were spread in 6 nodes. These workers are supported by are monitored by a common Kubernetes job. The workers are installed and removed from cluster using the artillery-operator project [26]. In each experiment all the resources of previous experiments were deleted, and a new namespace was created to avoid contamination in the metrics extracted. The experiments were carried out sequentially with a 5 minutes break in between.

Each experiment had three phases: a warm up phase of 30 seconds with an arrival rate of 5 virtual users/second; a ramp up phase of 2 minutes; a sustained load phase of 6 minutes. The metrics were only extracted during the sustained load phase. The first and last 30 seconds were left out to allow time for all the artillery worker threads to enter the sustained load phase. It was excluded tests that had a high enough throughput to saturate the application. The error rate and timeouts were 0 during all the presented experiments.

Metrics - In each experiment it was extracted 3 metrics: the CPU resources, the memory resources and latency of the system. The resources were only queried from the adapters, and the replicas that belong to the target application under test. The resources consumed by the monitoring system and artillery workers were excluded from the results. The throughput and latency were extracted by parsing the reports of the artillery load testing tool. The CPU resource samples were taken from each node at 5 seconds intervals using the prometheus node exporter. Memory resource samples were gathered from each container at 5 seconds intervals using the kubelet node agent API. The latency statistics were calculated with a period of 10 seconds over 5 minutes, the results obtained represent the average of the reported 30 periods.

6.3 Experiments

Four experiments were carried out in order to test the implementation of the prototype: the point-to-point experiment, the weighted point-to-point experiment, the bi-partition experiment, and the contract upgrade experiment. The experiments are detailed below.

6.3.1 Point to Point Experiment

The topology of this experiment is represent in Figure 6.3. The primary goal of this experiment was to assess the overhead of the adapter in terms of latency and computational resources.

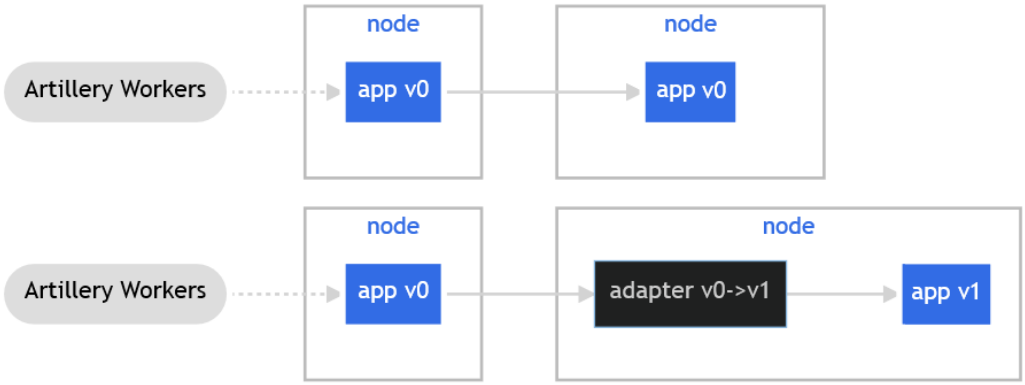

Figure 6.3: Point-to-point experiment

The experiment is composed by two microservices: the first receives requests from artillery workers and forwards them to the second microservice, which responds with the request. It was first established a baseline of comparison by doing the tests without the adapter, and then, the adapter was introduced and the tests were repeated.

The measured CPU overhead of the adapter over a range of request rates is depicted in Figure 6.4. The overhead was calculated by processing the averages of the measured CPU resources in the baseline and adapter tests, and then by dividing the two averages.

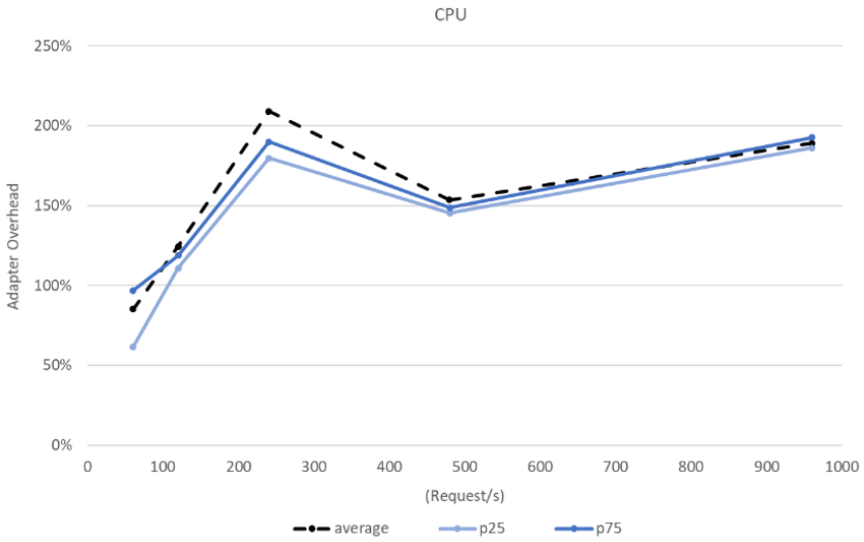

Figure 6.4: Adapter CPU overhead in the point-to-point experiment - 100 bytes (payload size)

In this experiment it is expected to have an overhead of 150% because: the adapter

and the microservices share the same request rate; the cost of the adapter is expected to be equal or to the cost of the tested microservices. The tested microservices were implemented with the same frameworks as the adapter, consequently adding the adapter will be equivalent to adding one more microservice to the existing two.

The overhead 84% in first data point is explainable by the configuration of the containers: the adapter and demo application containers were initialized in Kubernetes with a default of 100 millicpu (100 millicpu is equivalent to 10% of a CPU core), this CPU resources far exceed the capacity necessary to support a request rate 60 requests/second, consequently the container's CPU scheduling was throttle-down after start up; the overhead was inferior to 100% because the scheduled CPU resources in the experiment with the adapter, were throttled down faster than in the baseline experiment without the adapter. As can be seen in the results shown Figure 6.4 the measured overhead doesn't fall to far outside our expectations.

Figure 6.5 shows the measured memory overhead of the adapter over a range of request rates. The overhead was calculated by processing the averages of the measured memory resources in the baseline and adapter tests, and then by dividing the two averages.

The average cost of 190% in Figure 6.5 is understandable because: although being stateless, the adapter stores two transient messages in memory for each request, the request message and the adapted request message; the demo application only keeps one request in memory, it stands to reason that the adapter memory cost is roughly double of the demo application. As can be seen in Figure 6.6, the used memory of the adapter grows linearly with the request rate, as it was expected.

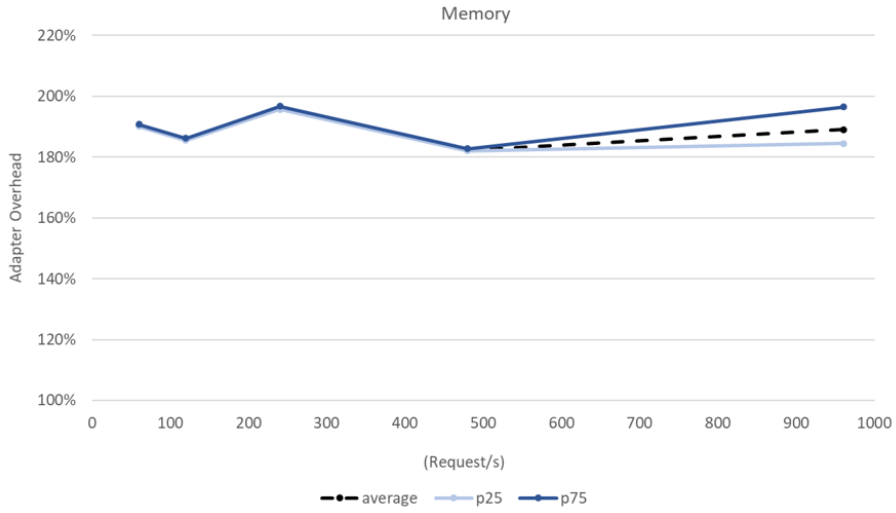

Figure 6.5: Adapter memory overhead in the point-to-point experiment - 100 bytes (payload size)

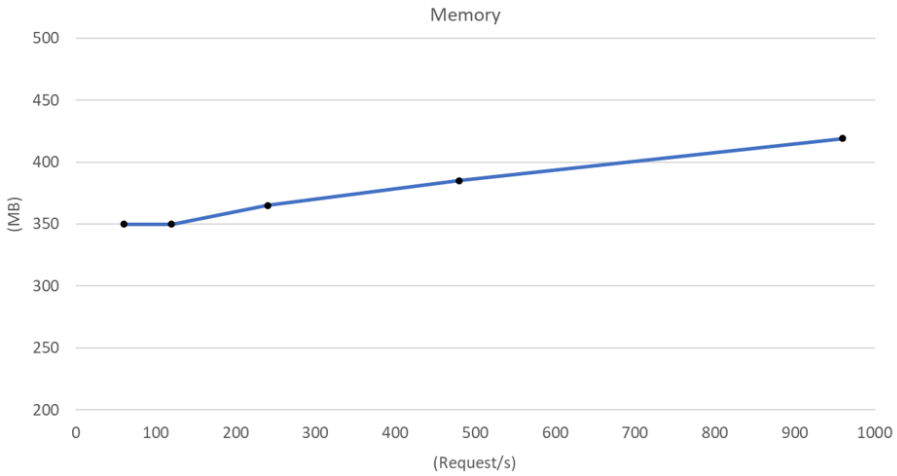

Figure 6.6: Adapter memory cost in the point-to-point experiment - 100 bytes (payload size)

The latency overhead of the adapter in the point-to-point experiment is depicted in Figure 6.7. The overhead was calculated by processing the averages of the measured latency in the baseline and adapter tests, and then by dividing the two averages.

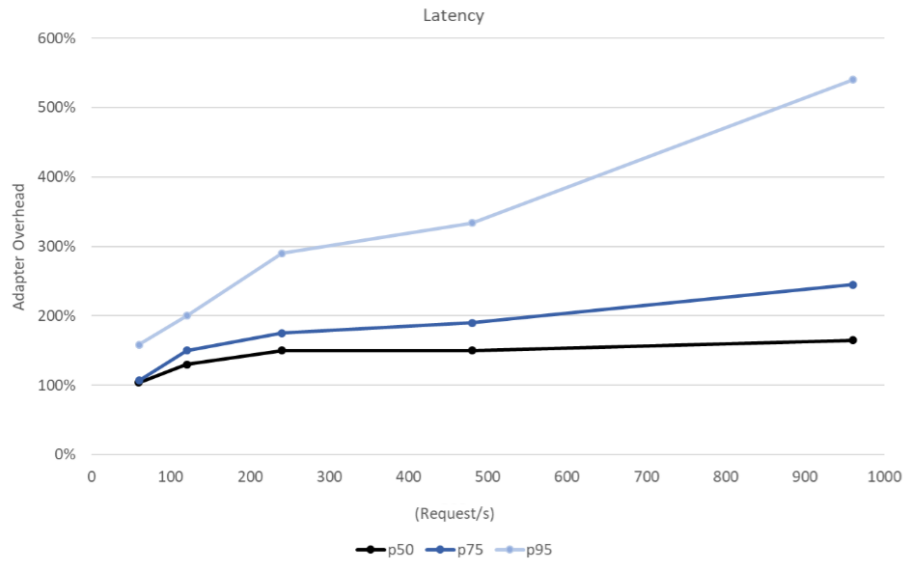

Figure 6.7: Adapter latency cost in the point-to-point experiment - 100 bytes (payload size)

It is expected for the overhead to be less or equal to 150%, because the latency overhead is largely attributable to communication costs. The adapter introduces an additional communication step ”app1→adapter→app2” to the baseline communication chain that already has two steps ”workers→app1” and ”app1→app2”. The introduced communication step is expected to have a slightly inferior cost to the other steps, as it is performed via interprocess-calls, while the others are done via a network with 2 GB/s of bandwidth. The hypothesis is confirmed by the experiment results in Figure 6.7.

The Figures 6.8 and 6.9 illustrate how larger payload sizes affect the latency and the consumed CPU resources. A request rate of 480 request per second was used in this experiment. The latency is expected to rise with larger payload sizes because the HTTP packet size is typically 1.5 KB, bigger messages will need to be sent in multiple packets. The CPU cost is also expected to rise because the cost of deserialization of messages grows with the message payload size.

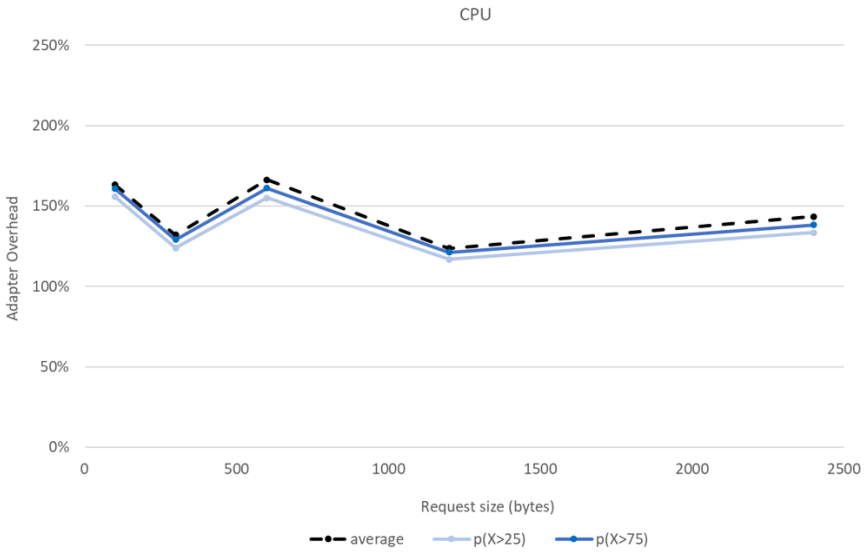

Figure 6.8: Adapter CPU cost in the point-to-point experiment - 480 request/s (arrival rate)

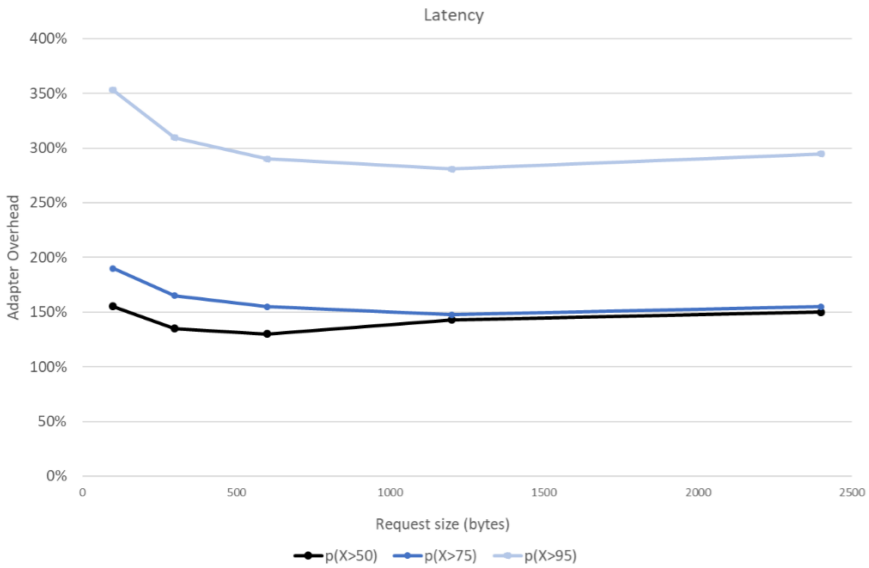

Figure 6.9: Adapter Latency cost in the point-to-point experiment - 480 request/s (arrival rate)

The results illustrated don't reveal any difference for the tested range of payload sizes. This can be explained by the fact that the messages were tested with payloads that ranged

in size from 100 to 2500 bytes, which fall within one or two HTTP packets. This range has initially selected because most services use HTTP messages with these payloads sizes. In future work this experiment should be repeated for larger payload sizes, up to 2 MB, to see how the adapter scales in this scenario.

6.3.2 Weighted Point to Point Experiment

The topology of this experiment is represent in Figure 6.10. This experiment is also composed by the two microservices, where the first microservice forwards a percentage of its request thorough the adapter and forwards its remaining requests directly to the other microservice. The primary goal of this experiment was to see if the adapter overhead is significantly lower in scenarios where only a small percentage of all requests require adaptation.

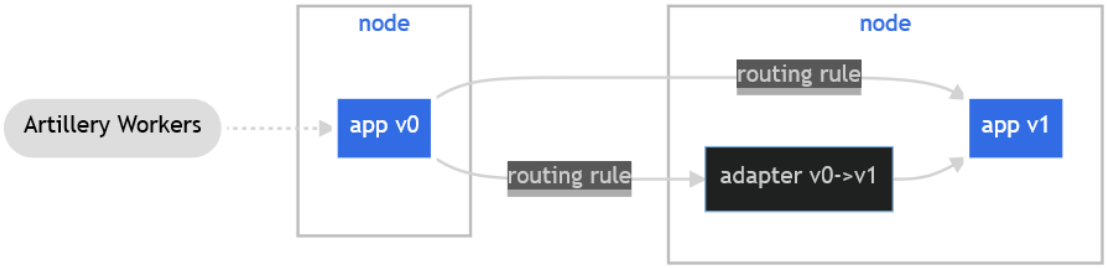

Figure 6.10: Weighted point to point experiment

Figure 6.11 depicts the measured CPU cost of the adapter, the horizontal axis represents the percentage of requests that need adaptation, and the vertical axis represents the CPU cores utilized by the adapter and demo applications.

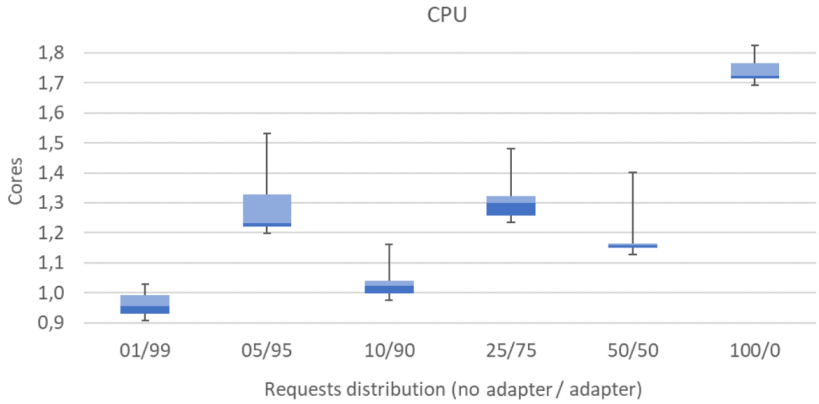

Figure 6.11: CPU used in the routing experiment - 480 request/s (arrival rate) 100 bytes (payload size)

In this experiment, the overhead is expected to gradually rise due to the increased

arrival rate caused by a higher proportion of adapted requests. To support higher request rates the adapter implementation will need to launch more client threads which, consequently, increase the consumed CPU resources.

When comparing the first and last data points in Figure 6.11, we can see that consumed resources have doubled, as expected.

The overhead also increases gradually in the other data points, with an estimated 20% margin of error. The large margin of error is explainable by the nondeterministic nature of the artillery workers. Remark that the artillery test has two scenarios: one where the worker calls an endpoint that can only be handled by using the adapter, and another in which the adapter is not utilized. Each scenario weight as set to reflect the corresponding test request distribution, however, the dispersion of requests is nondeterministic and will only be equal to the given weights over very long test periods. The duration of this experiment was set to 5 minutes for each data point.

The utilized memory of the adapter and demo applications in this experiment is depicted in Figure 6.12.

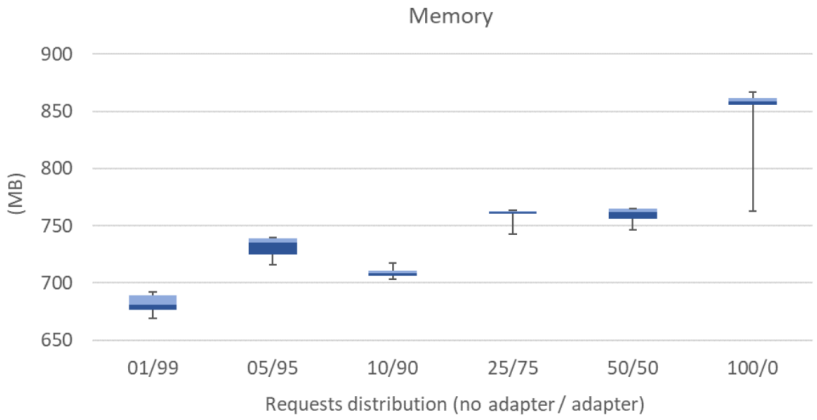

Figure 6.12: Memory used in the routing experiment - 480 request/s (arrival rate) 100 bytes (payload size)

It is expected for the used memory to increase linearly with the fraction of requests routed through the adapter because, as discussed previously, the adapter implementation spatial complexity is equivalent to double of the demo application implementation. The results obtained confirm the expectation.

Figure 6.13 shows the request latency as the proportion of adapted requests increases. It is expected that the latency will steadily increase, because of the same reasons previously stated. When comparing the first and last data points, we can see that the latency has doubled as expected. In the other data points, the latency remains relatively constant, this mainly due to the error introduced by the short test duration.

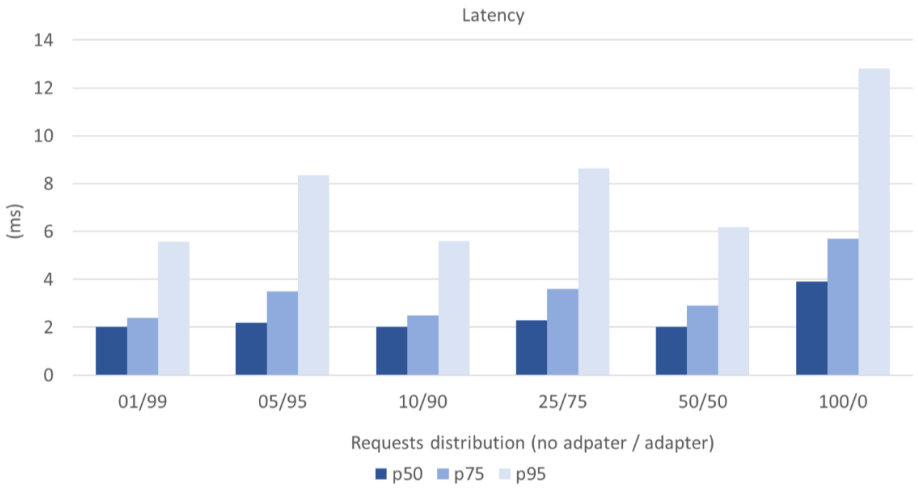

Figure 6.13: Request latency in the routing experiment - 480 request/s (arrival rate) 100 bytes (payload size)

6.3.3 Bi-partition Experiment

The goal of the experiment was to see if the conclusions obtained from the point-to-point experiment are also applicable in scenarios with multiple microservices. The experiment is composed by two disjoint sets of microservices, where the first set receives request from the artillery workers and forward the requests to the second of microservices. The topology is represent in Figure 6.14.

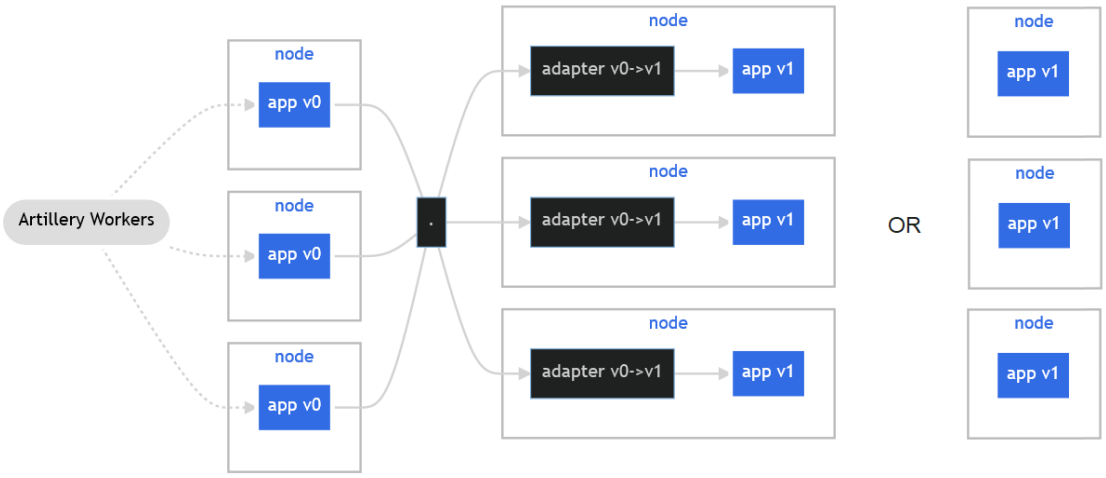

Figure 6.14: Bi-partition experiment

First, it was established a baseline of comparison by doing the tests without the adapter, then, it was upgraded all the microservices in the second set and added adapters to support communication across the disjoint sets.

The outcomes of the ”point-to-point” and ”bi-partition” experiments are equivalent, as seen in the Figures 6.15, 6.16 and 6.17.

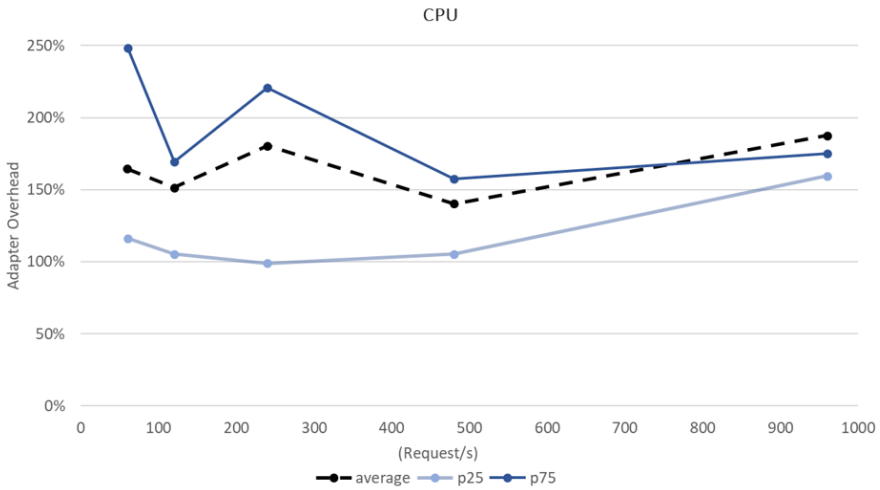

Figure 6.15: Adapter CPU overhead in the bi-partition experiment - 100 bytes (payload size)

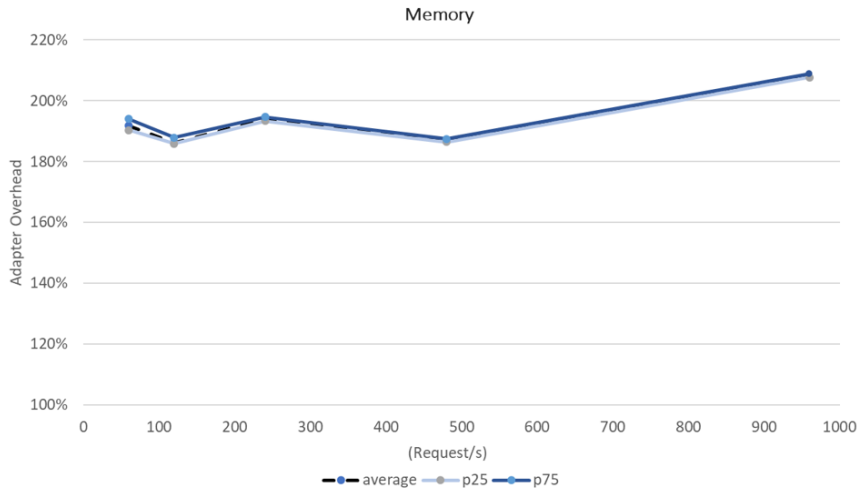

Figure 6.16: Adapter memory overhead in the bi-partition experiment - 100 bytes (payload size)

6.3.4 Contract Upgrade Experiment

The experiment's goal was to see if a service's throughput would decrease during a contract upgrade. The test was performed with the same topology as in the point-to-point experiment. A deployment operation, was started at minute 2, and finished at minute 3 during the test. In this deployment the proxy adapter was updated, and the contract of one of the services was changed without updating the consumer services.

The deployment was made with a rolling update strategy that incrementally updates pod instances with new ones. The error rate and timeouts during the experiment were zero. The experiment was done with an arrival rate of 960 requests/second. Multiple arrival rates were tested, it was selected the highest rate that didn't saturate the application and adapter servers.

The measured results showed that there was no change in throughput; the throughput

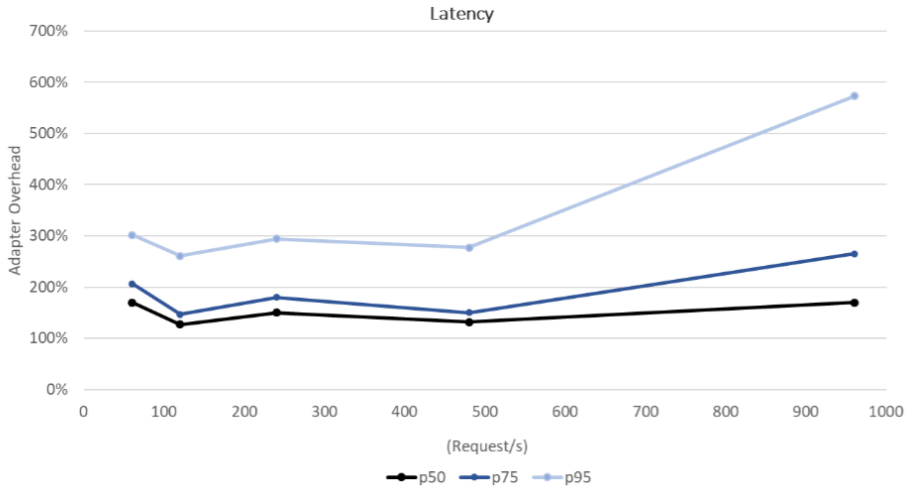

Figure 6.17: Adapter latency overhead in the bi-partition experiment - 100 bytes (payload size)

remained constant at 960 requests/s as expected. This is understandable because the rolling update strategy kills previous pods only when new pods are active and reachable.

7

Conclusions and Future Work

It was demonstrated that the evolution of service contracts can be achieved, without impacting consumer services, with the use of proxy adapters that dynamically convert the data sent between services. The safe evolution of microservice contracts can be achieved via mechanisms that support the automatic validation of deployment operations. Deployments that would lead to irremediably incompatible states are rejected, such as a microservice attempting to utilize a deleted or modified endpoint, or a microservice being deployed before its dependencies are alive and reachable.

Additionally, the proposed solution was designed over widely adopted workflows and documentation formats, such as DevOps pipelines and the OpenAPI description language, which enables its compatibility with other management tools.

The proposed solution requires the documentation of the services capabilities, dependencies and consumed computational resources. These aspects were documented using conventional description languages (eg. OpenAPI contracts and Helm charts). The only additional documentation necessary is the specification of contract evolutions, which is minimized via an IDE tool that automatically identifies unmodified service procedures and possible contract evolutions (Section 5.4).

The approach is agnostic to the underling application, all of its components don't require the modification of application microservices or their architecture. The solution may be plugged into any distributed application that uses the HTTP communication protocol and runs in a Kubernetes environment.

One conclusion drawn from the results is that the latency introduced by an adapter installed as side-car container is similar to the latency introduced by an additional call between two distinct nodes in the same datacenter. Thus, installing the adapter in the same node as the adapted service is unnecessary. Instead, it would be more beneficial to host the adapter in a different node of the same node pool, and use an adapter with a more generic implementation that can handle the adaptation of multiple services. Such an approach would require fewer computational resources since a distinct adapter replica wouldn't need to be installed on every service replica, in other words, a single adapter would serve multiple services.

Furthermore, the acquired findings show that the adapter CPU and memory cost grow linearly with the arrival rate of requests that must be adapted. If only a small fraction of requests requires adaptation, the computational cost of the adapter will be low in comparison to the cost of the application.

In future work, an interesting avenue to explore would be the implementation of the adapter using HTTP server's that support different threading models, the current implementation was made under the Blocking I/O (BIO) model. In the BIO model, each request is processed by a different thread. This adds a significant thread overhead and impairs highly concurrent applications. The adoption of a Non-blocking I/O (NIO) model could improve performance and lead to lower CPU resource usage.

Information gathered about procedure dependencies can be translated into an ontology language and stored in a registry. This approach would provide richer data and reduce the burden in developing other API management tools; however, it is outside the scope of this thesis and has not been investigated. This aspect could be investigated further in future work.

Lastly, it would be interesting to use a formal language to define function resolutions in contract evolution specifications. The current implementation supports the specification of function resolutions via one-line Java expressions, the adoption of a formal language would make the compatibility verification process more robust and enable the adapter to be implemented in programming languages other than Java.

Bibliography

[1] C. M. Aderaldo et al. “Benchmark requirements for microservices architecture research”. In: *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*. IEEE. 2017, pp. 8–13 (cit. on p. 47).

[2] C. Anderson. “Docker [software engineering]”. In: *Ieee Software* 32.3 (2015), pp. 102–c3 (cit. on p. 8).

[3] I. an API-Centric and S. Malvik. “Mastering Azure API Management”. In: () (cit. on p. 22).

[4] T. K. Authors. *Kubectl*. https://kubernetes.io/docs/reference/kubectl/ (cit. on p. 44).

[5] L. Bassett. *Introduction to JavaScript object notation: a to-the-point guide to JSON*. "O’Reilly Media, Inc.", 2015 (cit. on p. 13).

[6] B. Benatallah et al. “Developing adapters for web services integration”. In: *International Conference on Advanced Information Systems Engineering*. Springer. 2005, pp. 415–429 (cit. on pp. 20, 21).

[7] J. Bosch. “Architecture challenges for software ecosystems”. In: *Proceedings of the fourth European conference on software architecture: companion volume*. 2010, pp. 93–95 (cit. on p. 1).

[8] J. Bosch. “Software architecture: The next step”. In: *European Workshop on Software Architecture*. Springer. 2004, pp. 194–199 (cit. on p. 1).

[9] H. P. Breivold, I. Crnkovic, and M. Larsson. “A systematic review of software architecture evolution research”. In: *Information and Software Technology* 54.1 (2012), pp. 16–40 (cit. on p. 1).

[10] B. Burns et al. “Borg, omega, and kubernetes”. In: *Communications of the ACM* 59.5 (2016), pp. 50–57 (cit. on p. 8).

[11] J. R. Corbin. *The art of distributed applications: programming techniques for remote procedure calls*. Springer Science & Business Media, 2012 (cit. on p. 7).

[12] J. Deacon. "Model-view-controller (mvc) architecture". In: *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf* (2009) (cit. on pp. 1, 6).

[13] G. Developers. *Protocol Buffers*. http://code.google.com/apis/protocolbuffers/ (cit. on p. 13).

[14] O. Developers. *Java RMI*. https://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmiTOC.html (cit. on p. 14).

[15] N. Dragoni et al. "Microservices: yesterday, today, and tomorrow". In: *Present and ulterior software engineering* (2017), pp. 195–216 (cit. on p. 5).

[16] C. Ebert et al. "DevOps". In: *Ieee Software* 33.3 (2016), pp. 94–100 (cit. on p. 9).

[17] E. Evans and E. J. Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004 (cit. on p. 7).

[18] T. Freeman and F. Pfenning. "Refinement types for ML". In: *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 1991, pp. 268–277 (cit. on p. 12).

[19] A. Gómez-Pérez and O. Corcho. "Ontology languages for the semantic web". In: *IEEE Intelligent systems* 17.1 (2002), pp. 54–60 (cit. on p. 32).

[20] D. Gourley et al. *HTTP: the definitive guide*. "O'Reilly Media, Inc.", 2002 (cit. on p. 7).

[21] M. J. Hadley. *Web application description language (WADL)*. 2006 (cit. on p. 11).

[22] R. Hat. *Deployment Strategies*. https://docs.openshift.com/aro/3/dev_guide/deployments/deployment_strategies.html (cit. on p. 21).

[23] M. Howard. *Helm – What It Can Do and Where Is It Going?* 2022. DOI: 10.48550/ARXIV.2206.07093. URL: https://arxiv.org/abs/2206.07093 (cit. on pp. 37, 49).

[24] A. S. Inc. *Artillery*. https://www.artillery.io/ (cit. on p. 49).

[25] A. S. Inc. *Artillery Operator*. https://github.com/artilleryio/artillery-operator (cit. on p. 49).

[26] A. S. Inc. *Artillery Operator*. https://github.com/artilleryio/artillery-operator (cit. on p. 51).

[27] P. Kaminski, H. Müller, and M. Litoiu. "A design for adaptive web service evolution". In: *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*. 2006, pp. 86–92 (cit. on p. 17).

[28] M. Kleppmann. *Schema evolution in avro, protocol buffers and thrift*. 2013 (cit. on p. 15).

[29] J. Kreps, N. Narkhede, J. Rao, et al. "Kafka: A distributed messaging system for log processing". In: *Proceedings of the NetDB*. Vol. 11. 2011, pp. 1–7 (cit. on p. 7).

[30] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[31] J. L. Martin Fowler. *Microservices*. http://martinfowler.com/articles/microservices.html. 2014 (cit. on pp. 1, 2, 5, 7).

[32] R. Meng and C. He. "A comparison of approaches to web service evolution". In: *2013 International Conference on Computer Sciences and Applications*. IEEE. 2013, pp. 138–141 (cit. on p. 14).

[33] F. Menge. "Enterprise service bus". In: *Free and open source software conference*. Vol. 2. 2007, pp. 1–6 (cit. on p. 7).

[34] M. Murata et al. "Taxonomy of XML schema languages using formal language theory". In: *ACM Transactions on Internet Technology (TOIT)* 5.4 (2005), pp. 660–704 (cit. on p. 12).

[35] M. P. Papazoglou and W.-J. Van Den Heuvel. "Service oriented architectures: approaches, technologies and research issues". In: *The VLDB journal* 16.3 (2007), pp. 389–415 (cit. on p. 2).

[36] D. E. Perry and A. L. Wolf. "Foundations for the study of software architecture". In: *ACM SIGSOFT Software engineering notes* 17.4 (1992), pp. 40–52 (cit. on p. 1).

[37] F. Pezoa et al. "Foundations of JSON Schema". In: *Proceedings of the 25th International Conference on World Wide Web*. WWW '16. Montréal, Québec, Canada: International World Wide Web Conferences Steering Committee, 2016, 263–273. ISBN: 9781450341431. DOI: 10.1145/2872427.2883029. URL: https://doi.org/10.1145/2872427.2883029 (cit. on p. 35).

[38] R. Ratovsky. *OpenAPI*. https://swagger.io/specification/ (cit. on p. 11).

[39] W. Reese. "Nginx: The High-Performance Web Server and Reverse Proxy". In: *Linux J.* 2008.173 (2008). ISSN: 1075-3583 (cit. on p. 31).

[40] roboll. *Helmfile*. https://github.com/helmfile/helmfile (cit. on p. 49).

[41] Á. A. Santos et al. "Distributed Live Programs as Distributed Live Data". MA thesis. DI-FCT NOVA - Universidade NOVA de Lisboa, 2020 (cit. on p. 18).

[42] G. Sayfan. *Mastering kubernetes*. Packt Publishing Ltd, 2017 (cit. on pp. 8, 48).

[43] J. C. Seco et al. "Robust Contract Evolution in a TypeSafe MicroServices Architecture". In: *Art Sci. Eng. Program.* 4.3 (2020), p. 10. DOI: 10.22152/programming-journal.org/2020/4/10. URL: https://doi.org/10.22152/programming-journal.org/2020/4/10 (cit. on pp. 18, 19).

[44] M. Slee, A. Agarwal, and M. Kwiatkowski. "Thrift: Scalable cross-language services implementation". In: *Facebook white paper* 5.8 (2007), p. 127 (cit. on p. 13).

[45] J. F. Smart. *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. "O'Reilly Media, Inc.", 2011 (cit. on p. 34).

[46] F. Soppelsa and C. Kaewkasi. *Native docker clustering with swarm*. Packt Publishing Ltd, 2016 (cit. on p. 8).

[47] A. Sumaray and S. K. Makki. "A Comparison of Data Serialization Formats for Optimal Efficiency on a Mobile Platform". In: *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*. ICUIMC '12. Kuala Lumpur, Malaysia: Association for Computing Machinery, 2012. ISBN: 9781450311724. DOI: 10.1145/2184751.2184810. URL: https://doi.org/10.1145/2184751.2184810 (cit. on p. 35).

[48] swagger.io. *Swagger Parser*. https://github.com/swagger-api/swagger-parser (cit. on p. 38).

[49] J. Turnbull. *Monitoring with Prometheus*. Turnbull Press, 2018 (cit. on p. 48).

[50] V. T. Vasconcelos et al. "HeadREST: A specification language for RESTful APIs". In: *Models, Languages, and Tools for Concurrent and Distributed Programming*. Springer, 2019, pp. 428–434 (cit. on p. 12).

[51] D. Vohra. "Apache avro". In: *Practical Hadoop Ecosystem*. Springer, 2016, pp. 303–323 (cit. on p. 13).

[52] S. Wang, I. Keivanloo, and Y. Zou. "How do developers react to restful api evolution?" In: *International Conference on Service-Oriented Computing*. Springer. 2014, pp. 245–259 (cit. on p. 26).

[53] O. Zimmermann. "Microservices tenets". In: *Computer Science-Research and Development* 32.3 (2017), pp. 301–310 (cit. on p. 5).