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 TRANSACTION-LESS SOLUTION

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon

Final: *October 16, 2022*

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

SAFE API EVOLUTION IN A MICROSERVICE ARCHITECTURE WITH A PLUGGABLE AND TRANSACTION-LESS SOLUTION

RODRIGO JORGE RIBEIRO

Bachelor Degree in Computer Science

Adviser: Carla Ferreira
Associate Professor, NOVA University Lisbon

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

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
Final: *October 16, 2022*

Abstract

In contrast to traditional 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, however there is a scarcity of mechanisms for safely evolving and discontinuing functionalities of services.

Changing the definition of an element in a regular program can be accomplished quickly with the aid developer tools, such as IDE refactoring toolkits. In distributed systems there is a lack of tools comparable to those used in centralized systems, developers are left with the burden of manually tracking down and resolving problems caused by uncontrolled updates.

Whereas traditional approaches ensure that microservices are behaving properly by validating their behavior through empirical tests, our solution seeks to supplement the conventional approach by providing mechanisms that support the validation of deployment operations, and the evolution of microservice interfaces.

We present a microservice management system that verifies the safety of modifications to service interfaces and allows for the evolution of service contracts by using 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 either new or existing services.

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

Resumo

Em contraste com sistemas tradicionais, as arquiteturas de microsserviços permitem grande escalabilidade, disponibilidade e entrega separando os elementos de um grande projeto em entidades independentes ligadas através de uma rede (serviços), porém não oferecem mecanismos de segurança para evolução e descontinuação de funcionalidades fornecidas pelos serviços.

Alterar a definição de um elemento em um programa convencional pode ser feito rapidamente com o auxílio de ferramentas automatizadas. Os sistemas distribuídos não possuem ferramentas comparáveis às utilizadas em sistemas centralizados. Na ausência de ferramentas que amenizem este problema, os desenvolvedores tem que rastrear e resolver manualmente os problemas causados por atualizações não controladas.

Enquanto que as abordagens tradicionais garantem que os microsserviços obcedem a sua especificação, atráves da validação do seu comportamento por meio de testes empíricos, a nossa proposta procura complementar a abordagem convencional fornecendo mecanismos que suportam a validação das operações de deployment e a evolução das interfaces de microsserviços.

Apresentamos um sistema de administração de microsserviços que verifica a segurança de modificações nas interfaces de serviços e permite a evolução de interfaces compatíveis por meio de componentes proxy gerados em tempo de execução que adaptam dinamicamente os dados trocados entre serviços ao formato esperado pelo código de serviço estático, aliviando assim o developer da necessidade de adaptar manualmente serviços novos ou existentes.

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 and 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 composing the elements of a system. Each element of system typically falls into one of three overlapping categories: data elements, processing elements, and connecting elements. [25]. The data elements contain the state of system (eg. databases, variables, files); The processing elements transform the data elements (eg. procedures, services, sub-systems); The connecting elements serve as the bridge that permits different processing elements to collaborate (e.g. procedure calls, shared data, network messages). Software architectural models are frequently distinguished by the connecting elements they employ. Connecting elements have a great influence on non-functional criteria. This thesis aims to enhance the connecting elements used in microservice architectures in order to fulfill stricter extensibility and robustness requirements.

The field of software architecture traces back to 1960s, but it wasn't until 1992 that Perry and Wolf [25] created a firm basis on the subject. Software architecture has been extensively studied over the last several decades, and as a result, software engineers have developed novel methods for composing systems that offer broad functionality and fit a diverse set of criteria. Bosch's work [8, 7] overviews software architectural research.

A typical example of an architectural design is the Model-View-controller (MVC) [11, 20]. 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. This triple layer division improves software maintainability by separating distinct development concerns, however this architecture 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) [24] 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, where the exchanged messages adhere to standardized contracts. Microservices [20] are a modern iteration of the SOA model, that reduce the excessive and imposing layers introduced by earlier generations of the SOA model.

Some advantages of service orientation over monoliths are as follows:

- Functional independence - Each service is operationally independent, services only communicate through their published interfaces.
- Component scalability - When a single component becomes overloaded, additional instances of the component can be deployed separately.
- Separate development - Separate teams may work on different modules of the system simultaneously if they agree on the interfaces of modules ahead of time.
- Flexibility - Each service 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 it 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 application, interactions between different system components are conducted through function calls. In a distributed system, however, each individual

component has to communicate with other components across the network, without the same guarantees.

In a regular program, refactoring a function definition can be done quickly with the aid of an IDE. Distributed systems do not have equivalent tools to detect mismatches between endpoints and their consumers. Developers are left with the burden of manually tracking down and refactoring the system's dependencies across all consumers and producers; there is a lack of tools that alleviate this problem.

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 interface or contract modifications; in the context of iterative approaches to software development, such as agile, 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 use of contract management tools, and a common contract specification is necessary to ensure the security of service-based architectures.

- **The safety of a deployment operation is unknown:** Established platforms like Kubernetes can't guarantee the security of deployments because they rely on service level meta-information (like version IDs) to manage them. A new model for such environments is required to ensure the safety of deployment tasks, one that uses generic meta-information from previously deployed services, uses expanded type-based contracts, and employs a compatibility relation on service contracts.

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

1.2 Contributions

We present a microservice management system that aims to solve the problems stated above.

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 schema registry that offers a pre-flight safety check procedure that verifies whether two service contracts are compatible before deployment. The compatibility mechanism is akin to typechecking procedures in compiled languages and is applied upon a pair of service contracts, which are typically two distinct versions of the same service. The aforementioned mechanism preserves a distributed system's correct behavior by rejecting additions or modifications that would threaten it and by informing the programmer of all the repercussions of the deployment, such as consumer services that are outdated.

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 intercepting and evaluating all the outgoing and incoming TCP requests to determine whether they should be left unchanged or how they should be transformed using an adaptation protocol that extends the compatibility mechanism discussed above. We allow for the addition, deletion, renaming, and type migration of data fields in a producer module without disrupting or upgrading consumer services. More complex changes in data fields, such as altering the format of a date, will be permitted through the use of user-defined adaption functions. We also support the modifications on the signatures of WebAPIs, such as changing a path parameter in a HTTP endpoint to a query parameter. The contract translation specification between two compatible contracts can be audited and modified by a programmer before it is employed by an adapter.

Our work will make the following contributions, which will be discussed in depth in Chapter 4:

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

1.3 Document Scructure

The following sections of the document start with a review of the main the concepts, techniques and applications behind our approach (Chapter 2). Sequent sections, first introduce a summary of the key related work (Chapter 3), then we present the design of our system and discuss alternatives (Chapter 4), in (Chapter 5) we present the implementation and conclude with results in (Chapter 6).

2 Background

The technology underlying the development and deployment of microservices-based applications will be described in this section:

2.1 Microservices

Microservices [20, 35, 14] are an architectural style in which software is developed using self-contained components that communicate with one another via standardized interfaces and lightweight mechanisms. These services segregate fine-grained business functionalities and can be independently deployed, scaled, and tested by automated mechanisms. There is virtually no centralized management, and each service may be written in a different programming language and employ a different data storage technology.

To understand the microservice style it's useful to compare it to the monolithic style: A monolithic application is built often using the Model-View-Controller (MVC) pattern [11], 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.

When large monolithic applications must scale while maintaining a high level of availability, they become a source of frustration: Scaling the server-side application involves scaling all the application functionalities, rather than the functionalities that require greater resources; Any small change made to the server-side application, involves building and re-deploying the entire monolith;

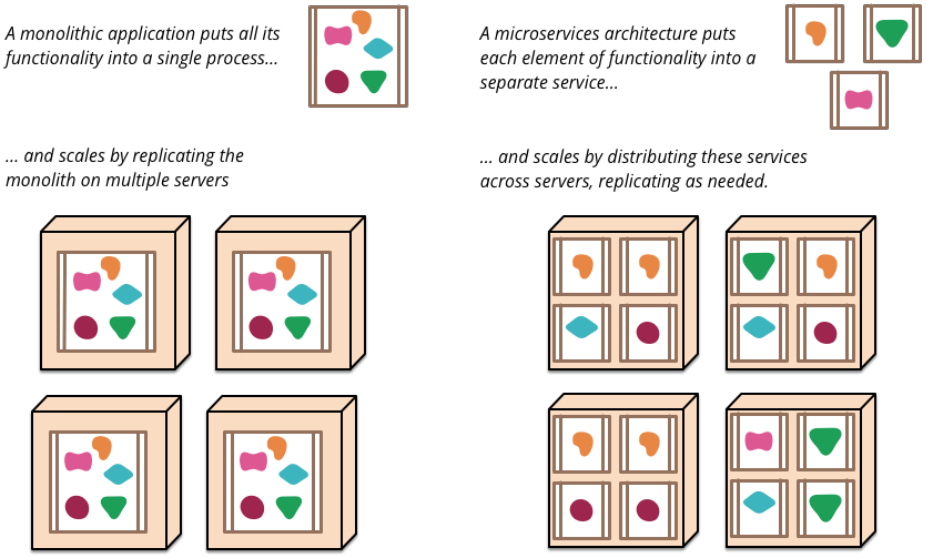

Figure 2.1: Monoliths and Microservices [20]

The requirements of system scalability, high availability, and continuous delivery are addressed in different microservice architectural characteristics.

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. Building and maintaining applications that employ distributed transactions is notoriously difficult; as a result, microservice architectures advocate for transaction-less service coordination while acknowledging that consistency will be only eventual and that divergence must be addressed with compensating operations.

Inter-process Communication Strategies 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) [22] 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, it was difficult to solve problems in production environments, and the tooling was complex and expensive.

The reverse approach has been adopted with microservices, where services own their domain-centric logic "smart endpoints"and transport messages via "dumb pipes". The majority of communication between microservices is done via request/response-based communication or event-driven messaging. Because these two methods have such dissimilar properties, it's important to weigh their strengths, and the scenarios that call for each.

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 and RPC protocols are the most widely adopted protocols that follow this approach.

Event-driven messaging communication protocols are suited for asynchronous settings, where the 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.

TODO: include reference to distributed transaction patterns, and the channels that are used in each approach https://developers.redhat.com/articles/2021/09/21/distributed-transaction-patterns-microservices-comparedthe$_{d}ual_{w}rite_{p}roblem$

Evolutionary Design When decomposing a software system into components, we must decide how to divide the pieces. The concept of independent replacement and upgradeability are critical properties when designing a component. The disadvantage of incorporating components into services is that we must be concerned about changes to one service breaking its consumers. The conventional approach is to resolve this issue through the use of versioning. In the context of microservices, versioning entails the maintenance of multiple deployments of the same service in distinct versions. We can minimize the use of versioning by designing services in such a way that they are as tolerant as possible to supplier changes, utilizing techniques such as Domain-Driven Design (DDD) [15]. 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.

2.2 Microservice Lifecycle Management

To run microservices in the cloud, we need two essential ingredients a method for packaging and isolating services, and a management system capable of provisioning physical hardware to support them.

The isolation of services is accomplished through resource virtualization in one of two ways: containers or virtual machines (VM).

Containers provide the most efficient approach because unlike VMs, containers share the host system's kernel with other containers. VMs virtualize an entire machine down to the hardware layers, while containers only virtualize software layers above the operating system level.

Docker [2] is the most popular container technology. It is built on top of the following technologies: Kernel namespaces, Cgroups, Copy-on-write File system.

- 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 [31] and Kubernetes [28].

Kubernetes [28] is an open-source container orchestration system that evolved from Google's Borg and Omega projects [10]. Kubernetes is an ambitious platform. 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, haves 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 is said to be more scalable.

2.3 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. [29]. The proposed approach to microservice contract evolution [29] 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 2.2 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"

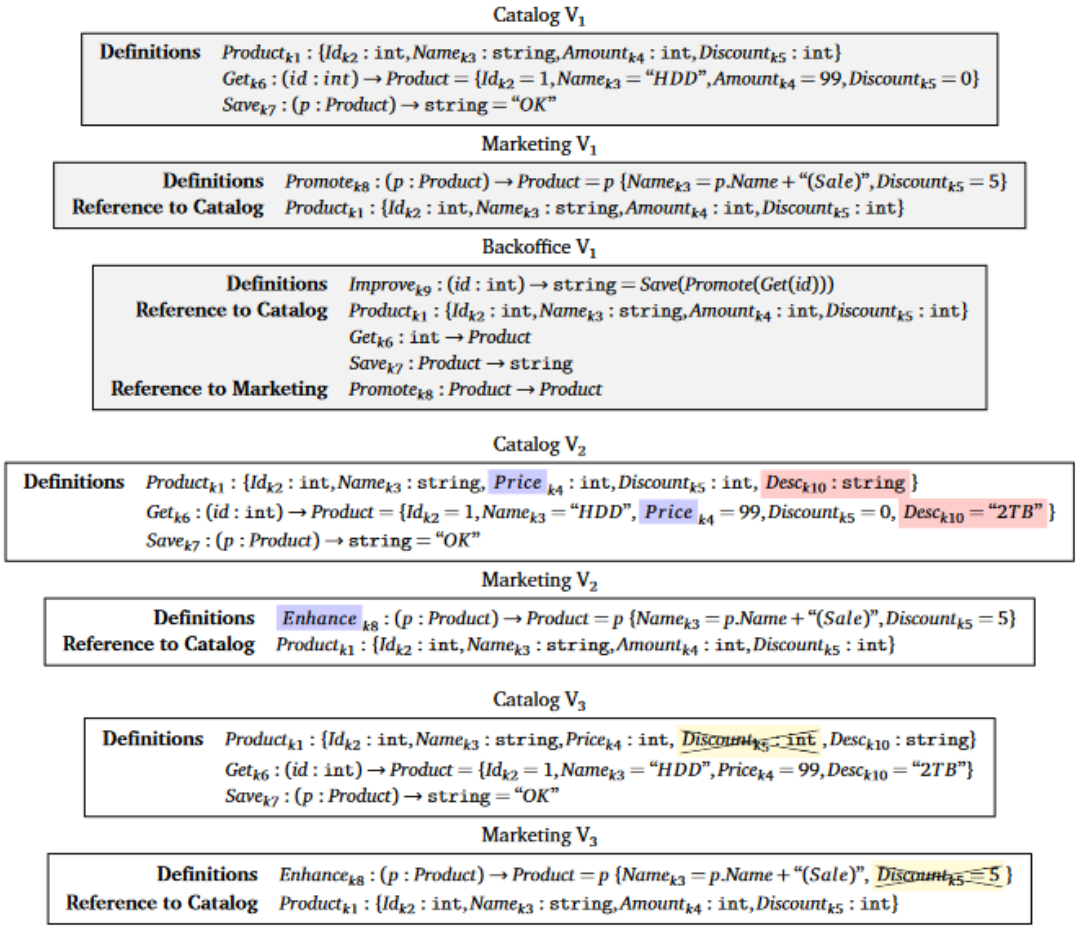

Figure 2.2: Evolution of modules Catalog, Marketing, Backoffice [29]

is k1. These keys enable elements to be identified even after they have been renamed or reinstated.

Initially, we have a system composed of a collection of services, each of which hosts an instance of one of the deployed modules shown in grey in Figure 2.2. The reference definitions indicate their interactions.

In a second phase, we develop version 2 of the Catalog and Marketing modules independently. In module Catalog, we rename the element "Amount" to "Price" and add the attribute "Desc", while in module Marketing, we rename "Promote" to "Enhance". The module BackOffice has not been modified at this point and will become out of sync with the other modules. In Figure 2.2, the newly added and modified elements are highlighted in red and blue, respectively.

In a third phase, Catalog version 3 is created, with the attribute "Discount" of the element "Product" removed. (Figure 2.2). This version cannot be deployed in conjunction with version 2 of Marketing, as the attribute is used by the function "Enhance".

We can deploy both modules together if we modify the definition of ”Enhance” to remove the use of the attribute ”Discount”, resulting in version 3 of Marketing. Additionally, we could deploy version 3 of Marketing first and then version 3 of Catalog, but not vice versa. Besides this, in the reference to catalog we are permitted to retain the attribute ”Discount” in the definition of the type ”Product”. As long as the attribute is not used, correctness is guaranteed. In Figure 2.2, the elements that have been removed are highlighted in yellow.

As illustrated by the multiple versions of the modules, the proposed system is capable of overcoming differences in contract definitions by utilizing a communication protocol that adapts automatically to the following types of changes:

- **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 "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, such as HTTP, 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 the presented 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.

Experimental data [29] collected over a five-year development period on the evolution of three large software factories each containing over 1000 modules (a raw dataset containing 8889 production deployments with a total of 23986 signature changes) indicates that this approach would be effective in an average of 57% of deployments.

2.4 DevOps

DevOps 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 typical build and release pipeline is shown in Figure 2.3.

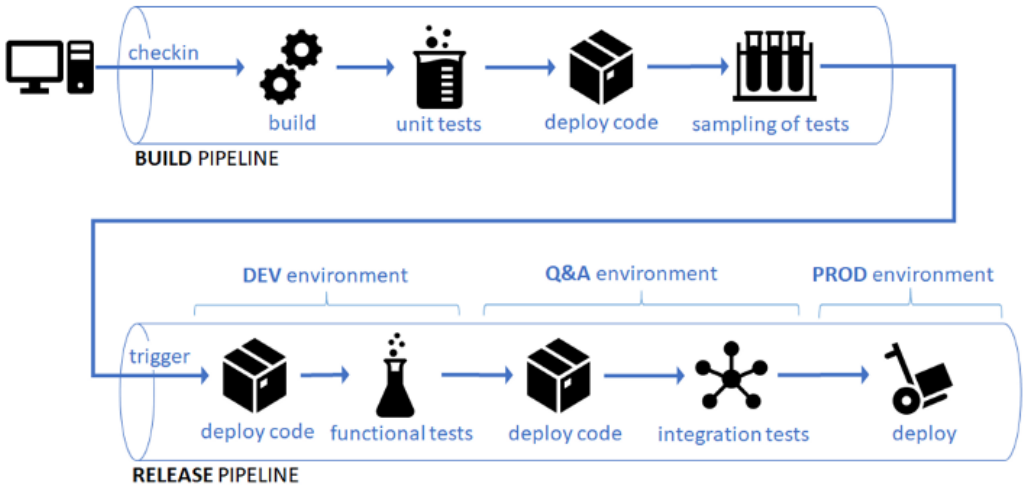

Figure 2.3: Pipeline Example

3 Related Work

As mentioned in Section 1, this work will encompass the development of mechanisms for evaluating and managing the soundness of updates in RESTfull [16] service contracts. We now summarize the most important works of literature relating to the evolution of microservices and their APIs.

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) are domain-specific languages used to describe web service contracts in a standardized structure; also sometimes 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) [26] is the most widely adopted Restful API Description Language; it defines a standard language-agnostic interface that allows both computers and humans to comprehend the capabilities of a service. OpenApi allows a detailed description of the syntactic aspects of the data transferred in REST interactions, however it ignores important 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 to relate the output against the input.

For instance, OpenAPI does not allow developers to indicate that the type of representation conveyed in the response to a GET operation is dependent on the value of a query parameter.

Vasconcelos et al. [33] 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 [17] 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.

Our solution will require the use on WADL for defining compatibility relations between service contracts, as well for the implementation of the adaptation protocol, which changes messages in accordance to an earlier contract.

The OpenAPI specification is the most widely adopted WADL for RESTfull services and already supports refinement types, instead of designing yet another WADL, we aim to extend the OpenAPI specification to incorporate support for Pre- and Post-conditions and other needs of our approach.

3.2 Platform Independent Schema Representations

Schema representations are need when delivering serialized messages across the network or storing data with durability. For serialization, each language usually provides a corresponding library, such as Java serialization. In the setting of microservices, serialization

libraries supplied by programming languages are typically not used to encode messages between services, because each service may be written in a different language. As a result, data consumers will be unable to comprehend data producers.

Cross-language serialization libraries, such as JSON, can solve this problem. However, formats such as JSON lack a strictly defined structure, making data consumption more difficult due to poor type-safety guarantees, and the ability for fields to be unilaterally added or withdrawn at any moment without the consumers' knowledge. What's missing is a schema for data exchange between producers and consumers, akin to an API contract.

The benefit of having a schema is that it explicitly defines the data's structure, type, and meaning. There have been a few cross-language frameworks that require the data structure to be properly described via schemas. XML, Avro, Thrift, and Protocol Buffers [34, 12, 30] are among them.

Because the schema of the messages exchanged between services can already be described using the OpenAPI description language, solutions that describe the schema of messages such as Avro, ProtoBuf, Thrift, and XML will be redundant in our solution.

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 a RESTful API and persistent storage for schema definitions, where each schema is given a unique ID.

Schema registries are commonly used in situations where data consumers need to know the structure of the data written by producers at runtime. A producer could send its schema to consumers along with the response to a request; however, this is usually a bad idea because it would result in duplicating functionality across all services, making the system more difficult to maintain.

One framework that makes use of this system is Avro [34], a data serialization framework developed within Apache's Hadoop project. Avro requires a schema registry because the serialized byte sequence of each record does not include field metadata such as the field name or a tag, but only the field value. This allows for a more compact serialization method, but it requires data consumers to understand the structure of the data written by producers. All field values are appended back to back, in the same order as they appear in the schema as seen in figure 3.1.

The deserializer knows which bytes belong to which field by comparing both the consumer and producer schemas and by matching fields by their names.

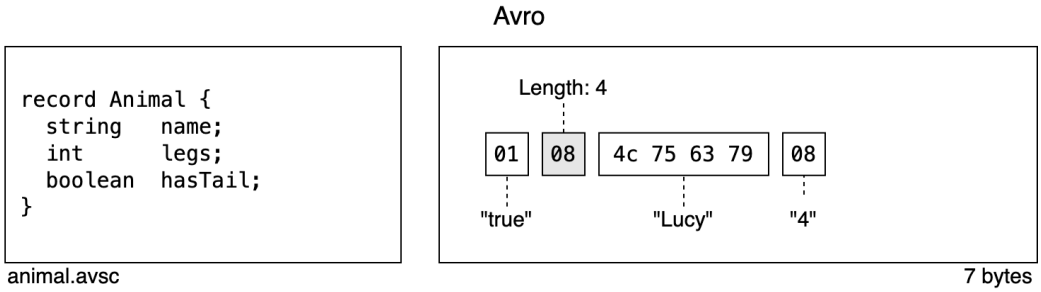

Figure 3.1: A Avro schema and its associated serialized record

Another system that makes use of schema registries, is the Java RMI [13], 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. The RMI registry is only used for locating the first object that a client needs, each reply from the registry provides support for finding other objects.

Our approach will need the use of a service registry, which will enhance the capabilities of a schema registry by storing not just schemas but also service-to-service dependencies.

3.4 Service Evolution Approaches

Due to various factors services can not keep their stability permanently, evolution occurs throughout their entire life cycle.

Controlling and supporting the evolution of services is a significant technological research endeavor. Meng and He propose five standards for evaluating existing methods for evolving Web services; these standards, and some methods are also applicable in a micro-service setting, as microservices can be viewed as web services:

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 evolve changes to 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.

3.4.1 Hand-off Period

The conventional approach for evolving microservices is to keep the various deployments of the different versions of a service implementation available with a hand-off period, and to explicitly redirect requests to the supported 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 its possible to support backwards compatibility without distinct deployments, by including in the new service contract the endpoints of the prior contracts that are still in use. When a breaking contract change occurs, rather than supporting the change by modifying the impacted endpoint implementation, we preserve the old endpoint and implement the new contract version in a completely separate endpoint. 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. 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 [19]. The following are the common limitations of the aforementioned protocols:

- To ensure backwards compatibility, newly added fields must be optional.

- The only fields that can be removed to maintain forward compatibility are optional fields.
- Its 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.

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 new producer implementation. Schemas that are used both in request and response messages most support forward and backward compatibility.

The need for optional fields in the above cases is due to a lack of information on the message receiver, typically this information is supplied with the use of default values.

All fields that are optional exclusively because of resolution rules should be treated as required when a consumer and producer agree on the schema version, and the default value should not be utilized, since there is no absence of information in this case. To simply put it, the system should enforce required fields when the consumer and producer agree on the schema, and only use 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.

The aforementioned frameworks include built-in support for event-driven and RPC communication methods, however, if a HTTP communication approach is adopted, these frameworks 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 record schemas 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, we believe 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 unsupervised clients while preserving strict backwards compatibility. This approach decomposes long update/rollback transactions into smaller, independent transactions.

This technique is discussed in depth in the work [18]. The key concepts of the approach are presented below.

When a new service API is published, the old API is not decommissioned; instead, it is made available in a different namespace. In a RESTful system, this often entails supporting operations in different versions; Endpoints belonging to the API in v1.0 can be made accessible via the path `http://example.system/v1`, whereas endpoints belonging to the API in v2.0 are accessible via the path `http://example.system/v2`.

The previous API implementations are replaced by adapters that redirect all calls to the next API version implementation and also translate data structures as necessary.

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

- Duplicate all modified endpoints in service contract into a different namespace.
- Implement an adapter that translates the endpoints in version vi->vi+1.

The same web service that supports the revised service endpoints also supports the older endpoints that require adapters. It is not necessary to deploy the web service in several versions.

When a consumer is using version v1, and the producer is using version v4, the endpoints that provide the service in version v1 use the adapters v1 →v2, v2 →v3, and v3 →v4 to translate the operation before forwarding it to the endpoint in the current version. The service's backwards compatibility is ensured by the composition of adapters in a chain, as seen in figure 3.2.

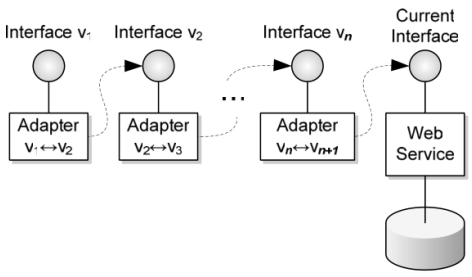

Figure 3.2: Chain of Adapters structure after n versions have been published

3.4.5 Lazy Proxy Adapter

In this strategy the knowledge of a microservice's interfaces is used to create a lightweight proxy capable of dynamically adapting the messages exchanged between services to match them with the static service code. The adapter 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 [29, 27].

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.

With this approach, changes to definitions in a module do not have an immediate effect on the whole system, because they do not require the redeployment of consumers. For instance, if two services agree on a definition for a given endpoint, their existing implementations will continue to function even if that endpoint definition evolves over time.

3.5 Service Integration Adapters

There has been substantial investigation on the adaptability of services in the context of service-oriented architectures. Although SOA [24] and Microservices [20] 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 of differences 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 of differences 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 of mismatch 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 message failures.

The "Message Merge Mismatch"can also be solved via an adapter, but this would require the adapter to become **stateful**, since it would need store all messages before they could merged and sent through the new endpoint.

The two first types of mismatches can be handled in the context of microservice evolution without the use of adapters. Essentially, the old endpoints are kept operational but marked as deprecated until there are no more consumers using them, and the new endpoints are provided in parallel.

The latter form of mismatch "Differences at the Operation Level"has an impact on microservice evolution only 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.

3.6 Deployment Strategies

Deployment strategies 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 our 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. The service registry will aid in the application of these strategies, since it is responsible for managing service deployments and un-deployments.

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 an 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.

Our solution will use the rolling deployment strategy because it provides the best availability guarantees and is natively supported by Kubernetes, the microservice orchestration tool that has been adopted.

3.7 API Management Tools

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

Microsoft Azure’s API Management platform (AAMP) is a tool that provides some of the above functionalities in a manner that is similar to that of our solution. In this platform, programmers are allowed to manually define transformations that should be performed on all responses from a microservice []. The primary distinction is that Azure API Management transformations are intended to be specified by the programmer in policy statements, whereas our transformations can be applied automatically without the intervention of developers.

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.

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, like our adapter proxy, 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 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;
- Transforming requests and responses as specified in policy statements;
- Caching responses to improve response latency and minimize the load on backend services;
- Emitting traces for monitoring, reporting, and troubleshooting;
- Collecting API usage metrics

We aim to include some of the aforementioned features in our service registry implementation, particularly the collection of usage metrics.

4

Design

In this chapter, we present the design of our system and discuss alternatives to the decisions we took in our implementation. A modular approach was adopted to construct the system; each module is entirely independent, making it easy to interchange alternative approaches. We dedicate the first half of this chapter to define the goals that guided the design of system, the remainder of the chapter will be used to discuss each design aspect in turn.

4.1 Discussion

Recall that the problem we are trying to solve 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 goals that informed our decisions, are outlined below by their priority:

- Supporting all types of contract changes
- Integration with existing tools and workflows
- Minimize conception and maintenance effort
- Automatic validation of the safety of deployment operations
- Consumers and producers are unaware to the presence of adapters
- Transactionless deployment and upgrade of service contracts
- No downtime when upgrading service contracts
- No overhead when services communicate using the same contract version

Four principles 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, we prioritized utility above performance because, in the common case, each service will communicate via up-to-date contracts and because the effort associated with adopting an approach will be proportional to the complexity and size of a contract and its evolution, whereas the performance overhead will remain mostly static under diverse contract changes, as it is largely attributable to communication costs. We prioritized flexibility and effectiveness above other principles because if an approach isn't applicable or effective under a scenario, then it will be necessary to adopt hybrid solutions, 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. Our 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 Most applications require a web presence and microservice systems that rely on choreography still require the HTTP request-response protocol to have a web presence, for this reason we chose to design our system around the orchestration pattern and the HTTP protocol.

Alternative Other event-driven and request-response protocols, such as RPC, have a simplified syntax, and there are already a number of tools that support the evolution of schemas 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. We use these interface specifications to clearly define and document the procedures of a service and its consumed resources and inter-service dependencies.

The definition of a service dependencies and requirements has three benefits: it allows a deployment to be validated and blocked if the corresponding requirements are not met within the system, it allows for the automatic removal of un-utilized adapters without developer intervention, and it allows developers to clearly see the impact of a contract change on the entire system, as well as the effort required to support the change.

Because most API definition frameworks do not allow the explicit declaration of dependencies and requirements of a service, contracts are represented in two distinct manifest files, one defines the procedures of a service, and the other dictates its consumed resources and requirements.

Approach The definition of a service dependencies can be done from the perspective of the whole service or the perspective of each service procedure. If we map the dependencies of each procedure individually, we gain the ability of detecting procedures that are un-used internally by other services. With this ability, we can avoid installing unnecessary adapters in cases where only unused procedures are updated; this case is expected to be very uncommon, and having unused procedures in a microservice is undesirable due to API bloat, so we believe it is not worth the documentation effort and that specifying service dependencies from the perspective of the service as a whole is sufficient. The only true benefit of explicitly specifying procedure dependencies would be the ability to view the impacted consumer procedures when a breaking change occurs, but this information can be obtained through other methods that do not require such a high documentation effort; one such solution is presented in the following paragraph.

Alternatives Services dependencies can be determined indirectly by automatically 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 or upgraded; it is only possible to validate the safety of the removal of services after they have been running over a long enough period, and even then there could be very rare calls between services that are not mapped by this method, these calls would make removals unsafe. Un-needed adapters would also have to be removed manually by developers or through an automated inactivity-based warning system. This approach could be instead employed with a different objective, discovering procedures that are rarely called, this information could be used to reduce the bloat of on API by eliminating or merging procedures, but this falls outside the scope of this thesis and has not been explored.

4.2.3 Compatibility Verification

Intuitively, compatibility verification determines whether all the edited elements in a producer service 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 both the producer contract and all consumer references to determine the safety of a deployment operation. Edited elements are

considered compatible if the information supplied by prior elements is sufficient to meet the edited elements parameters. Contract requirements are fulfilled if there are enough computational resources to accommodate the service and if all the service dependencies are available and reachable in the system.

Alternative Instead of determining if a new contract version is compatible with all consumer references, we evaluate only whether the new contract is compatible with the prior versions that are still in use. If a new contract is missing a capability that was present in a prior version, the developers will need to verify if the capability is not being consumed and mark it as obsolete in the evolution manifest for the contract to be considered safe. This approach has the advantage of streamlining the verification process by eliminating the need for all contracts to be accessible by machine code in a standardized fashion. The downside is that it introduces human error in the verification of un-utilized service capabilities.

4.2.4 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, we discuss how such tools could be implemented.

Approach Since all service contracts declare their dependencies on other services procedures, determining the impact of change in a procedure is as simple as looping over all active services contracts and verifying whether the changed procedure is being consumed. Then, using this information, a report indicating all affected services and other metadata may be created. We store service contracts in a version control system, with semantic versioning, where each major version represents incompatible contract changes and minor versions represent backward compatible contract changes. To discover the active services, we run a query against the container orchestration tool, which returns the deployed services. A tool like this may be easily created in a bash script and its results shown in an IDE. If kubernetes and git are utilized, the active services may be queried with the **kubectl** command line tool, and the service contracts can be fetched using the git repository urls and a standard naming convention for the service contracts.

Alternative The information about the active services and individual service contract can be translated into an ontology language and hosted in a centralized registry. The main advantage of this approach is that allows us to easily integrate the registry with a reasoning system capable of performing deductive inferences over the service's data, such as inferring the consumers of a service procedure given the procedure and service name or generating a graph structure that maps all participating services dependencies. This approach would allow us to obtain richer data and reduce the burden of the development

of sophisticated tools. The drawback is that the update of some information might require the use of distribute transactions to avoid incoherent data, for example renewing the services that are participating in the system.

4.2.5 Evolution Specification

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 one field **A->B** is indistinguishable, from inserting a new field **B** and removing another field of the same type **A**. The developer must declare which evolution occurred explicitly.

Approach The evolutions are specified in a manifest file, complete with its own syntax and rules, which outlines how to adapt calls between two versions of the 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 X on the previous contract is equivalent to element Y 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 Xn 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 Xn elements.

Renamed fields in contracts are supported with the use 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.

Alternative The evolution specification is defined in conjunction with the contract specification. Most web API specification languages already support default values and name aliases. With these two features it would be possible to support the following contract evolutions: renamed fields, addition of fields and removal of fields. For simple contract this alternative can seem appealing, but as the number of revisions and consumers in prior versions rises, this solution becomes impractical. This approach clutters the contract with information that is irrelevant to consumers, and makes the specification of complex contract changes more cumbersome.

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 adapter is installed in a separate container but on the same node as the 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".

Alternative A. The adapter code is embed 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.

Alternative B. The adapter is installed in the nodes of all the service consumers. This approach is problematic because a service upgrade would entail the re-deployment of all consumers. If one of the 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 adapters can be avoided with the use of mobile code that is provided by the service in a dedicated endpoint. This could be accomplished in Java with the use of network class loaders. We find this approach ineffective because it provides no benefits and requires more resources than alternative approaches since, instead of one adapter, we would need N adapters, one for each of the service consumers.

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 exchange of messages for one service.

Approach The adapter implementation is derived from a template implementation, that is completed with the substitution of template variables with code generated from

a single evolution specification manifest file. The advantage of this approach is that the implementation will be more performant than generic approaches.

Alternative The adapter haves a generic implementation and determines how to adapt messages by inspecting specialized headers in the message that indicate the message type, service and version and by consulting the corresponding evolution manifest. 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 other approach requires one adapter per service. The downside of this approach is that it requires modifying the service's implementation to add the message headers. The mentioned benefit is obviously lost if the adapter is embedded in every service's node.

4.2.8 Message Routing

Messages will need to reach either an application or an adapter, depending on the consumer expected contract, and the current producer contract. We can construct routing rules based on the incompatibility of two contracts, or the incompatibility of individual contract procedures.

Approach We route messages thorough the adapter if the expected contract of a consumer is not the current producer contract. When a producer service's contract is upgraded, all of its customers will begin diverting their calls to the adapter with the old version. The developers are notified, and must manually update all consumers if they wish for them to direct their calls to the producer service application instead of the adapter. The downside of this approach is that it requires immediate developer intervention if optimal performance and resource consumption is desired.

Alternative A. Each contract procedure includes the contract version in their endpoint path, in other words each service endpoint is versioned with the corresponding contract version. We direct all calls through a reverse proxy such as nginx, and we redirect requests based on custom routing rules that are injected on the reverse proxy. The routing rules are injected when a service with contract changes is deployed, in other words the injection is done in a DevOps pipeline stage after the service deployment. If no custom rules are submitted the expected behaviour is equal to the behaviour of the approach discussed above. This approach is counter-productive since it necessitates the installation of a reverse proxy, which use more resources and adds the same latency as the previous approach.

Alternative B. Instead of using reverse proxies, we use custom DNS rules to direct request. This approach is problematic because DNS rules can only be applied over the domain of requests, and we need to apply them based on the path of requests. We could create a DNS entry for each service procedure, where each entry contains a subdomain that indicates the service version and another subdomain that indicates the service procedure (eg. users.v3.getUser). This would achieve the desired behaviour, however it would force developers to change the service implementation and start referencing different services procedures by different domains, which could cause interference with other API tools that rely on service domains.

4.2.9 Adapter Management

The management of adapters should be automated similarly to the management of services. In this section, we discuss 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.

We propose to introduce the operations of contract compatibility verification, adapter generation, deployment, and removal as additional stages in typical continuous deployment pipelines. The generation of the adapter components could be preformed outside the deployment pipeline, this would increase the solution flexibility by allowing developers to manually extend adapters to cover unforeseen corner cases, however, it is preferable to generate the adapters in the pipeline so that adapter generation program has clearer view over the participating services in the system.

4.2.10 Supported Changes

"Message Split Mismatch"In order to keep the adapter implementation as lightweight as possible, it is preferable to maintain the old endpoint operational but marked as deprecated, and provide the new endpoints in conjunction with the old endpoint.

"Message Merge Mismatch"It is preferable to keep the old endpoints operational but marked as deprecated, and provided the new endpoint in conjunction with the old endpoints.

5 Implementation

In this chapter, we present the implementation of our system. We begin by offering an overview of system's architecture, before delving into the implementation of each module in depth.

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 architecture is split into two distinct areas: the runtime, and the pipeline. The runtime architecture comprises all essential components to the system liveness, whereas the pipeline architecture covers all components needed during the system's provisioning and development.

5.1.1 Runtime Architecture

The runtime environment is supported by Kubernetes, a simplified configuration with three microservices is depicted in Figure 5.1, in this example, we have one micro-service that has undergone three major API changes the accounts micro-service, and two micro-services that rely on it, the inventory micro-service that was updated to account for the changes, and the shipping micro-service which is two versions behind.

In Kubernetes a microservice is internally accessible through the Service abstraction; a Service defines a policy for accessing a logical set of Pods via a load balancer with a unique IP address that can be discovered thorough an environment variable, or an DNS name. The containers of a pod 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 we expose one port for each of the supported versions and name each port with the respective version.

We define one service for each major version that is still being consumed, as shown in Figure 5.1, 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 haves the same version as the one that it's expected in the static code.

When a microservice app is upgraded via a rolling update [1], 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, whereas in the traditional approach all consumers would need to be upgraded in tandem with consumed services in order to avoid downtime.

The only novel module introduced in the runtime environment is the proxy adapter; the forthcoming modules are either developer tools or are invoked on stages of the DevOps pipeline.

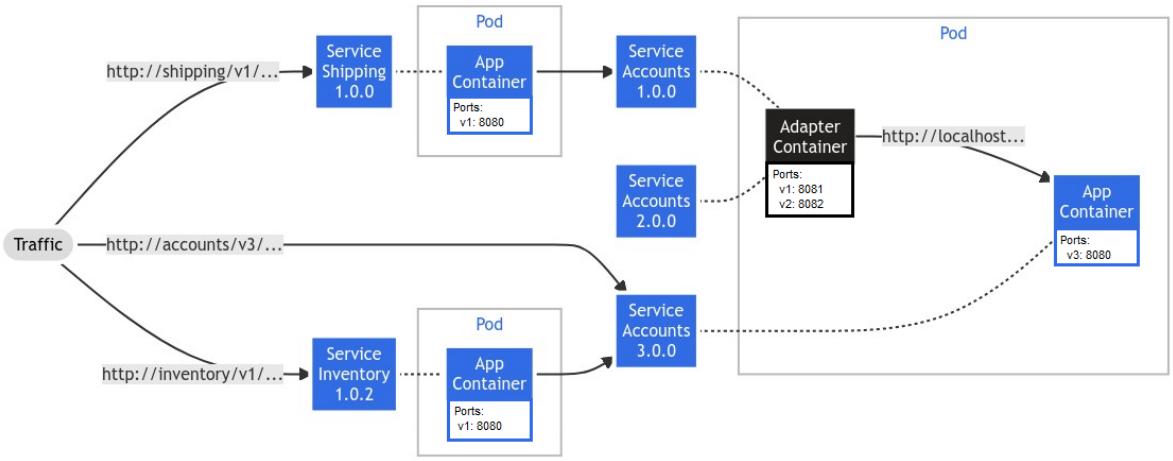

Figure 5.1: Example of a runtime configuration

5.1.2 DevOps Pipeline Architecture

A proof-of-concept deployment pipeline as developed, its architecture is depicted in Figure 5.2. The nodes on the left side of the graph represent the pipeline; development stages are represented in green, while operational stages are represented in red. The other nodes on the graph's reflect the resources used in each stage. External resources are depicted in blue, while the novel modules developed in this thesis are depicted in black. Two novel stages were introduced in the development phase: the writing of service contracts, and the specification of contract evolutions. These two stages are skipped if a service only underwent changes that did not affect its API or dependencies. We also introduced two novel stages in the development phase: the verification of the service compatibility with the rest of the system, and the generation of adapter component.

The pipeline was developed using the Jenkins 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 Jenkins suite toolkit. The modules that enable each novel stage will be discussed in detail in the following section.

[1] A rolling update gradually replaces old replicas with new replicas

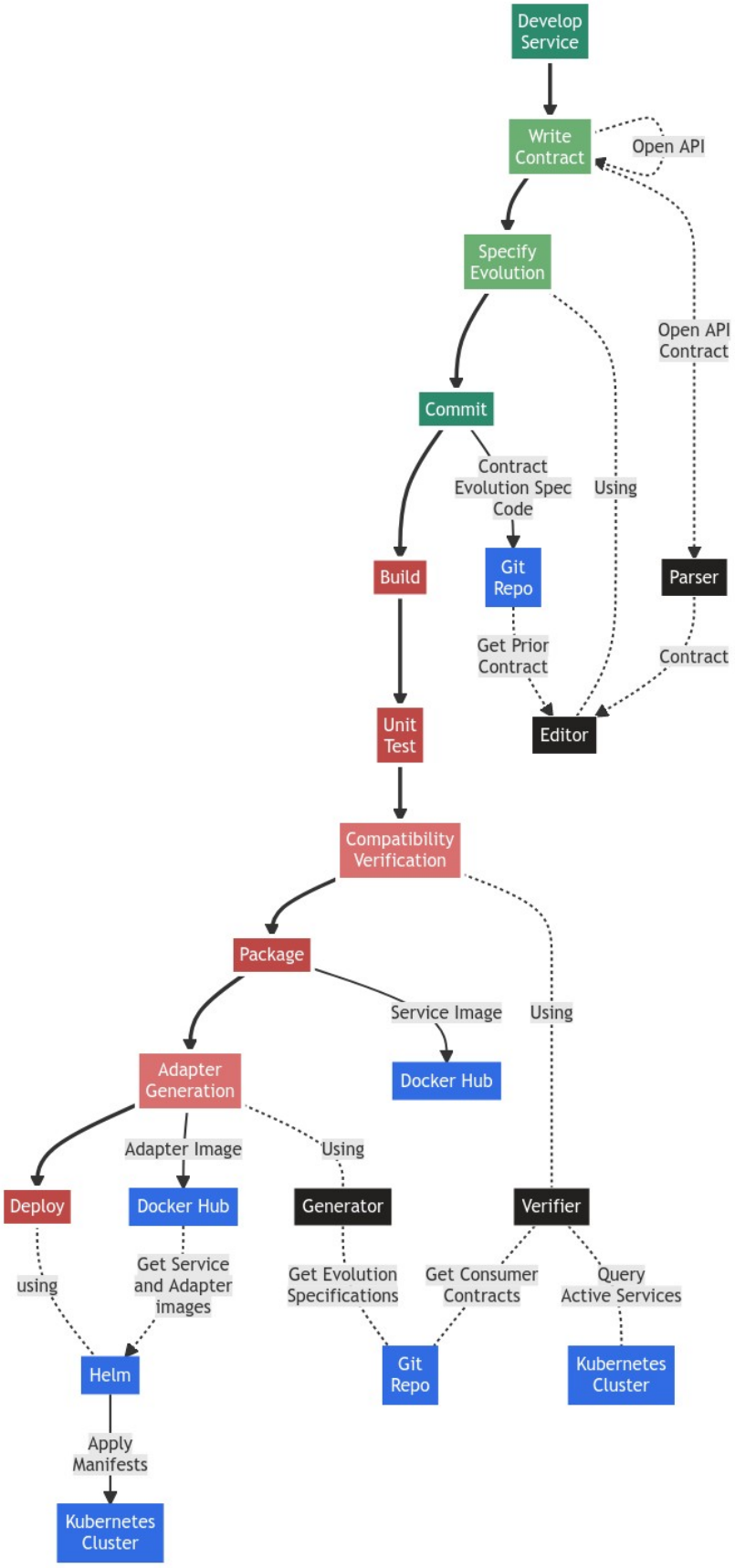

Figure 5.2: DevOps Pipeline

5.2 Contract Representation

The functionalities of web service's can be expressed with the use Web Api Description Languages (WADL). The OpenAPI specification is the most widely adopted WADL for HTTP services, instead of designing yet another WADL, we adopted the OpenAPI specification to support the needs of our implementation. 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 description document as input, then route incoming HTTP requests to example responses or dynamically generates examples.
- Data Validators: Verifies 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.

Designing a WADL tailored for this problem would be counterproductive because many teams already rely on the aforementioned 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, we chose the Json format, because of its simplicity and widespread adoption. Json is a human-readable data-interchange format, there are more performant formats that have binary representation, however such formats are usually not supported by front-end frameworks and for the purpose of this prototype we are not interested in measuring 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 [**serializationBenchmark**].

The schema of JSON records must also be described, the OpenAPI specification supports the description of records in conjunction with the signature of HTTP endpoints. Other Json schema specification languages, such as the Json-Schema project exist, but using languages other than OpenApi would necessitate decoupling the specification of schemas from the specification of http endpoints, which 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; while property formats are free-form, however there are established conventions for the most common formats.

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; while property formats are free-form, there are established conventions for the most common types.

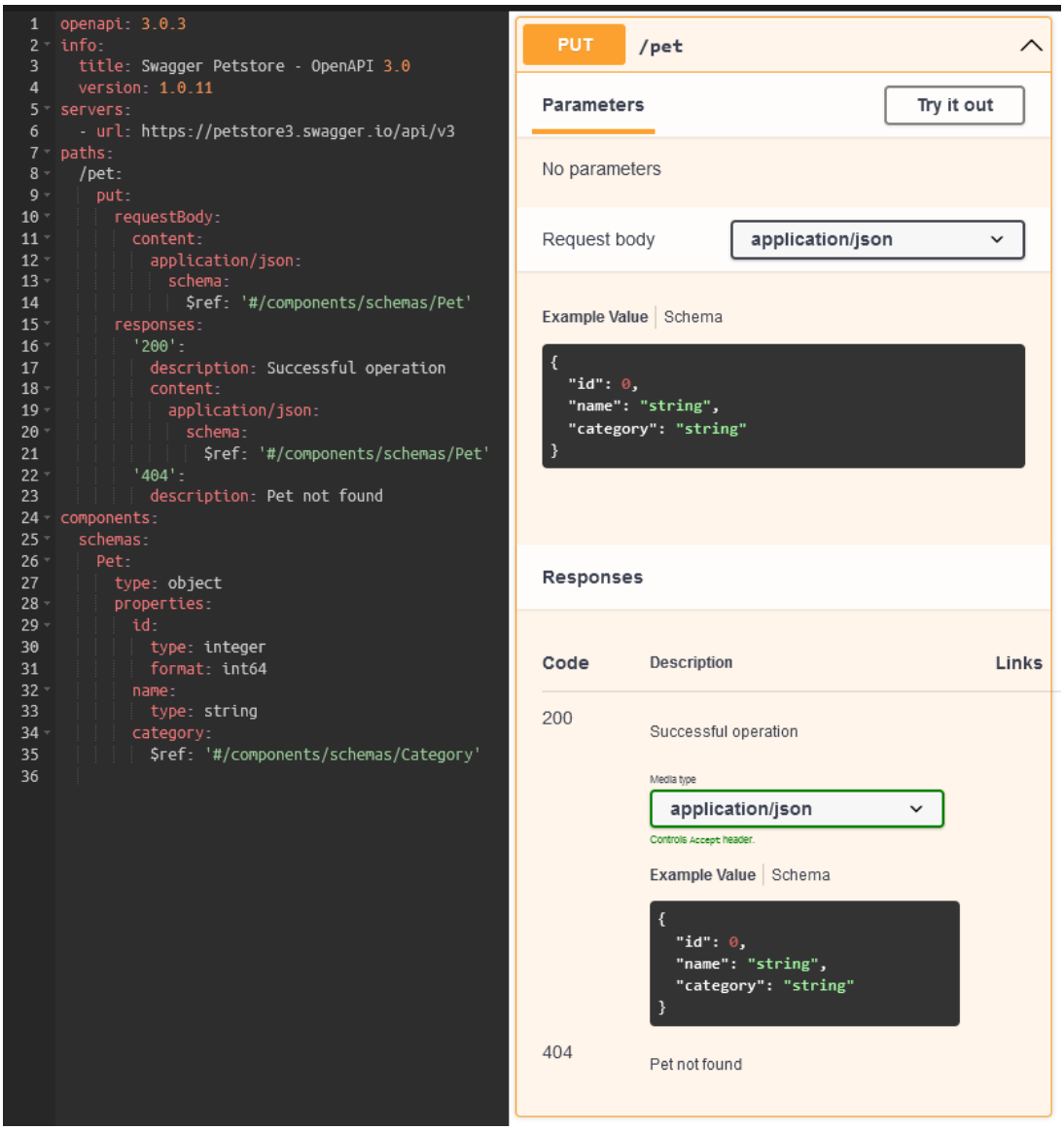

Figure 5.3: OpenAPI Contract

The definition of the dependencies of a service is done using Helm. Helm is package manager for Kubernetes that allows developers and operators 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, this values can be overridden later 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 service dependencies can be seen in Figure 5.4. The dependencies of a service are mapped indirectly, each dependency point to a Helm Chart that is implicitly associated with one service. Helm Charts can be hosted on local or remote repositories.

```
# Chart.yaml
dependencies:
- name: nginx
  version: "1.2.3"
  repository: "https://example.com/charts"
- name: memcached
  version: "3.2.1"
  repository: "https://another.example.com/charts"
```

Figure 5.4: Helm Chart Dependencies Definition Example

The declaration of resources consumed by a service is defined in Chart template files, for each deployment template we define the resources consumed by its containers by defining 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 Figure 5.5.

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

Figure 5.5: Helm Template Resources Definition Example

5.3 Contract Interpretation

The interpretation of contract specification is done in the Parser module. 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. All the forthcoming modules operate under this interface not the OpenApi format. Although we adopted the OpenApi format our solution is not dependent on it, this 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 actual parsing of OpenApi contracts is supported by the swagger-parser project, **[TODO: reference]** which validates and parses OpenAPI definitions in JSON or YAML format into a Java POJO representation.

The interpretation of service resources and dependencies written in Helm charts is also supported by the Parser module under a separate interface. The SnakeYAML library **[TODO: reference]** was used to parse the Charts.yaml and template files, into a Java class that implements a read-only Java interface. The following are also decoupled from the Helm and Kubernetes API's they rely only on the read-only interfaces.

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 evolutions in service dependencies and resource requirements because they can be inferred by comparing the last two contract versions. The description language is hierarchical, and contains array elements. The following Figure 5.7 represents its structure, and Figure 5.6 gives an example. Array elements are depicted by the [.] notation.

As previously mentioned in the design chapter, resolutions can have one of three distinct types DefaultValues, Link or Function. The type of resolution is represented in the first segment of a resolution, while the second segment represents the resolution itself.

Link resolutions have the following format "link=Parameter Location|Parameter Name", the parameter of the prior contract must be of the same type and format.

Default value resolutions have the following format "default=Value", the value that must be of the same type and format.

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 (eg. function= json|firstName + json|lastName). These references are replaced in the adapter implementation by their real value.

```
methods:
- endpoint: /pets GET
  endpointPrior: /pets GET
  messages:
  - parameters:
    - key: query|tags
      resolution: link=query|tags
      type: String
    - key: query|limit
      resolution: link=query|limit
      type: String
    type: Request
    typePrior: Request
  - parameters:
    - key: json|
      resolution: link=json|
      type: Array
    type: '200'
    typePrior: '200'
```

Figure 5.6: Evolution Specification Example

- **[Methods]**: List of the procedures supported by the new service.
  - **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]:** The list of the messages exchanged by the procedure, request messages and response messages.
    - **Type:** The response or request type in the new contract (e.g 200,400,... and Request there is only one request type)
    - **Prior Type:** The response or request type in the previous contract
    - **[Parameters:]** The list of the parameters of the message (e.g Path, Query, Header, Cookie and Body parameters)
      - **Key:** The identifier of the parameter, the first segment represents its location (e.g Path, Query, Header, Cookie and Body parameters), the second segment represents the name of the parameter. Hieratical Body parameters can be mapped at any level by specifying the hierarchy path in the parameter name (eg. json|specie.class.name; json|specie)
      - **Resolution:** The evolution resolution for the parameter, details will be provided in the next paragraph.
      - **Type:** The parameter type: Number, String, Array or Object

Figure 5.7: Evolution Specification Structure

To minimize the documentation effort we developed a GUI editor for the specification of evolutions. The GUI editor was implemented as an IDE Plugin, the current implementation supports the Intellij IDE. The editor compares two contract versions, and automatically maps all the parameters that haven't changed from one version to the other, the developer is left with the task of explicitly mapping the remaining procedures. The editor starts by requesting the developer to map endpoints in the new contract to endpoints in the old contract, afterwards the developer must edit 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 holds the resolutions that were already mapped and uppermost left column allows the developer to write rules for each unresolved parameter. The developer can select leaf parameters in the hierarchy, or we can choose intermediate properties. If he chooses an intermediate property and inputs resolution rule all child parameters are also resolved. After all parameters are resolved the developer can submit the specification, and the editor outputs an evolution spec with the same description language presented in the beginning of this section.

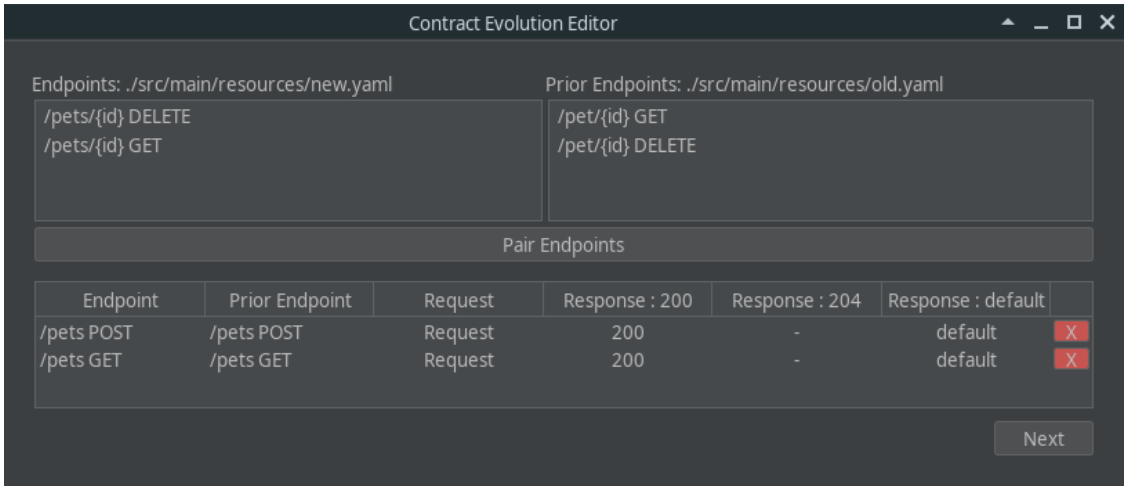

Figure 5.8: Evolution Editor

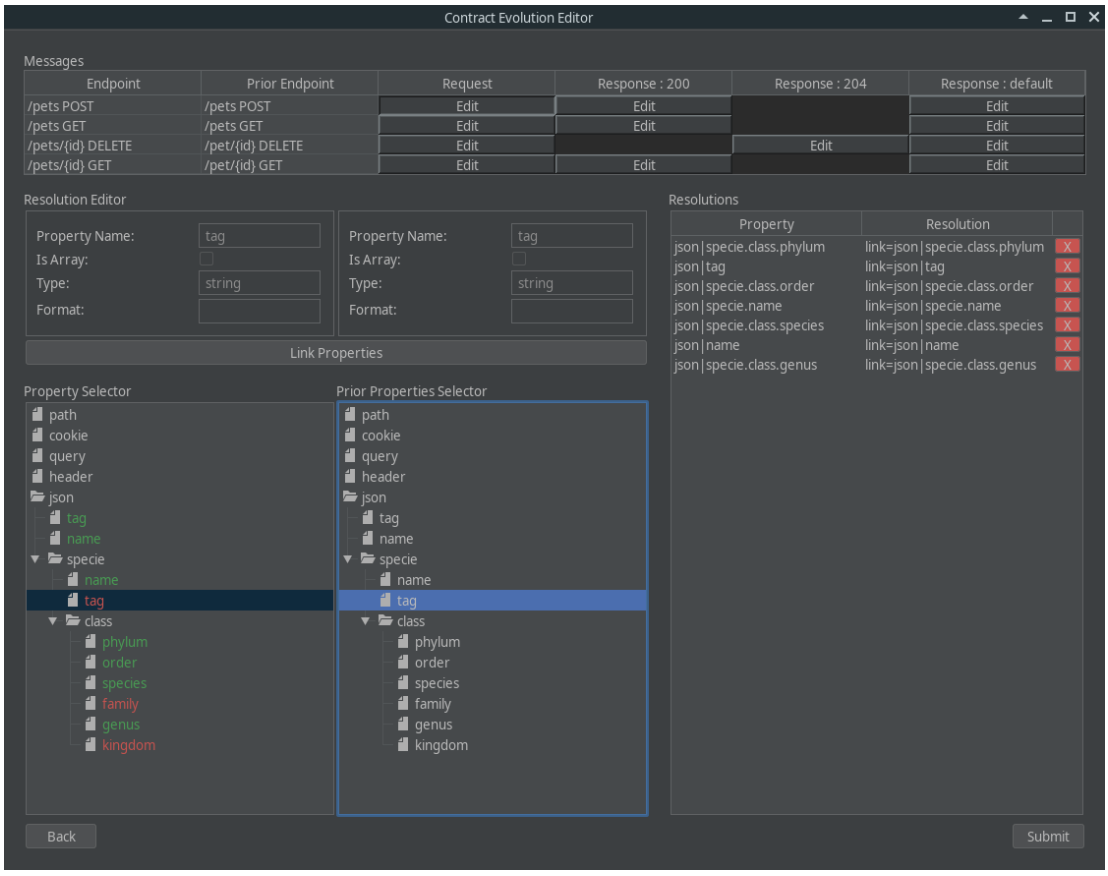

Figure 5.9: Evolution Editor

5.5 Compatiblity Verification

The compatibility verification process is done by the verifier module. This module was implemented in java as command-line application. The module verifies if the provided evolution specification is valid. It starts by checking if all the new contract procedures, messages and parameters are present in the specification, then it verifies if all the resolutions for each parameter are valid. 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 test 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 would be useful in order for the solution to be decoupled from Helm.

5.6 Adapter Generation

The generation of the adapter module is done by the adapter module. The adapter is generated by populating a template implementation with the information provided in evolution specs. The adapter must be able to support all prior versions that are still in use, and adapt to the last version, in other words a single adapter is used to support the adaptation of multiple versions to the current version.

The 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 (e.g merge[v1->v2, v2->v3] = v1->v3).

This module begins by querying the active versions of the target service with the kubectl command-line tool. Kubectl is 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. Figure 5.10 exemplifies the merge process.

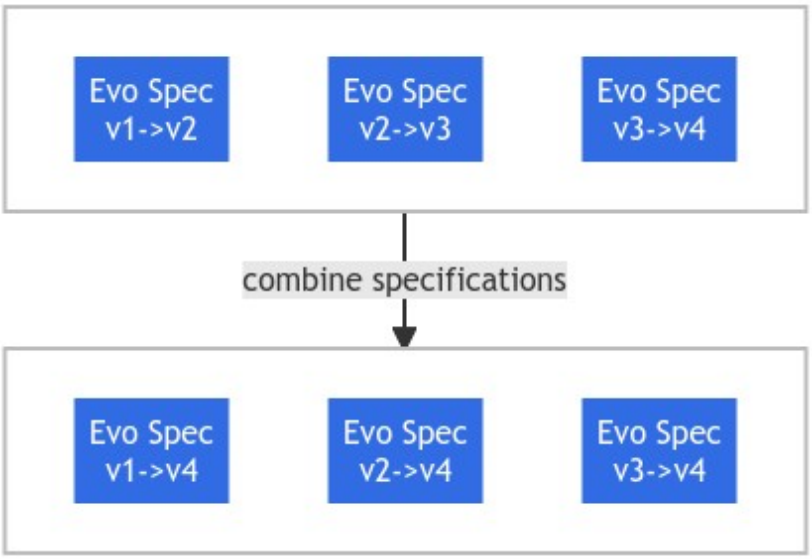

Figure 5.10: Evolution Specification Merge Process

The generated adapter contains one endpoint for each of the procedures in each version. Identical procedures in different versions are distinguished by versioning their endpoint paths. It is also possible to version procedures with header parameters, but the generated adapter code must be more complex because a single endpoint must be able to adapt messages from multiple versions.

5.7 Adapter

The adaption approach is straightforward: for each endpoint the adapter re-constructs the entire request from zero and populates the parameter resolutions with the information of the original request. The implementation of "Function"resolutions is simple since the expressions are written in java and only have one line; they may be injected directly into the template implementation, and the parameter references are then replaced with the parameters themselves.

The adapter's current template implementation does not allow HTTPS, however this may easily be added. 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 maximum number of threads for Tomcat was set to 2000, in order to support a higher load of requests.

6 Results

6.1 Benchmark platform

We plan to construct a benchmark platform to compare our solution to existing ones; this section discusses the proposed benchmark platform's requirements and design.

Context	Requirement	Assessment Rationale
Architecture	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.
DevOps	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.
General	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.

Figure 6.1: Benchmark Requirements [1]

Aderaldo et al. propose an initial set of requirements to support repeatable microservices research. In addition to the requirements listed in the 6.1, the platform's essential requirements are as follows:

- The ability to evaluate different solutions in comparable scenarios while utilizing

the same evaluation criteria.

- Evaluating solutions without requiring modifications to implementations.
- Experiments must be simple to share and reproduce by different individuals.
- 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 section 6.2. The architecture components, and their applications are described below:

Kubernetes [28] will be the test environment. It will host the services holding the solutions implementations. The loading testing scripts will also be managed via services in Kubernetes.

Promotheus [32] is a pull-based monitoring system. It periodically sends HTTP scrape requests, the response to this 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 will be visualized in this platform via Grafana dashboards.

Artillery Artillery is an open-source performance and reliability testing suite for developers and system administrators.

Jenkins Jenkins is a free and open automation server. It aids in the automation of portions of software development such as building, testing, and deploying, allowing for continuous integration and continuous delivery.

6.2 Evaluation Methodology

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.

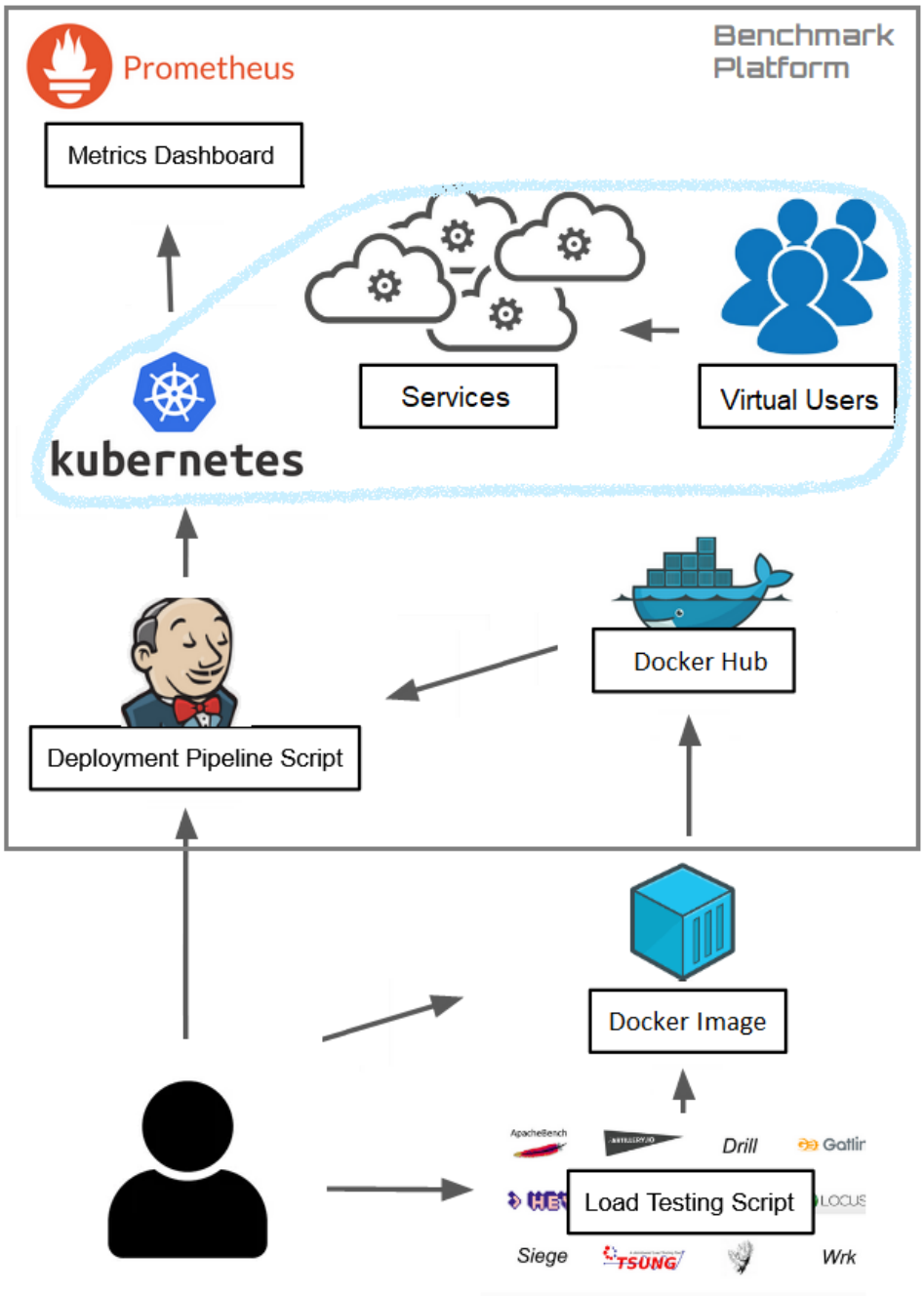

Figure 6.2: Benchmark platform components

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 returns received requests unchanged. The interconnections between microservices is configurable thorough environment variables. The number of synchronous calls (SyC) between microservices for a single request is also configurable. The number of parallel calls (PrC) between microservices is also configurable. Two http endpoints are contained in the image, which was developed in two separate versions with unique contracts. In one of the endpoints, all the parameters in the body's schema were given new names.

Experiment Setup To inject load in the experiments, 10 workers of artillery load testing tool were used, the workers are executed remotely in the cluster and were spread in 6 nodes. The workers are supported by kubernetes pods that are monitored by a common kubernetes job. The workers are installed and removed from cluster using the artillery operator project, that enables the creation and execution of distributed Artillery load tests from a Kubernetes cluster, at scale. In each experiment all the resources of previous experiments were deleted and a new job and namespace were created in the cluster to avoid contamination in the metrics extracted. Each experiment was executed in sequence with a pause of 2m.

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

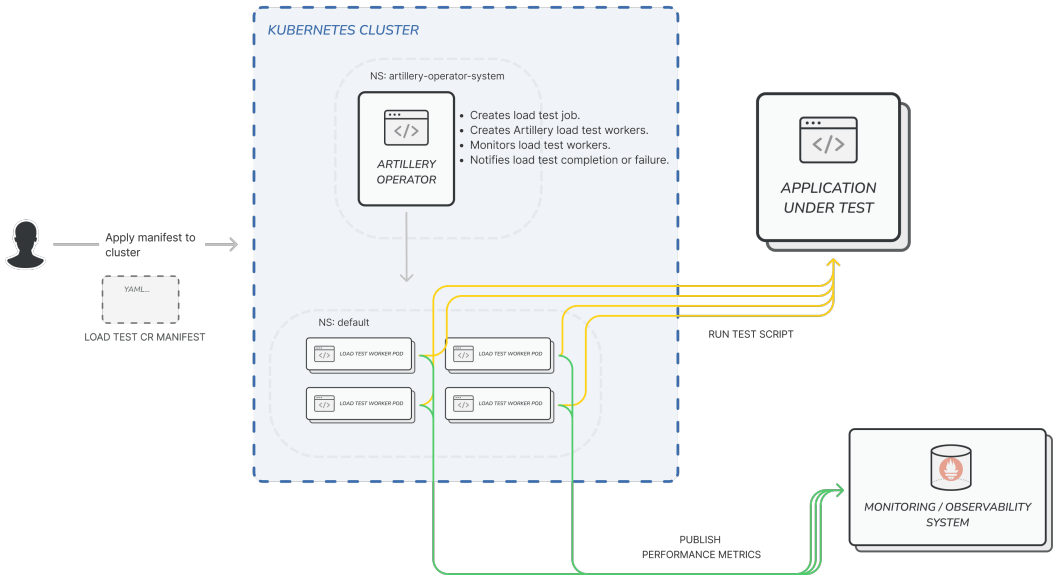

Figure 6.3: Artillery Operator Architecture

Metrics In each experiment we extracted 3 metrics, the cpu/memory resources and latency of the system. The resources were only queried from the adapters, and the replicas that belong to the target microservice application. The resources consumed by the monitoring system and artillery workers were excluded and have no effect on the results. The throughput and latency were extracted with the use of the artillery load testing tool.

The CPU resource samples were taken from each node at 5-second intervals using the prometheus node exporter. Memory resource samples were gathered from each container at 5-second intervals using the kubelet node agent api. The latency statistics were calculated with a period of 10s over 5 min, the results obtained represent the average of the reported 30 periods.

6.3 Experiments

This section presents measurements obtained from a prototype implementation throughout various experiments.

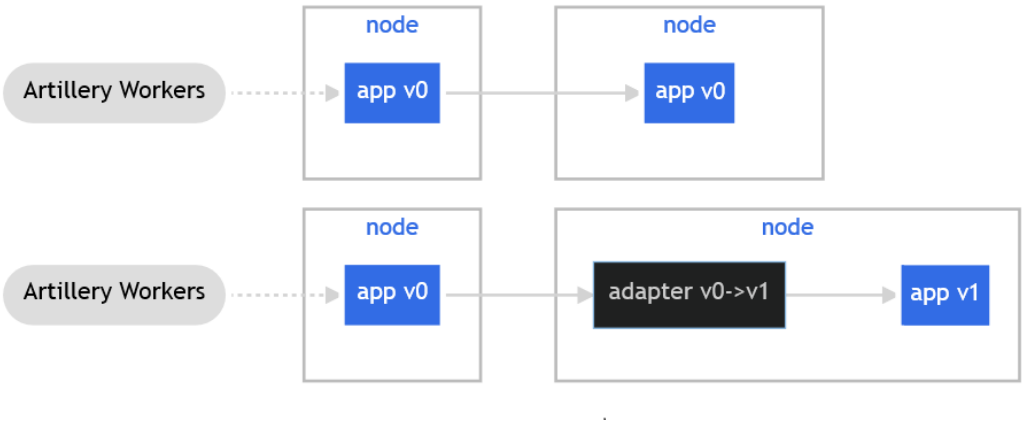

Figure 6.4: point-to-point experiment

Point to Point The topology of this experiment is represent in Figure 6.4. This 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.

We first established a baseline of comparison by doing the tests without the adapter, and then we introduced the adapter and repeat the tests. The primary goal of this experiment was to assess the overhead of the adapter in terms of latency and computational resources.

The measured cpu overhead of the adapter over a range of request rates is depicted in Figure 6.5. 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.

In this experiment it is expected to have an overhead of 150%, because the adapter and the microservices share the same request rate, and because 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.

As can be seen in the results shown Figure 6.5 the measured overhead doesn't fall to far outside our expectations, with the exception of the first data point. The overhead 84% is explainable by the configuration of the containers. The adapter and demo application containers were initialized in kubernetes with a default of 100 millicpu (10% of tested 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.

Figure 6.6 shows the measured memory overhead of the adapter over a range of request rates. The average cost of 190% in Figure 6.6 is understandable because, although being stateless, the adapter stores two messages in memory for each request, the request message and the adapted request message. Because the demo application only keeps one request in memory, it stands to reason that the adapter memory cost is the double of the demo application. As can be seen in Figure 6.7 the adapter real memory cost grows linearly with the request rate.

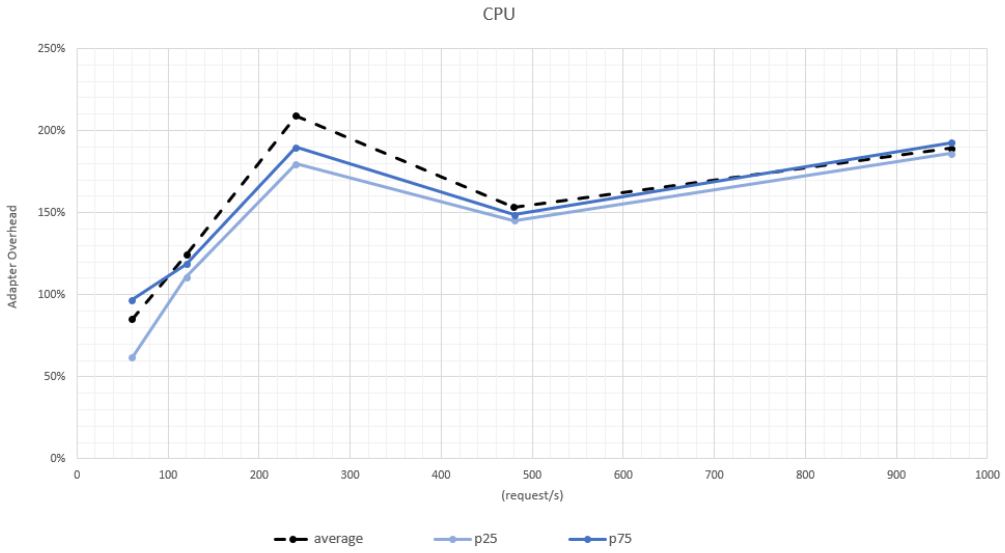

Figure 6.5: Adapter CPU overhead in the point-to-point experiment
100 bytes (payload size)

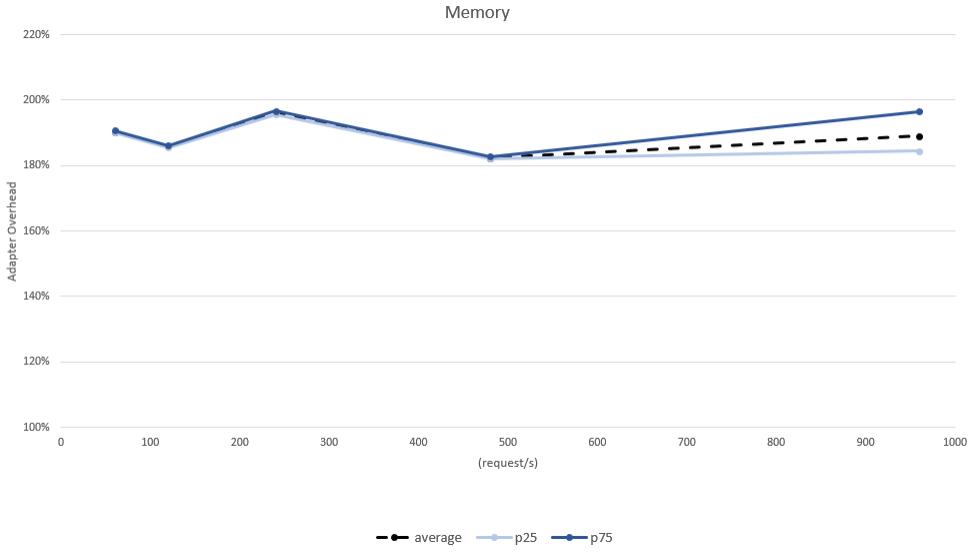

Figure 6.6: Adapter memory overhead in the point-to-point experiment 100 bytes (payload size)

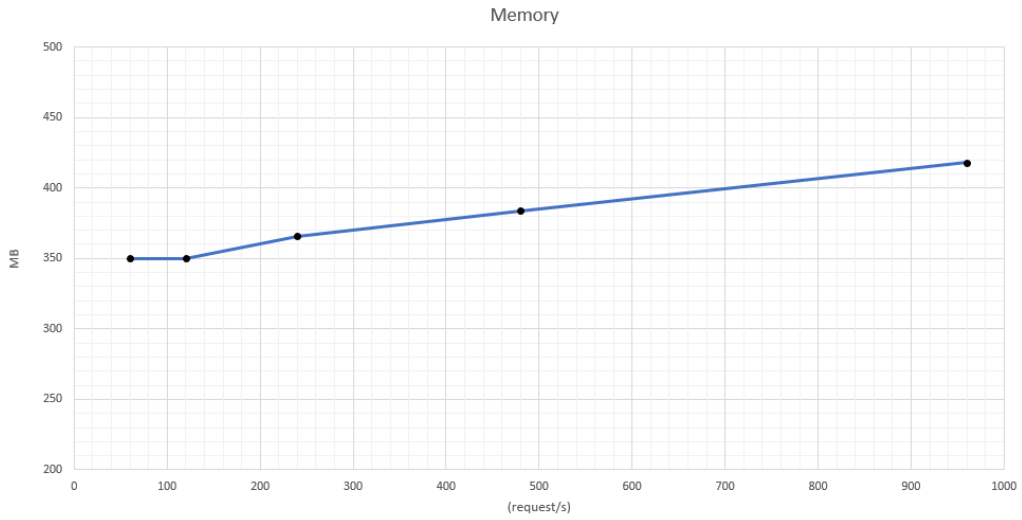

Figure 6.7: Adapter memory cost in the point-to-point experiment 100 bytes (payload size)

The latency overhead of the adapter in the point-to-point experiment is depicted in Figure 6.8.

It is expected for the overhead to be less or equal to 150%, because the latency overhead is largely attributable to communication cost. The adapter introduces an additional communication step "app1->**adapter->app2**"to the baseline communication chain that already haves two steps "workers->app1"and "app1->app2". The introduced communication step is expected to have a slightly inferior cost to the other steps, because it is performed via interprocess-calls, while the others are done via a private network with 2 GB of bandwidth. Our hypothesis is confirmed by the experiment results in Figure 6.8.

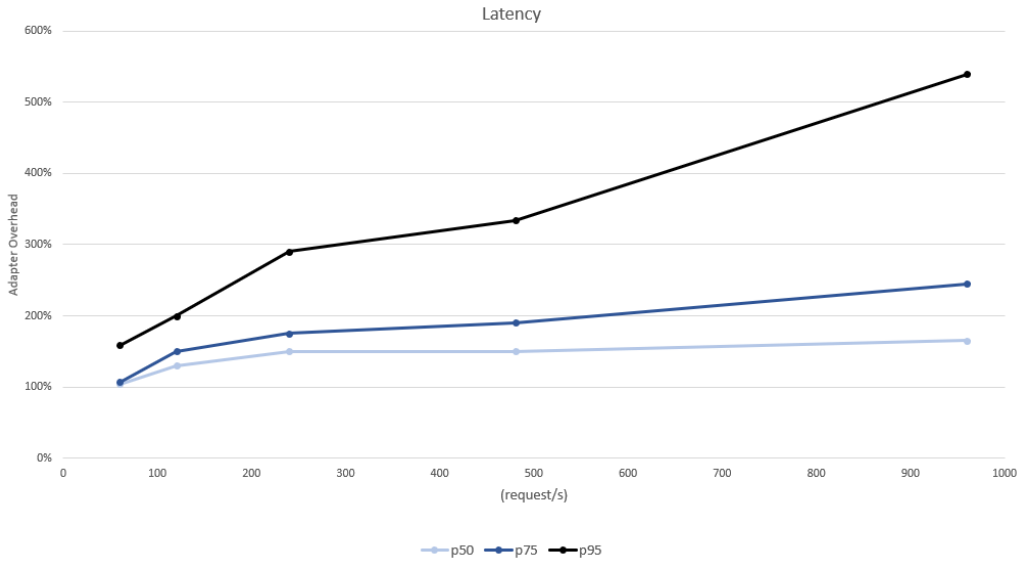

Figure 6.8: Adapter latency cost in the point-to-point experiment
100 bytes (payload size)

The Figures 6.9 and 6.10 illustrate how larger payload sizes effect 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 typical HTTP packet size 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. The results illustrated don't reveal any difference for the tested range of payload sizes. We selected the range of 100 bytes to 2 KB because most services use http messages that fall under this range. We intend to repeat this experiment for larger payload sizes up to 2 MB. With larger payload sizes it will be necessary to sent multiple packets to transmit one message, we intend to see how the adapter scales in this scenario.

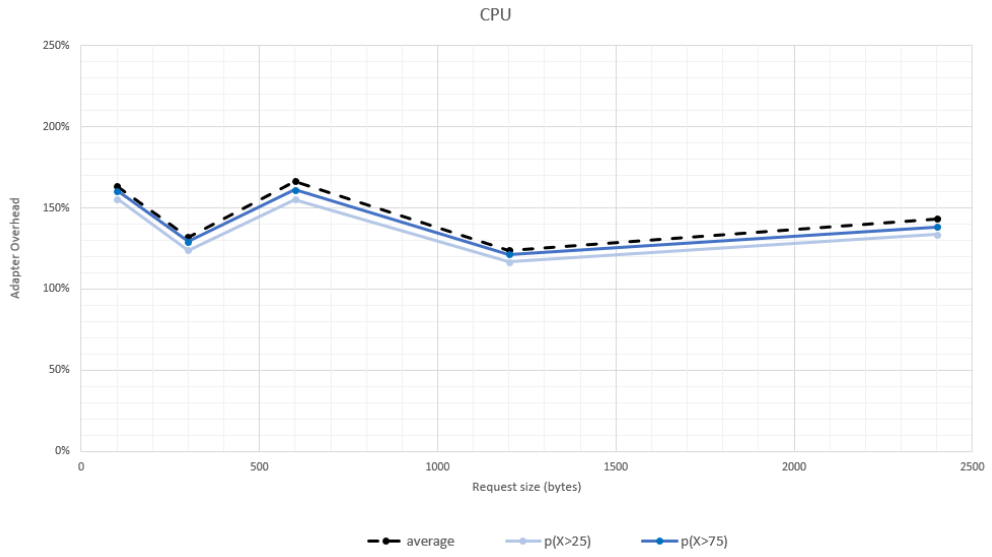

Figure 6.9: Adapter CPU cost in the point-to-point experiment
480 request/s (arrival rate)

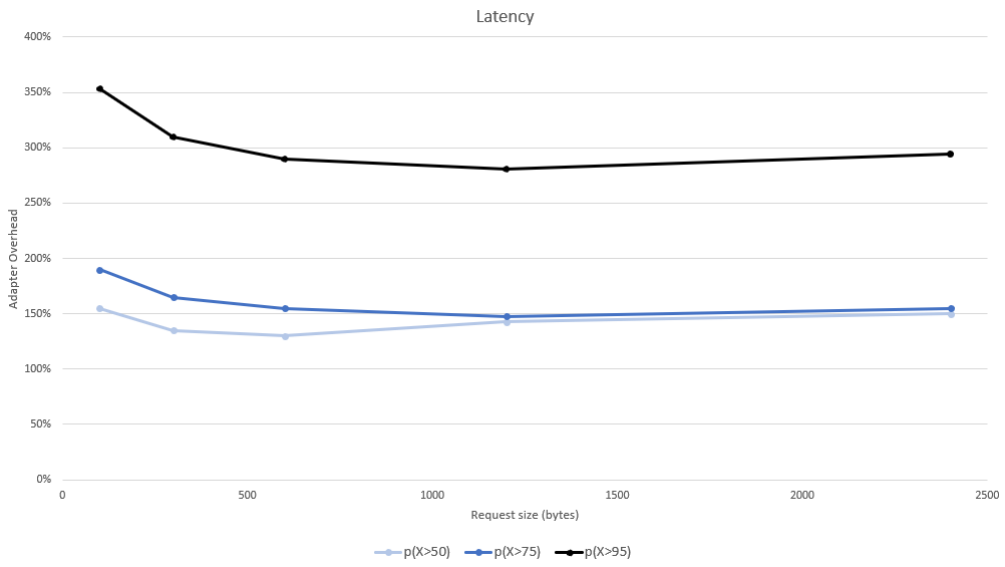

Figure 6.10: Adapter Latency cost in the point-to-point experiment
480 request/s (arrival rate)

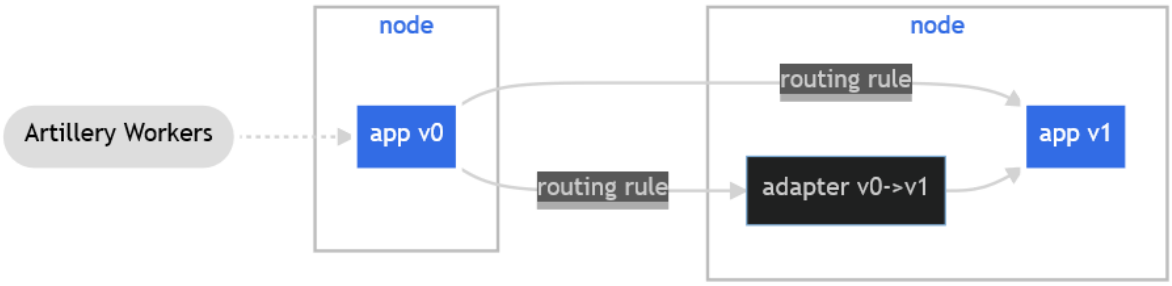

Figure 6.11: routing experiment

Weighted Point to Point The topology of this experiment is represent in Figure 6.11. 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 6.12 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. In this experiment it is expected for the overhead to steadily increase, because as the adapter receives a higher percentage of requests, its request arrival rate will increase, to support higher request rates the adapter implementation will need to launch more client threads, which consequently increase the consumed CPU resources.

When we compare the first and last data points in Figure 6.12, we can see that consumed resources have doubled as expected.

In the other data points, the overhead remains relatively constant with a margin of error of 20%, this explainable because the default amount of allocated threads by Tomcat (200 threads) is sufficient to handle the arrival rate in these data points. When 50% of requests are routed through the adapter, the arrival rate is 240 requests/s, the tomcat thread model uses a thread per request, consequently 240 threads are necessary to handle the arrival rate, which can be translated to an overhead of 120%. The 20% margin of error in the other data points explainable by the nondeterministic nature of the artillery workers. 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 percentages; however, the distribution of requests is nondeterministic and will only be equal to the given weights over very long test periods; the test time for 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 6.13. 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 spatial complexity. The results obtained confirm our expectations.

Figure 6.14 shows the request latency. In this experiment it is expected for the latency to steadily increase with the fraction of requests routed through the adapter, because of the same reasons previously stated. When we compare the first and last data points in Figure 6.12, 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, we intend to repeat this test with a longer time window.

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

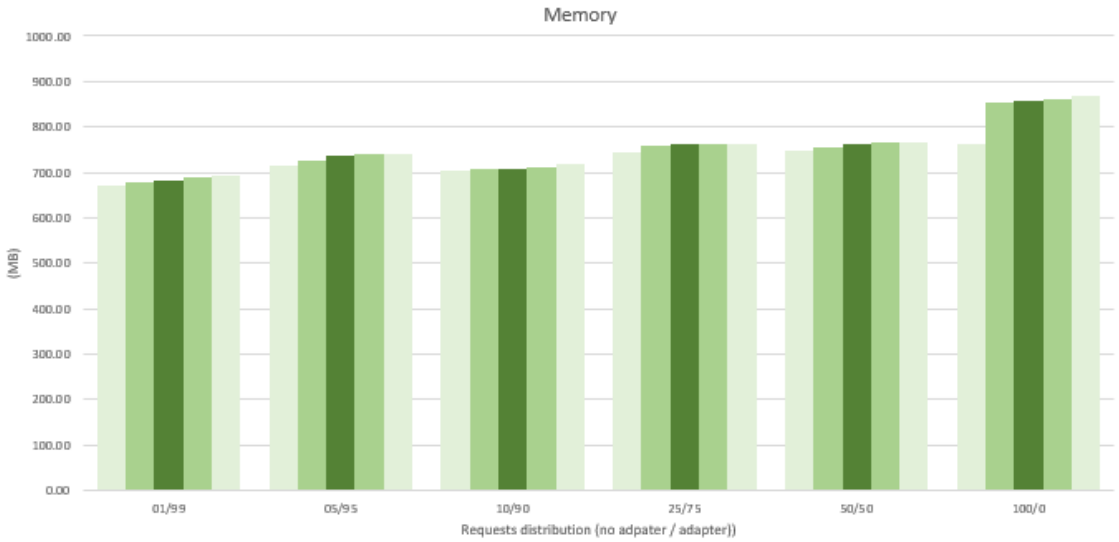

Figure 6.13: Memory used in the routing experiment
480 request/s (arrival rate) 100 bytes (payload size)

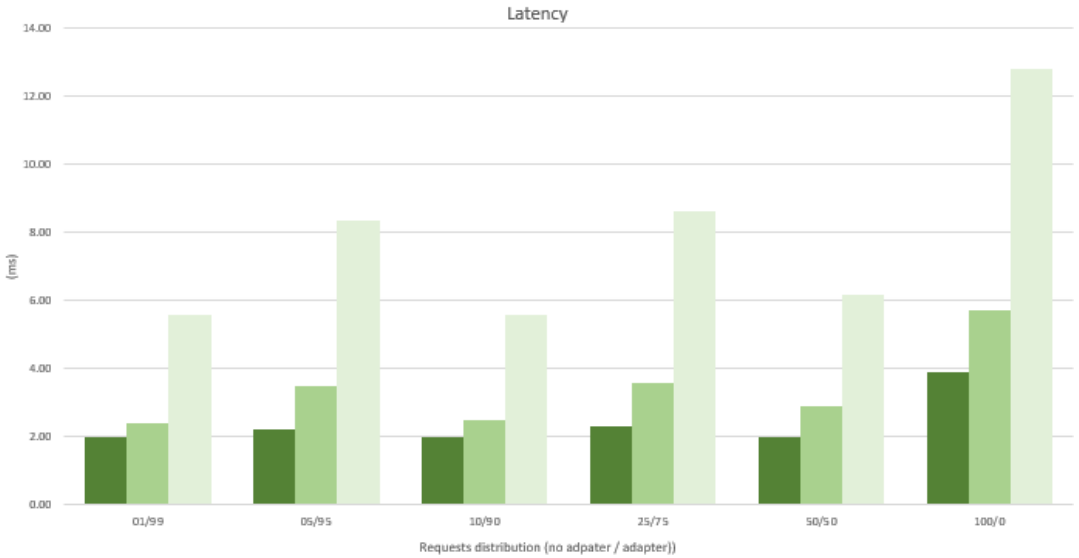

Figure 6.14: Request latency in the routing experiment
480 request/s (arrival rate) 100 bytes (payload size)

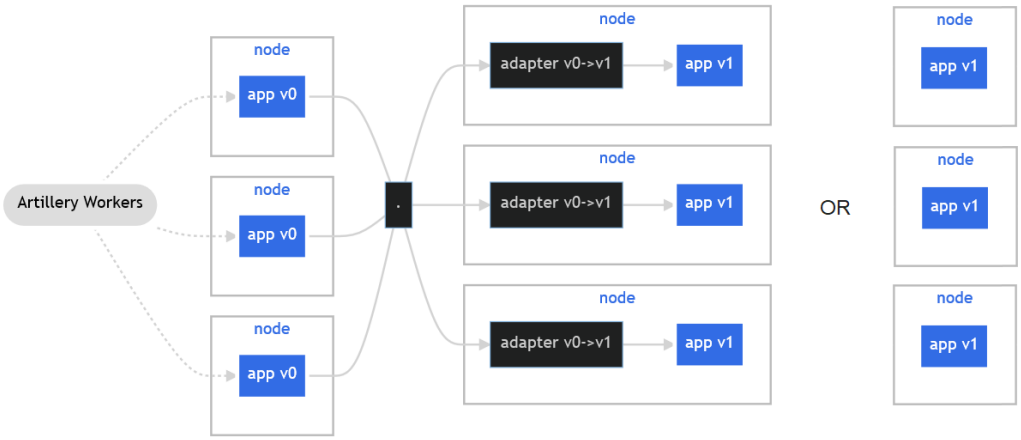

Figure 6.15: bi-partition experiment

Bi-partition The topology of this experiment is represent in Figure 6.15. This 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. We first established a baseline of comparison by doing the tests without the adapter. Then we upgraded all the microservices in the second set and added adapters to support communication across the disjoint sets. The goal of this experiment was to see if the conclusions obtained from the point-to-point experiment are also applicable in scenarios with multiple microservices.

The outcomes of the "point-to-point"and "bi-partition"experiments are equivalent, as seen in the Figures 6.16, 6.17 and 6.18.

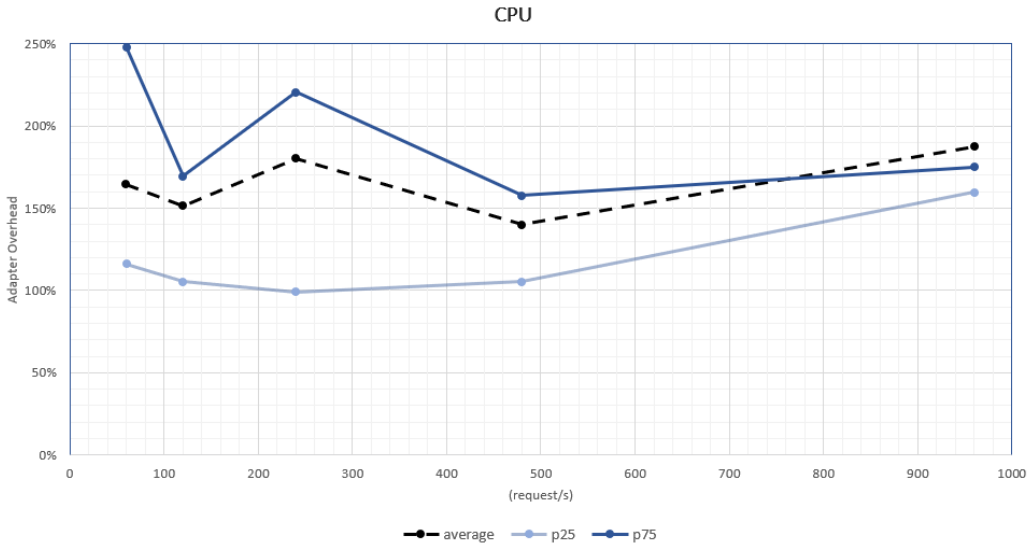

Figure 6.16: Adapter CPU overhead in the bi-partition experiment
100 bytes (payload size)

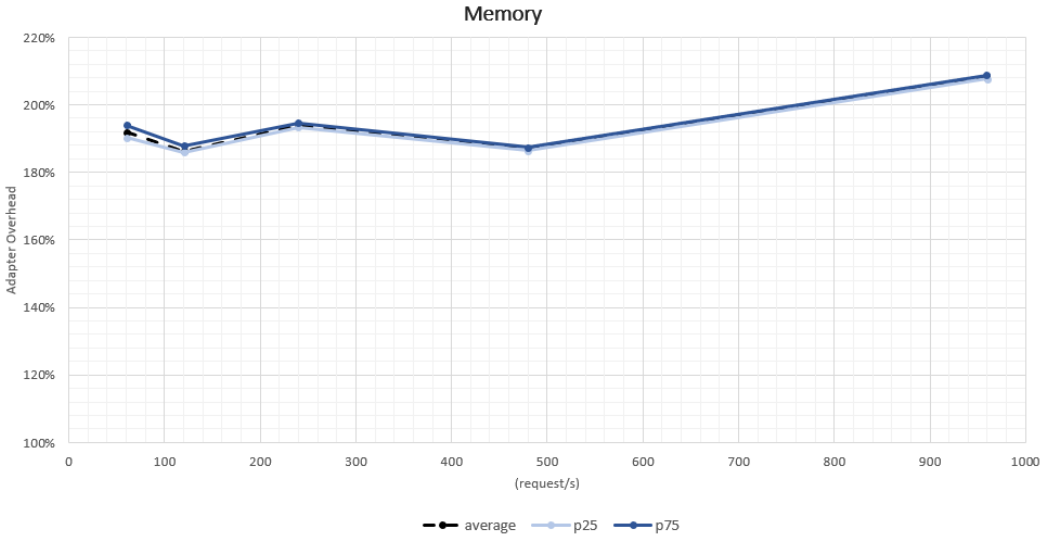

Figure 6.17: Adapter memory overhead in the bi-partition experiment 100 bytes (payload size)

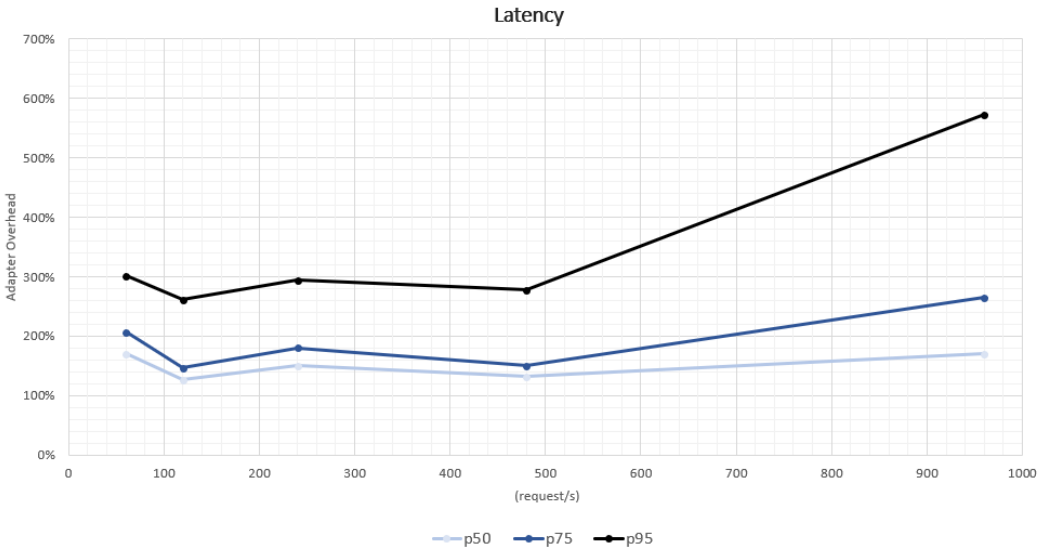

Figure 6.18: Adapter latency overhead in the bi-partition experiment 100 bytes (payload size)

7

Conclusions and Future Work

Throughout the sections 4.2.3 and 5.5, we demonstrated the safety guarantees of our solution while refactoring parts of a distributed system: deployments that would lead to irremediably incompatible states are rejected, such as a microservice attempting to utilize a renamed endpoint, or a microservice being deployed before its dependencies are alive and reachable.

Additionally, we argued how the proposed solution could be supported under widely adopted workflows and documentation formats, such as DevOp pipelines and the OpenAPI description language.

We also discussed approaches to lessen the solution's additional documentation requirements. Section 5.4 provides an IDE tool that automatically identifies unmodified service procedures and possible contract evolutions, with the goal of minimizing the effort necessary to explicitly document all contract evolutions.

The proposed solution is agnostic to the underling application, all of its components don't require the modification of application microservices or their architecture. We offer a solution that may be plugged into any distributed application that employs HTTP as its communication protocol, with the only condition being that the application endpoints be versioned through path segments or header parameters.

One conclusion we can draw from the results is that the latency introduced by an embed adapter is similar to the latency introduced by an additional call between two distinct nodes; 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 with the arrival rate of requests that must be adapted. If only a small fraction of requests require adaptation, the computational cost of the adapter will be low in comparison to the cost of the application.

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, which has a significant thread overhead and cannot handle highly concurrent applications. The adoption of a Non-blocking I/O (NIO) model,would result in improved performance and lower CPU resource usage than the traditional BIO model.

Lastly, we believe it would be interesting to use a formal language to define function resolution in contract evolution specifications. The current implementation supports function resolutions with 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. 9).

[3] A. P. Authors. *APIs.guru*. https://apis.guru/.

[4] A. P. Authors. *Argo*. https://argoproj.github.io/argo-workflows/.

[5] O. G. P. Authors. *OpenAPI Generator*. https://github.com/OpenAPITools/openapi-generator.

[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 p. 22).

[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. 9).

[11] 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).

[12] G. Developers. *Protocol Buffers*. http://code.google.com/apis/protocolbuffers/ (cit. on p. 16).

[13] O. Developers. *Java RMI*. https://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmiTOC.html (cit. on p. 17).

[14] N. Dragoni et al. "Microservices: yesterday, today, and tomorrow". In: *Present and ulterior software engineering* (2017), pp. 195–216 (cit. on p. 6).

[15] E. Evans and E. J. Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004 (cit. on p. 8).

[16] X. Feng, J. Shen, and Y. Fan. "REST: An alternative to RPC for Web services architecture". In: *2009 First International Conference on future information networks*. IEEE. 2009, pp. 7–10 (cit. on p. 14).

[17] 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. 15).

[18] 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. 20).

[19] M. Kleppmann. *Schema evolution in avro, protocol buffers and thrift*. 2013 (cit. on p. 18).

[20] J. L. Martin Fowler. *Microservices*. http://martinfowler.com/articles/microservices.html. 2014 (cit. on pp. 1, 2, 6–8, 21).

[21] 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. 17).

[22] F. Menge. "Enterprise service bus". In: *Free and open source software conference*. Vol. 2. 2007, pp. 1–6 (cit. on p. 8).

[23] M. P. Papazoglou et al. "Service-oriented computing: State of the art and research challenges". In: *Computer* 40.11 (2007), pp. 38–45.

[24] 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 pp. 2, 21).

[25] 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).

[26] R. Ratovsky. *OpenAPI*. https://swagger.io/specification/ (cit. on p. 14).

[27] 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. 21).

[28] G. Sayfan. *Mastering kubernetes*. Packt Publishing Ltd, 2017 (cit. on pp. 9, 48).

[29] 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. 10–12, 21).

[30] 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. 16).

[31] F. Soppelsa and C. Kaewkasi. *Native docker clustering with swarm*. Packt Publishing Ltd, 2016 (cit. on p. 9).

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

[33] 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. 15).

[34] D. Vohra. "Apache avro". In: *Practical Hadoop Ecosystem*. Springer, 2016, pp. 303–323 (cit. on p. 16).

[35] O. Zimmermann. "Microservices tenets". In: *Computer Science-Research and Development* 32.3 (2017), pp. 301–310 (cit. on p. 6).