NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF INFORMATICS

RODRIGO JORGE RIBEIRO

Bachelor Degree in Computer Science

CONTRACT EVOLUTION IN A MICROSERVICES ARCHITECTURE

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon

Final: *February 18, 2022*

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF INFORMATICS

CONTRACT EVOLUTION IN A MICROSERVICES ARCHITECTURE

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: *February 18, 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 (services), however they offer no 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 of automated tools. 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

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

Architecture is the most important factor that influences the capabilities and constraints of a system's evolution throughout its lifecycle [9]. In software engineering, the subject of architecture is the conciliation of various architectural systems or components with the objective of satisfying a system's functional and non-functional requirements.

That is, a software architecture can be seen as a set of architectural elements that are divided into three classes: processing elements, data elements, and connecting elements [25]. The data elements contain the state of system; The processing elements transform the data elements. The connecting elements, which may be either processing or data elements, or both, are the glue that keeps the architecture together. (e.g. procedure calls, shared data, messages) The connecting elements are frequently used to differentiate one architecture from another, because among the other elements, they have the greatest impact on non-functional requirements, such as the system's ability to evolve, security, scalability, and so on.

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 a monolithic architectural design pattern is the Model-View-controller (MVC) [11, 20]. Although the MVC pattern encourages separate layers, which can improve software maintainability, such structures are notorious for their lack of agility and scalability due to their monolithic nature. Creating instances of massive monoliths is time-consuming and inefficient; even minor system updates require the complete redeployment of the application, severely limiting a system's ability to evolve.

Service-Oriented Computing (SOC) [23] is a paradigm for distributed computing. In SOC, a program known as a service provides functionality to other components via network message passing. Messages are exchanged using service interfaces, which are

implementation-independent. The first generation of service-oriented architectures (SOA) [24] set imposing and ambiguous criteria for services, which hampered its adoption. Microservices [20] are a modern iteration of the SOC idea; their aim is to reduce excessive layers of complexity so that developers can focus on building simple services that efficiently fulfill a specific task.

Some advantages of service orientation over monoliths are as follows:

- Functional independence - Each service is operationally independent, the only way services communicate is through their published interfaces.
- Component Scalability - When a single component becomes overloaded, additional instances of the component can be deployed separately.
- Distributed development - Separate teams may work on different modules of the system simultaneously if they agree on the interfaces of modules ahead of time.

These advantages boost the system's ability to evolve by allowing it to be disassembled into subsystems composed primarily of loosely coupled services.

In a microservice architecture, subsystems can fully evolve independently of one another, as long they don't share services or components. 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 (Section 2). Sequent sections, first introduce a summary of the key related work (Section 3), next a more detailed description of the proposed approach to the problem stated (Section 4) and conclude with the expected phases of the work plan (Section 5).

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;

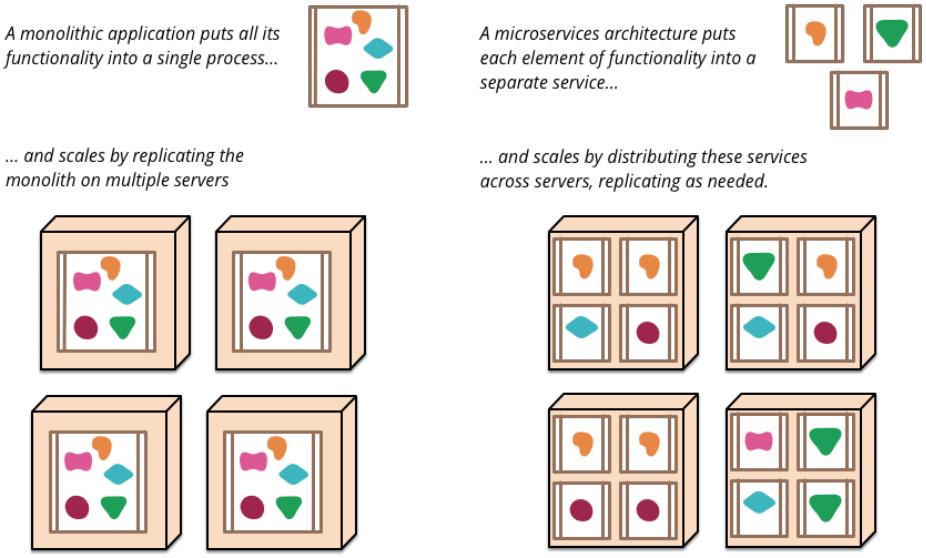

Figure 2.1: Monoliths and Microservices [20]

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.

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"

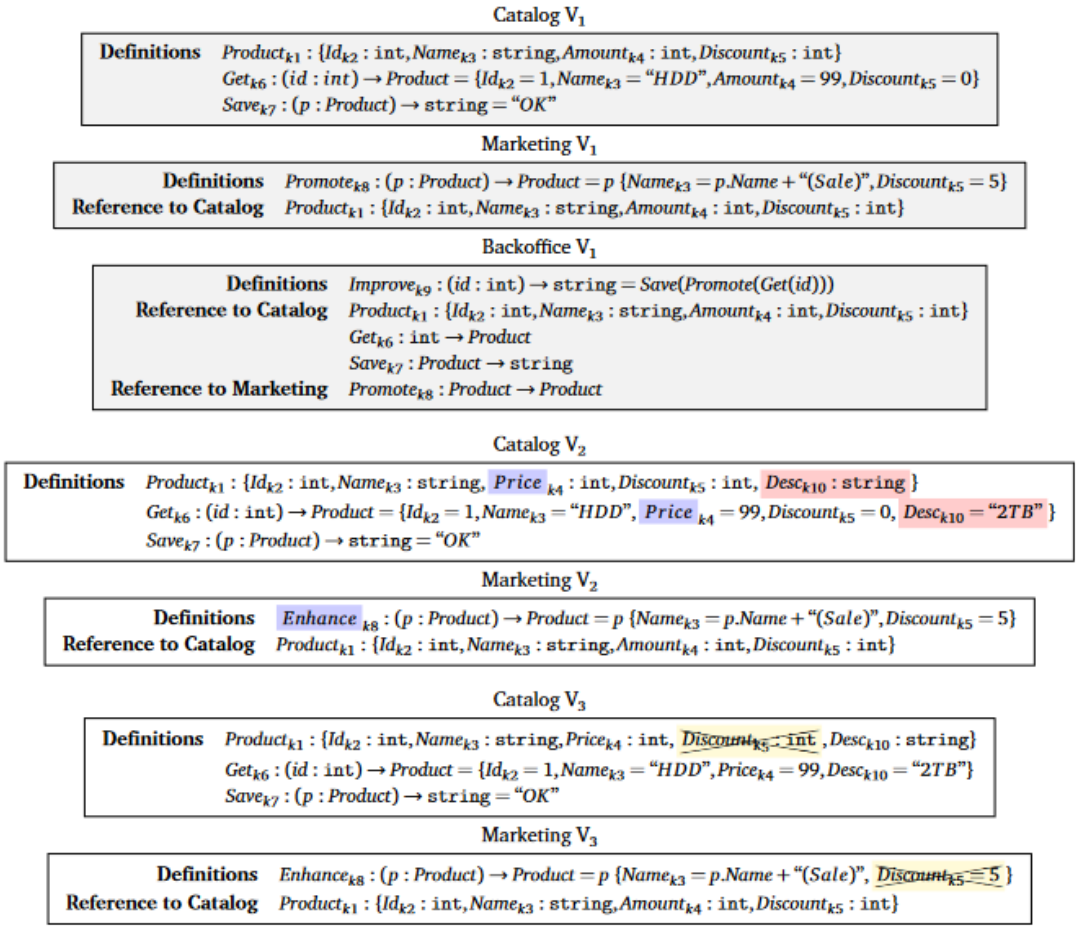

Figure 2.2: Evolution of modules Catalog, Marketing, Backoffice [29]

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.

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.

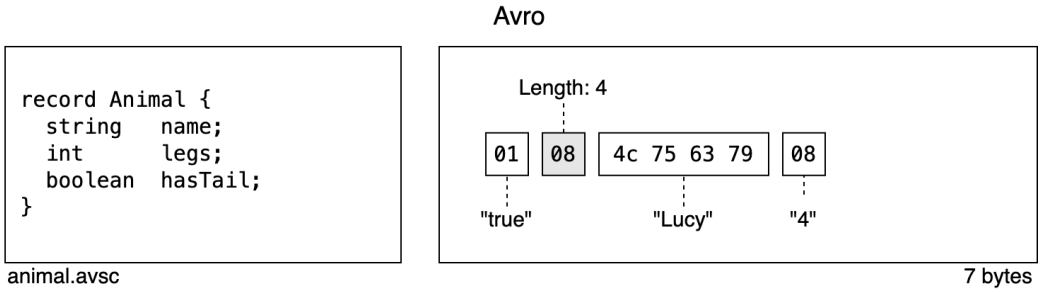

Figure 3.1: A Avro schema and its associated serialized record

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.

The conventional approach for evolving microservices is to keep various 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. Bellow we present more effective service evolution approaches.

3.4.1 Schema Resolution Rules

Frameworks such as WDSL, Avro, Protocol Buffers and Thrift support schema evolution 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, their schema evolution support is limited [19]:

- Only the addition, removal and renaming of fields is allowed.
- To ensure backwards compatibility, added fields must be optional.
- The only fields that can be removed to maintain forward compatibility are optional fields.
- To provide both forward and backward compatibility, removed and added fields must be optional.

Backward compatibility refers to a consumer using schema X to process data produced by schema X-1, whereas forward compatibility refers to data produced by schema X being read by consumers using schema X-1.

The need for optional fields in the above cases is due to a lack of information on the consumer, typically this information is supplied with the use of default values. In order to support required fields with backwards-forwards compatibility, 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 on the schema.

These limitations are problematic because, 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 RESTful communication approach is adopted, these frameworks will be unable to manage changes in the signature of endpoints (e.g. changing the method of REST endpoint's from GET to POST); only changes in record schemas will be managed in this case. To decrease the complexity of the evolution adaptation process, we believe it is necessary to handle both the evolution of records schemas, and the evolution of API signatures in a single integrated approach.

3.4.2 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.

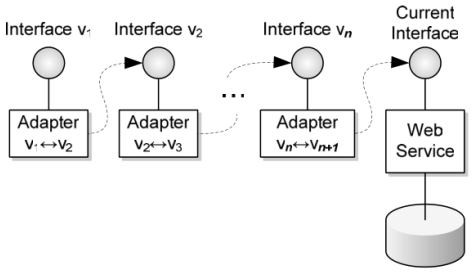

Figure 3.2: Chain of Adapters structure after n versions have been published

The method used in this technique may be used in our solution as well. The availability of our system can be increased by employing this method as a placeholder adaption mechanism for when the adapter proxy is being built and initialized for earlier contract versions.

3.4.3 Proxy Adapter

Seco et al. and Santos et al. investigate this approach [29, 27], this method will be further explored and evaluated in our solution.

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 proxy creation mechanism makes use of the available meta-information to automatically update the proxy components as needed.

The proxy can be installed on either consumers or producers; however, it is usually preferable to install the proxy on consumer services to avoid centralizing the load of message adaptation in a single point. When the service is also consumed by externally developed clients, it is usually beneficial to install the proxy in producers, since it is difficult to install the adapter in uncontrolled clients.

Services are deployed without a proxy, proxies are installed on demand using a lazy instantiation method, that creates them when the first communication between corresponding services occurs.

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 replaceability 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 S imposes constraints on input parameters, which are less restrictive than those of OR input parameters in SR (e.g., differences in value ranges).

The two first types of mismatches can be handled in the context of microservice evolution without the use of adaptors. 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 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

Proposal

This section defines all our approach for microservice contract evolution. The compatibility verification, adaptation approach, and benchmark platform will all be described.

4.1 Scope

The proposed solution will support HTTP contracts, because most applications require a web presence and microservice teams prefer to use a single protocol for both internal and external communications.

Other event-driven and request-response protocols, such as RPC, have simplified contracts, and there are already a number of tools that support the evolution of schemas for these protocols, albeit with some limitations.

4.2 Adaptation Approach

The adaptation approach was already introduced, now we will give a more detailed description. The evolution of contracts is supported by lightweight proxy's capable of dynamically adapting messages exchanged between services to match them with the static service code. To support this mechanism we will need tree ingredients, a contract description language, a compatibility verification and adaptation protocol.

Contract Description Language Contracts in web services can be described 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 aim to extend the OpenAPI specification to incorporate support for the needs of our approach.

We will use the Json format to represent records in messages. Json lacks a language for describing schemas, however, the OpenAPI specification supports the description of record schemas in conjunction with the signature of HTTP endpoints.

Compatibility Verification Intuitively, compatibility verification determines whether all the edited elements in the system's modules are compatible with the ones effectively used by the remaining modules, and whether the new modules' requirements are met by the system's existing resources. An HTTP contract contains the HTTP method, the path, the parameter schema, and the location of parameters (path, query, header); the proposed compatibility verification supports all of these elements. Two approaches will be presented for the verification of contracts:

- **Approach A.** In a simplified approach, instead of verifying if a new contract version is compatible with all consumer references the verification process determines only if the new contract is compatible with the previous versions of the producer.

If a new contract contains an endpoint that is incompatible with a previous version, all consumers of that version, even if they do not use the incompatible endpoint, are unable to use the new version.

The advantage of this approach is that consumer references don't need to be described and documented; the verification only processes the producer's new contract and its previous versions.

The disadvantage of this approach is that there are deployments that are deemed unsafe despite being safe. One way to reduce false negatives is to detect unused endpoints through system log analysis and ignore unused endpoints when performing compatibility checks.

- **Approach B.** In an alternative approach, the verification processes both the producer contract and all the consumer references in each endpoint for assessing the safety of a deployment operation.

The advantage of this approach is that are no false negatives when accessing the safety of deployments.

The disadvantage is that developers will face an additional burden because consumer references will need to be documented.

The compatibility of endpoints and references is typically accomplished via the comparison of element names. Elements with the same name are considered to be the equal. In this case, some ambiguous evolutions can only be solved with human intervention and knowledge of the domain. For example: renaming two fields of the same type; inserting a new field and removing another of the same type, is indistinguishable from renaming a field. To solve this problem without requiring human intervention, all elements must be tagged with an immutable and unique key, and the verification process must use element keys rather than element names. Low-code visual language editors, such as the OutSystems platform [**golovin2017outsystems**], can manage element keys transparently and automatically; however, in the proposed approach, tags must be manually inserted in the WADL definitions.

In the case of solution A. the compatibility verification can be done with human intervention without imposing a significant burden because it involves comparing only the active versions of one producer.

In the case of solution B. the compatibility verification needs to be fully automatic because it involves comparing a new version of a producer, with all consumers.

In approach A the verification process automatically outputs a compatibility file for each pair (N,O) where N is the new contract version, and O is an earlier contract version that is still being consumed. In this file all mappings between elements in each contract are explicitly defined. The file can be audited by a developer before it is used in the adaptation protocol. If the compatibility verification detects an ambiguous case, the developer is notified, and the system only continues after the ambiguity is resolved.

Approach A has an additional advantage over approach B in that it allows for more complex evolution types that can be facilitated with human intervention. Unlike approach B, which only supports the removal, addition, and renaming of fields, approach A can accommodate complex contract changes, such as changing the format of a date, through the use of user-defined adaption functions.

`e.g. new_contract.startdate = formatDate(earlier_contract.startdate, format)`

These functions are used in the compatibility file. Their definition can be provided thorough a static library, or it can be contained within the compatibility file and then executed by mobile code.

This study is leaning toward solution A because, while it produces false negatives, it is a more transparent and flexible approach.

Adaptation Protocol The protocol haves two responsibilities, assuring the safety of deployment operations and adapting messages so that they conform with the static service code.

The safety of deployment operations is assured with the help of a service registry that provides a procedure that employs the compatibility verification described above when a deployment operation is performed. If the compatibility verification fails, the deployment operation is halted. To simplify the compatibility verification mechanism, the service registry stores all service contracts and dependencies. The contracts of services are inserted in the registry when the service is successfully deployed, and removed when the service is discontinued.

The adaptation of messages is supported at runtime by a generated proxy component that dynamically adapts the data exchanged between services. The proxy adapter intercepts and evaluates all the outgoing and incoming TCP requests to determine whether they should be left unchanged or how they should be transformed. The proxy will be installed in both consumers and producers;

When exchanging messages internally the adaption protocol is performed on proxy's installed on consumers, to avoid centralizing the load of message adaptation in a single point.

When receiving requests from web clients the adaption protocol is performed on the proxy adapter installed in the producer, in order to keep the endpoints usable by both external and internal clients, while also keeping the evolution of the contract independent.

The proxy creation mechanism makes use of the available meta-information present in the service registry to automatically update the proxy components as needed. Services are deployed without a proxy, proxies are installed on demand using a lazy instantiation method, that creates them on the first communication between services.

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.

4.3 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 4.1: 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 4.1, 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 approaches proposed to be evaluated in comparison to our suggested solution are schema resolution rules, chain-adapter pattern, and traditional versioning. Each solution will be evaluated in terms of granularity, terminal, type, scalability, maintainability and performance.

The architecture of the benchmark platform can be seen in section 4.2. 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.

Argo [4] is a robust pipeline engine for Kubernetes, It provides simple, flexible mechanisms for specifying constraints between tasks and for linking the output of any task as an input to subsequent task. The argo framework will be used to specify the evolution of services through tasks in an argo workflow, as well as to manage active virtual users by deploying and stopping docker images that contain load testing scripts. Complex experiments can be made with Argo since the state of each deployment in Kubernetes can be queried via a task and utilized as input in a decision that leads to different tasks.

API.guro [3] is repository of applications Web APIs written in the OpenAPI specification. The experiments will use the apis available in this repository as a datasets.

OpenAi Generator [5] is a tool for generating API client libraries, server stubs, configurations, and documentation from OpenAPI documents. It will be used to enforce the conformance of tested solution implementations to their OpenAPI contracts.

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.

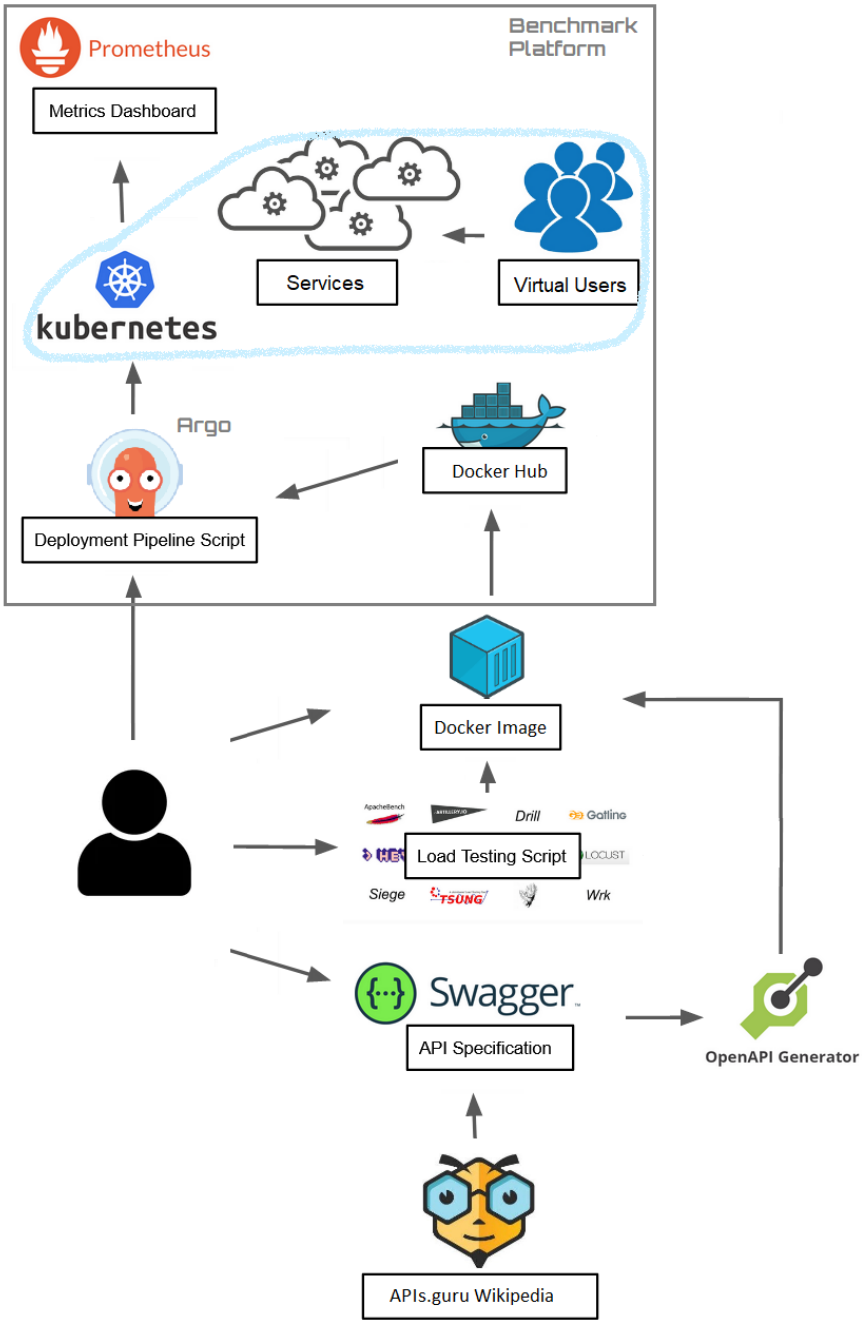

Figure 4.2: Benchmark platform components

5

Work Plan

We have divided the work to be done into the following tasks, which are depicted in Figure 5.1:

- **Development of the compatibility verification procedure:** This task consists on the implementation of the compatibility verification procedure that compares OpenAPI contracts.
- **Development of the service registry:** This task entails designing and implementing the service registry.
- **Development of the adapter proxy:** This task entails designing and implementing the adapter proxy.
- **Development of the benchmark platform:** This task involves writing load testing, argo, prometheus, and kubernetes scripts to support the benchmark platform, as well as a bash script that acts as an interface to the platform.
- **Development of the benchmark target solutions:** This task consists on the development of the various solutions to the evolution of contracts in microservices.
- **Experimental evaluation:** This will consist on the setup of kubernetes cluster and the evaluation of each solution granularity, terminal, type, scalability, maintainability and performance via the benchmark platform and empirical evidence.
- **Writing of the dissertation document:** This task involves on the continuous writing of the dissertation, which will be done in parallel with the other activities.

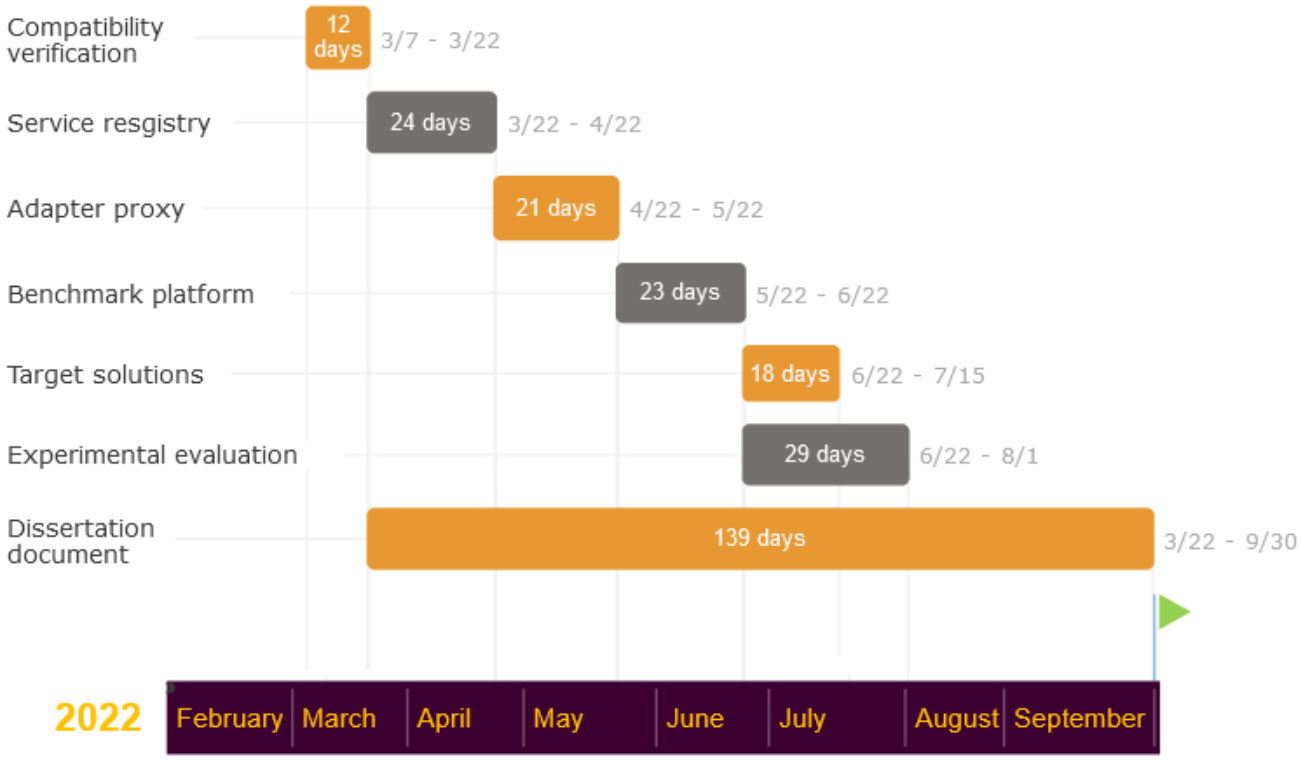

Figure 5.1: Proposed thesis work plan.

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. 25).

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

[3] A. P. Authors. *APIs.guru*. https://apis.guru/ (cit. on p. 26).

[4] A. P. Authors. *Argo*. https://argoproj.github.io/argo-workflows/ (cit. on p. 26).

[5] O. G. P. Authors. *OpenAPI Generator*. https://github.com/OpenAPITools/openapi-generator (cit. on p. 26).

[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. 19).

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

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

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

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

[11] J. 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, 5).

[12] G. Developers. *Protocol Buffers*. http://code.google.com/apis/protocolbuffers/ (cit. on p. 14).

[13] O. Developers. *Java RMI*. https://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmiTOC.html (cit. on p. 15).

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

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

[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. 12).

[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. 13).

[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. 17).

[19] M. Kleppmann. *Schema evolution in avro, protocol buffers and thrift*. 2013 (cit. on p. 16).

[20] J. L. Martin Fowler. *Microservices*. http://martinfowler.com/articles/microservices.html. 2014 (cit. on pp. 1, 2, 5–7, 19).

[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. 15).

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

[23] M. P. Papazoglou et al. "Service-oriented computing: State of the art and research challenges". In: *Computer* 40.11 (2007), pp. 38–45 (cit. on p. 1).

[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, 19).

[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. 12).

[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. 18).

[28] G. Sayfan. *Mastering kubernetes*. Packt Publishing Ltd, 2017 (cit. on pp. 8, 26).

[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. 9–11, 18).

[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. 14).

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

[32] J. Turnbull. *Monitoring with Prometheus*. Turnbull Press, 2018 (cit. on p. 27).

[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. 13).

[34] D. Vohra. "Apache avro". In: *Practical Hadoop Ecosystem*. Springer, 2016, pp. 303–323 (cit. on p. 14).

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