João Miguel Tomás Ferrão Correia Perdigão

A Software Architecture for Highly-Available Cloud-Native Applications

1 2 9 0 UNIVERSIDADE D COIMBRA

UNIVERSIDADE D
COIMBRA

João Miguel Tomás Ferrão Correia Perdigão

# A Software Architecture for Highly Available Cloud-Native Applications

**Dissertation in the context of the Master in Informatics Engineering, Specialization in Software Engineering, advised by Professor Raul Barbosa and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.**

September 2021

Faculty of Sciences and Technology

Department of Informatics Engineering

# A Software Architecture for Highly Available Cloud-Native Applications

João Miguel Tomás Ferrão Correia Perdigão

Dissertation in the context of the Master in Informatics Engineering, Specialization in Software Engineering advised by Professor Raul Barbosa and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

September 2021

UNIVERSIDADE D
COIMBRA

This page is intentionally left blank.

## Abstract

Cloud-native applications are becoming the *de facto* approach for building systems. With their heterogeneity, increased size and dynamic nature new challenges arise to ensure important requirements such as their availability. In a scenario of business-critical applications, where an outage of a single minute has significant economical impacts, making sure that the system remains available is mandatory. Hence, autonomic computing, a paradigm introduced by IBM, intends to find better ways to manage systems, while reducing the cost to maintain them. More precisely, and among the self-managing properties it suggests, we are particularly interested in self-healing capabilities, *i.e.*, identifying and recovering from failures or bugs (and repairing whenever possible) without human intervention. In this thesis, we propose a solution through the implementation of an autonomic framework capable of detecting failures in the managed system and applying mitigation plans accordingly, to recover from failures. A Publish-Subscribe middleware is used as the underlying infrastructure for communication among components. With this approach, we intend to provide better availability for cloud-native applications whenever subject to failures. The proposed solution was evaluated under different failure scenarios and considering distinct recovery actions and strategies. The outcome of the experiments shows that the framework developed is capable of effectively recovering from failures, while achieving the desired performance. These results represent a step further to guarantee high availability of cloud-native applications.

## Keywords

Microservices, Autonomic Computing, MAPE-K Loop, Fault Tolerance, Cloud Computing

This page is intentionally left blank.

## Resumo

As aplicações nativas da *cloud* estão a tornar-se na abordagem aceite para construir sistemas. Com a sua heterogeneidade, aumento crescente de tamanho e natureza dinâmica, surgem novos desafios de forma a assegurar requisitos importantes, tais como a sua disponibilidade. Nos cenários de aplicações críticas para o negócio, no qual uma interrupção do seu funcionamento que dure um minuto tem impactos financeiros significativos, é imperativo garantir que o sistema permanece disponível. Deste modo, a computação autonómica, um paradigma introduzido pela International Business Machines Corporation (IBM), pretende encontrar maneiras mais adequadas de gerir sistemas, enquanto reduzindo o custo da sua manutenção. Mais precisamente, e entre as propriedades de auto-gestão sugeridas, estamos particularmente interessados nas capacidades de "auto-cura", isto é, identificar e recuperar de falhas ou *bugs* (e reparar sempre que possível) sem intervação humana. Nesta tese, propomos uma solução através da implementação de uma *framework* autonómica capaz de detetar avarias no sistema gerido e aplicar planos de mitigação de acordo com as mesmas, de forma a recuperar das avarias. Um *middleware* Publish-Subscribe é utilizado como infraestrutura adjancente para comunicação entre componentes. Com esta abordagem pretendemos fornecer melhor disponibilidade a aplicações nativas da *cloud* quando estas forem sujeitas a falhas. A solução proposta foi avaliada sob diferentes cenários de avarias, tendo sido consideradas ações e estratégias de recuperação distintas. O resultado das experiências demonstra que a *framework* desenvolvida é capaz de recuperar eficazmente de avarias, enquanto garantindo o desempenho desejado. Estes resultados representam um avanço na garantia de alta disponibilidade de aplicações nativas da *cloud*.



## Palavras-Chave

Microsserviços, Computação Autonómica, Ciclo MAPE-K, Tolerância a Falhas, Computanção na Nuvem

This page is intentionally left blank.

# Acknowledgements

First and foremost, I want to express my gratitude to my supervisors. Professor Raul Barbosa always provided the best feedback and was always open to debate new ideas throughout our discussions. His continuous support, ambitious ideas and vast wisdom were fundamental for the success of this work. Engineer André Bento, also had a core role while co-supervisor of my thesis, being always available to help with his immense technical knowledge and constructive suggestions.

I would also like to express my grateful thanks to professor Filipe Araújo and Engineer Jaime Correia for their advice which helped improve the work performed throughout this thesis. An additional thank you must be sent to the staff at Fiercely, namely António Howcroft, Luís Ribeiro and João Soares for their input and insightful ideas and reviews.

Faculdade de Ciências e Tecnologia da Universidade de Coimbra (FCTUC) was my home for the last five years, so a special thank you must be sent to everyone which helped me become the professional that I am today.

I would also like to thank the National Institute of Distributed Computing (INCD), which provided the computational resources that had a core importance for the experimental work, and the Project Autonomic Service Operation (AESOP), P2020-31/SI/2017, No. 040004., for supporting this thesis.

Finally, a warm thank you must be sent to my mother and my father which always cared for my well-being and education. I must also express my very profound gratitude to my family, for always supporting me, my girlfriend Marília, who was always by my side and provided unfailing support and encouragement, and my friends, with whom I had the pleasure to work with and share this journey.

This page is intentionally left blank.

# Contents

This page is intentionally left blank.

# Acronyms

**AESOP** Autonomic Service Operation. vii

**API** Application Programming Interface. 23

**AWS** Amazon Web Services. 20

**CI** Continuous Integration. 8

**CI/CD** Continuous Integration/Continuous Delivery. 8

**CNA** Cloud-Native Application. 10

**CNCF** Cloud Native Computing Foundation. 10

**CPU** Central Processing Unit. 35

**ECA** event-condition-action. 12

**FCTUC** Faculdade de Ciências e Tecnologia da Universidade de Coimbra. vii

**FMEA** Failure Mode and Effect Analysis. 17

**HTTP** Hypertext Transfer Protocol. 1, 10, 17, 19

**IBM** International Business Machines Corporation. iii, v, 1

**INCD** National Institute of Distributed Computing. vii

**JSON** JavaScript Object Notation. 25, 28, 30, 32, 35, 37, 39

**MAPE-K** Monitor-Analyze-Plan-Execute over a shared Knowledge. 2

**MEPFL** Microservice Error Prediction and Fault Localization. 16

**MTBF** Mean Time Between Failures. 3, 73

**MTTD** Mean Time To Detect. 62

**MTTR** Mean Time To Recovery. 3, 59, 62, 65

**RPC** Remote Procedure Call. 23

**RQ** Research Question. 2

**SDK** Software Development Kit. 23

**SLA** Service Level Agreement. 1

**SOA** Service-Oriented Architecture. 8

This page is intentionally left blank.

# List of Figures

This page is intentionally left blank.

# List of Tables

This page is intentionally left blank.

# Listings

This page is intentionally left blank.

# Chapter 1

# Introduction

## 1.1 Context

With the ever-growing size of computer systems, new approaches to assure important quality attributes such as availability, performance and scalability are required. A very popular approach in cloud computing are microservices [68]. These decouple the monolith into functionally independent services, which perform simple tasks and communicate through lightweight protocols such as Hypertext Transfer Protocol (HTTP). The huge amount of complexity and dynamic nature of such systems makes it more difficult to manage and monitor them, thus surpassing human cognitive capabilities.

To help self manage systems in an economic-effective manner, IBM proposed in 2001 a paradigm for self-management of computer systems [55], resembling the autonomic capabilities of the human body. This paradigm includes *self-healing* to help detect, diagnose and repair problems, *self-optimization* for automatic configuration of the system while making it more cost effective, *self-configuration* according to high-level policies defined by the system's administrators and *self-protection* against attacks or failures. A self-manage or self-adaptive system would usually consist in a feedback loop to deal with adaptation aspects of the managed system. The feedback loop proposed by IBM is the MAPE-K loop [55, 45, 99], which consists of several entities, namely Monitor(M), Analyse(A), Plan(P), Execute(E) over a shared Knowledge(K), as explained in Section 2.2.

However, the implementation of such paradigm still poses a challenge for researchers and developers and not many implementations exist that can fulfill all the characteristics of such systems. Nonetheless, it is crucial and very needed to address their implementation, to help keep applications available.

## 1.2 Motivation

With many enterprises adopting a cloud-first model, it is crucial for business-critical applications to guarantee the availability of their systems while maintaining acceptable management costs and honoring any Service Level Agreement (SLA) defined. However, such applications usually consist of large-scale systems, with complex request flows and runtime behavior. These characteristics make it harder to monitor failures and optimize performance only with human interaction. Nonetheless, not only the systems increase their size and complexity, but the same applies to their corresponding failures, and systems

lack effective fault tolerance and recovery strategies to handle them [31]. Furthermore, some faults can only be detected in runtime environments, thus the importance of failure injection techniques such as chaos engineering.

Hence, it is imperative that systems start providing self-managing capabilities in a cost-effective manner to cope with such complex environments. This thesis proposes a solution to detect and recover from failures without human intervention, to help guarantee system's availability.

## 1.3 Goals

The objectives of this thesis can be split in two different goals:

- Development of a framework with autonomic capabilities, intended to maintain a cloud-native application available, through the enforcement of self-healing actions, namely detection of faults and error recovery. The error recovery is performed through a set of actions defined in mitigation plans to be applied to the system upon the detection of a failure. A Publish-Subscribe middleware constitutes the core innovation of this implementation. It will help in the process of information exchange among the fault detection and recovery components, whose information will be published to and consumed from topics. This solution is documented in further detail in Chapter 3.
- Validation of the framework resorting to fault injection techniques. Knowing that the system must issue recovery actions according to the detected faults, a set of faults will be injected to assess the framework's behavior. This evaluation is present in Chapter 4.

## 1.4 Research Questions

The present section depicts the research questions that will be assessed in the context of this work:

RQ1. Study the feasibility of building a MAPE-K loop using a Publish-Subscribe middleware.

RQ2. Evaluate the possibility of achieving an availability of 99.99% for a cloud-native application with the self-healing capabilities provided by our framework.

RQ3. Evaluate the effectiveness of different recovery strategies as a means to complement single recovery actions.

The first research question (RQ1) is intended to study the feasibility of using a Publish-Subscribe middleware as a fast and reliable middleware for inter-component communication. Publish-Subscribe is a type of asynchronous communication among services, where messages are sent to and consumed from topics. These are places where messages are stored and where a publisher can publish a message and all the subscribed consumers will receive it. In the context of the present work, we will study the feasibility of Apache Kafka [4] as the chosen middleware, according to the results obtained from the experiments performed

after finishing the development of the framework. This was the chosen technology due to its performance, scalability, durability and fault tolerance attributes.

Regarding RQ2, it is concerned with the ability of the framework to maintain a cloud-native application available 99.99% of the time. Availability is the "readiness to provide correct service" [8]. It can be measured resorting to the following formula:

$$Availability = \left( \frac{MTBF}{MTBF + MTTR} \right) \times 100$$

Figure 1.1: Formula to evaluate the availability of a system

MTBF is the Mean Time Between Failures and is the average time between repairable failures of a system. The Mean Time To Recovery (MTTR) is the average time taken to recover from outages. For an application to maintain an availability of 99.99%, its yearly downtime (period when the system is not providing service) cannot be greater than 52 minutes and 35 seconds. Thus, if we have outages whose error detection and recovery takes between 1 to 5 minutes, we can withstand several of those while still maintaining the high availability desired, being the downtime considered a reasonable value for a business-critical application.

Finally, RQ3 is related to evaluating the effectiveness of different recovery strategies under distinct failure scenarios. Recovery actions, by themselves, are a candidate approach to restore the system's correct service. However, since there is no *silver bullet* to mitigate failures due to their diversity and complexity, another possible solution is to combine different recovery actions into a recovery strategy. This final research question addresses this approach and intends to evaluate the effectiveness of the strategies in the different failure scenarios considered.

## 1.5 Workplan

The work developed within the scope of this thesis encompasses two semesters. The initial plan was to devote the first semester to research about topics and core concepts related to the scope of this work, as well as define the requirements and architecture of the solution and how it must be validated. In the second semester, the implementation must take place, as well as the validation of the solution and answering to the research questions. According to the proposed hours that must be dedicated to the Master Thesis, in the first semester the student is supposed to devote 16 hours each week, resulting in a total of 320 hours (20 weeks $\times$16 hours). In which concerns the second semester, there are 800 hours of estimated work, resulting from 40 hours of work throughout 20 weeks (see Figure 1.2).

In the first semester, the work went according to schedule. It was focused on researching about the topics related to the research subject, which allowed to compose the state of the art. Meetings took place every week to clarify any doubts, discuss some relevant ideas about the project and plan work to be done until the next meeting. In the beginning of the semester, the scope of the project was defined. Likewise, the requirements of the framework being developed and its architecture were also defined. Since the component interacts with a Publish-Subscribe middleware, namely Apache Kafka [4], some time was devoted to the configuration of this middleware inside a Kubernetes cluster. To have a better understanding of the work that was carried out, Figure 1.3 presents the Gantt chart

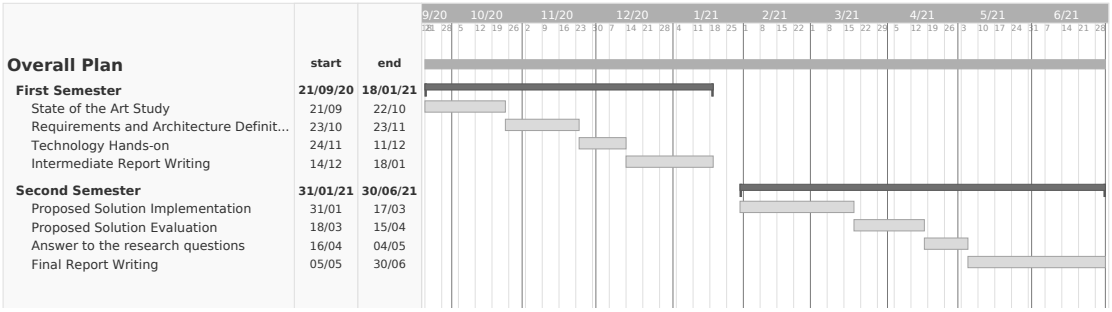


Figure 1.2: Overall Plan

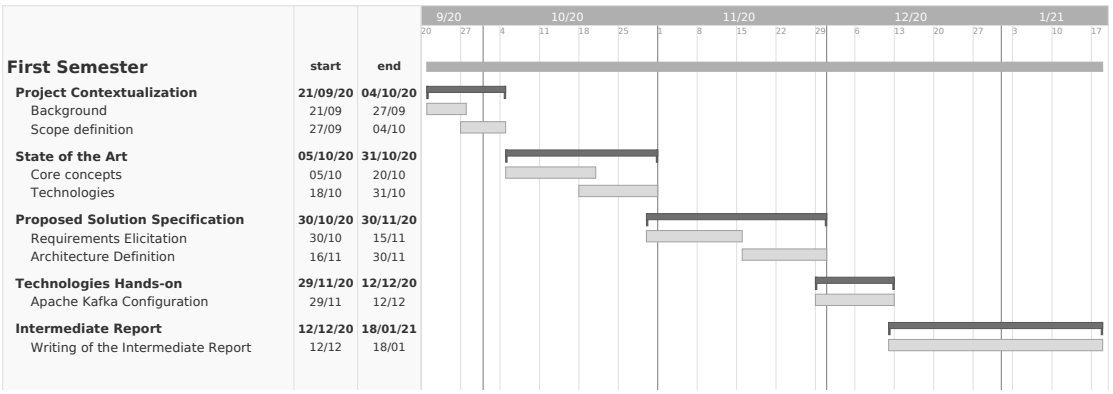


Figure 1.3: Work performed in the first semester

of the first semester.

Regarding the second semester, Figure 1.4 presents its work estimation. This estimation was performed based on the Fibonacci Agile Estimation. This technique allows a rational estimation of the effort required for each task to be done, resorting to the Fibonacci sequence.

The real work performed in the second semester, which is exhibited in Figure 1.5 showcases a few key changes when comparing with the estimation performed. In order to support the development and testing of the proposed solution, additional components needed to be developed. These include *StreamTrap*, which reconstructs traces from individual spans, *Fault Detection System*, in charge of analysing incoming traces and detecting failures, and the *Executor*, which receives the mitigation plans and applies the recovery actions to the system. Another change was that before implementing the "streaming" solution, which operates in the intended feedback loop, an "offline" version was developed to facilitate testing. Additionally, a research paper was written for the *2nd Workshop on Dynamic Risk Management for Autonomous Systems (DREAMS 2021)*. Finally, besides developing and assessing the performance and effectiveness of different recovery actions, recovery strategies were also developed and their effectiveness was evaluated.

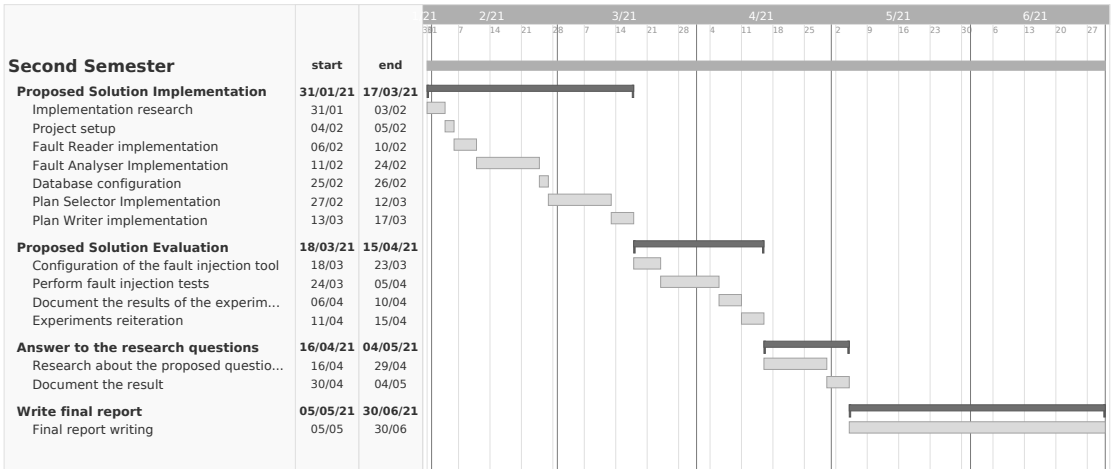


Figure 1.4: Second Semester Plan

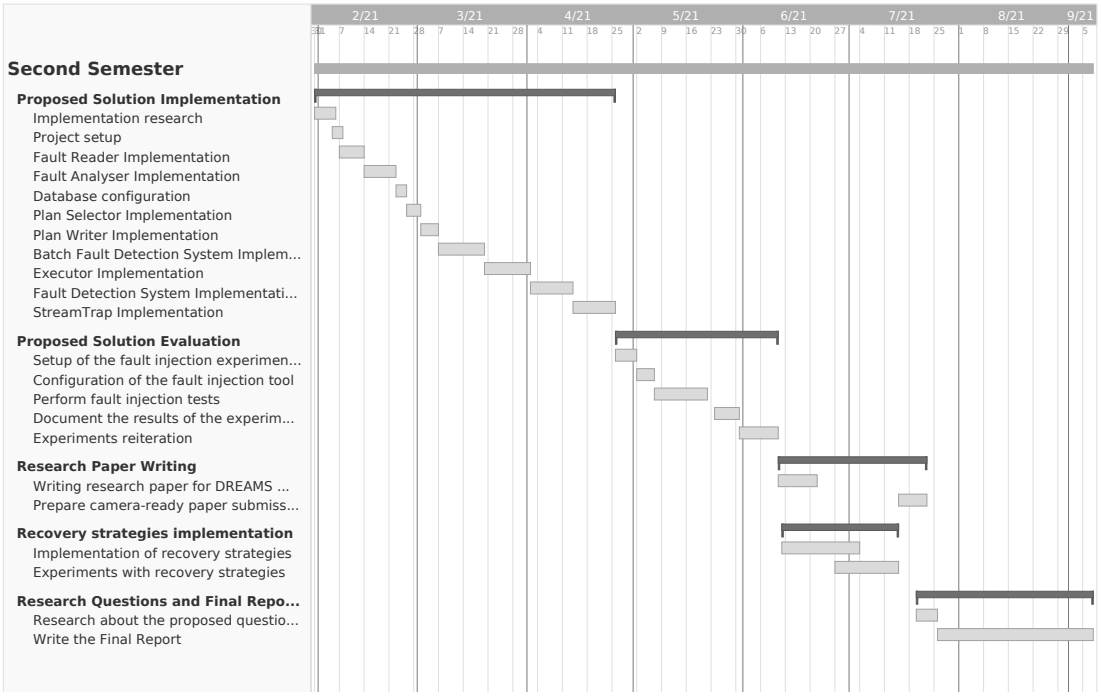


Figure 1.5: Work performed in the second semester

## 1.6 Research Contributions

The work performed throughout this thesis converged in a research paper to be presented at the *2nd Workshop on Dynamic Risk Management for Autonomous Systems (DREAMS 2021)*. This contribution is entitled as follows:

- João Tomás, André Bento, João Soares, Luís Ribeiro, António Ferreira, Rita Carreira, Filipe Araújo, Raul Barbosa. *Autonomic service operation for cloud applications: Safe actuation and risk management*. 17th European Dependable Computing Conference (EDCC 2021).

## 1.7 Document Structure

This section aims to present the structure of this document, which is organised as follows:

- Chapter 2 presents theoretical background and core concepts that help understand the work being performed. Related work in the field is also documented in order to assess the existing work and raise a critical overview of possible improvements in the system to be developed.
- Chapter 3 documents the proposed solution. It includes both the functional and non-functional requirements, as well as its architecture. The data model used for the mitigation plans is also documented and the validation of the non-functional requirements is performed. A detailed description of the recovery actions and strategies considered is also performed.
- Chapter 4 explains how the proposed solution was evaluated, through the use of fault injection techniques. A detailed description of the experimental runs is provided, which includes the faultload and workload used. Furthermore, the infrastructure where the experiments are executed is also described.
- Chapter 5 provides a detailed description of the proposed architecture's implementation. The chapter is split into several sections, each of them describing one of the components from the architecture.
- Chapter 6 exhibits and interprets the results obtained throughout the experimental runs. Moreover, the answers to the research questions are also provided.
- Chapter 7 summarizes the research subject, how the proposed solution helps solve the documented problems and the conclusions obtained from the present work A brief discussion of the experimental results is also provided, as well as future work to help feed research directions to complement and improve the proposed solution.

This page is intentionally left blank.

# Chapter 2

# Background

This Chapter pinpoints the main concepts that are related to the scope of this thesis, the technology which implements those concepts and related work in the field. The information gathered results from literature review, documentation and exchange of ideas from the meetings.

## 2.1 Microservices

With the increasing adoption of cloud computing and with many enterprises migrating their applications to the cloud or intending to do so in the following years, there is the necessity for a suitable architectural style and that is where microservices emerge. Some research has already been performed about the concerns of migrating from a monolith architecture to a microservices one, in order to achieve the fast pace of delivery, scalability and availability features that this design pattern provides [59].

The previously mentioned **monolith** was the traditional approach to software development where we would witness all the functionalities of a piece of software deployed into a single application (Figure 2.1). This was only feasible when dealing with a small application, even showing some strengths such as being easy to develop, test and deploy [59]. However, when the size and complexity of an application starts to increase, some problems arise. For instance, if there is an increase in load, we are forced to scale the whole application. The same applies if a bug is found, since all the application may be affected due to the code being gathered in one place, creating a single point of failure.

That said, the need to decouple the components of an application arose, leading us to the **microservices** architectural style (Figure 2.1). These can be defined as "an approach to developing single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms" [30], following the principle of "small and focused on doing one thing well" [68].

An important aspect that must be noticed is that microservices can be seen as an improvement over **Service-Oriented Architecture (SOA)**. These had as main objective "the integration of different software assets, possibly from different organizations, in order to orchestrate business processes" [38]. In which concerns microservices, these are more focused on improving the development, delivery and deployment and are seen as an enabler for surfacing software development practices such as DevOps and Continuous Integration/Continuous Delivery (CI/CD). Another advantage, related to Continuous In-

tegration (CI), is the possibility for new versions of components to be gradually introduced in a system through the side by side development with previous versions [65]. Besides that, service-oriented architectures focused on sharing as much as possible, while microservices encourage in decoupling services and sharing as little as possible [32].

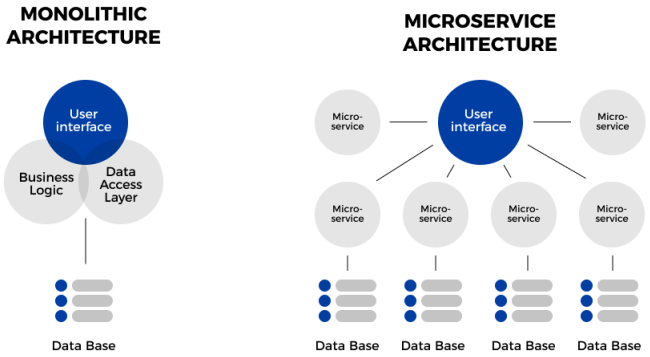


Figure 2.1: Comparison between monolith and microservices. Figure from [21]

Among all the benefits of microservices, we can identify [59, 68, 80]:

- **Technology Heterogeneity** - developers are not attached to using the same technology for all the services that compose an application. Contrariwise, we have the possibility to have each single service written in a different programming language.
- **Scaling** - when a service is under heavy load, it can be scaled individually, without the need to scale any other services.
- **Resilience** - if an instance of a service fails, the fault can be isolated and the remaining services can keep working (unless we are dealing with a cascading failure, which occurs when a failing service has other services that depend on it, which in turn can become unresponsive [65]).
- **Ease of Deployment** - when we need to deploy changes to a certain service, this can be done independently to the desired service, without affecting the remaining ones.
- **Replaceability** - due to their small size, it is easier to replace an existing service with another with a better implementation or even delete it if it is no longer used.
- **Composability** - due to the decoupling of microservices, it is easier to change our application if the business demands change, which is something that would be very hard to accomplish with a monolithic architecture.
- **DevOps** - enables a "collaborative symbiosis" between developers and operators, having one team working on the architecture and development phases. This results in improved quality and faster delivery to production.

Notwithstanding, there are also some issues that arise from using this kind of architecture:

- **Monitoring** - having such a fine-grained system makes it harder to pinpoint where a certain failure occurred and what caused it. This brings the need to use new

monitoring solutions such as **distributed tracing** (see Section 2.5), since logging becomes unsuitable when we are dealing with such distributed systems.

- **Complexity** - development and production become more complex due to provisioning and orchestration of services.
- **Testing** - new testing approaches are required due to the dynamic and distributed nature of microservices.
- **Communication overhead** - services communicating over protocols such as HTTP can result in high latency.

A widely adopted concept nowadays is the one of **Cloud-Native Application (CNA)**. This is the name used to refer to an application that is "specifically designed to provide a consistent development and automated management experience across private, public and hybrid clouds"[37]. The development of cloud-native applications helps speed up how they are built, optimize them and connect all of them. This way it is possible to deliver applications that the user wants at the pace a business needs [37]. According to CNCF [29], their objective is to drive adoption of this paradigm by serving and sustaining an ecosystem of open source and vendor-neutral projects, thus helping innovations become accessible to everyone.

Naturally, to allow this type of software architecture to be developed, specific infrastructure technologies are required, which allow to deploy, manage and control distributed services and have basic self managing properties such as horizontal scaling and restarting pods that failed. Among these technologies we can identify *Docker* [96], *Docker Swarm* [22], *Kubernetes* [56], *Apache Mesos* [5] and *Amazon Elastic Container Service* [84].

In the scope of this thesis there is a special interest in Docker and Kubernetes since these are some of the technologies to be used in the framework development. **Docker** is a widely used container-based technology that allows to build, ship, and run distributed applications [96]. These containers are a lightweight approach which many prefer in contrast to hypervisor-based virtualization [16]. The latter type of virtualization works at the hardware-level, with an hypervisor establishing complete virtual machines on top of the host operating system. These machines contain not only the application and its dependencies, but also the entire guest operating system [16].

To help manage containers we may resort to orchestrators such as *Docker Swarm*, *Kubernetes* and *Apache Mesos*. An orchestrator can be seen as technology that helps manage containers and automate tasks such as their deployment and scaling. One of the best known orchestrators is **Kubernetes**. It is an open source platform that allows to manage Docker containers in a cluster. Some of its features include healing by automatically restarting pods that failed and rescheduling when their hosts die [92]. A Kubernetes cluster is composed of at least one master node [74]. The master node is in charge of managing and controlling the worker nodes. This node is usually tainted with a policy of NoSchedule, to avoid The pods are hosted in the worker nodes and are the smallest deployable units, which can have one or more containers [58, 77].

Since our intention is to build a cloud-native application, we must consider the usage of a **service mesh**, which can be defined as a "dedicated infrastructure for handling service-to-service communication" [83]. These are usually implemented as an array of lightweight proxies attached to containers in a sidecar pattern which allow reliable message delivery and hold a set of important techniques such as service discovery, circuit-breaking and load balancing. Sidecars interact with the service mesh control plane, which is in charge of

managing and configuring the proxies to route traffic. These proxies can be individually configured with timeouts or retries if the associated microservice does not receive a timely response or to retry the request. Mechanisms such as the previously mentioned circuit-breaker can prevent cascading failures, where a failure in a microservice can propagate to subsequent services. **Circuit-breaker** is a design pattern that, whenever a certain number of requests to a downstream service fail, stops all further requests (fail fast) to that service. After a certain period of time, the client can send requests to assess if the downstream service has recovered. If this happens, the circuit breaker resets and the request flow is restored [68]. Nonetheless, there are trade-offs that should be considered when configuring mechanisms such as the circuit-breaker. If the circuit breaker decides too quickly that a service has failed, it can unnecessarily abort requests, which will affect the application's availability. Furthermore, if the decision is made too slowly, this failure can propagate to other services (cascading failure), which will also impact the application's availability [50]. Thus, we can conclude that the adoption of a service mesh in the technology stack is critical for applications that can have thousands services, each of them with hundreds of instances that dynamically change their state.

Nonetheless, with the increasing complexity of systems we need more than ever the existence of self-adapting capabilities upon different situations, without the need for human intervention.

## 2.2 Autonomic systems

The concept of Autonomic Computing was first introduced by IBM in 2001 [40]. Foreseeing the increasing complexity of software systems, which are becoming more interconnected and diverse, they envisioned the necessity to start developing "computing systems that can manage themselves given high-level objectives from administrators" [55], which can be defined as **autonomic computing**. The main objectives are to reduce the system's ownership cost and to find better ways of managing their increasing complexity [87]. This paradigm is influenced by the human body's autonomic system, which regulates vital body functions such as the heart rate without conscious effort [87].

The core idea for this kind of systems is self-management, which would allow the system to monitor its own usage and do its own maintenance when there is the need to. Inside self-management there are **four aspects** that we must consider [55]:

- **Self-configuration** - the ability of the system to configure itself according to high level policies. Based on that information, it must adapt automatically.
- **Self-optimization** - intended to eradicate the concern of tuning the parameters of software systems and continuously trying to improve their operation, thus becoming more performance and cost efficient.
- **Self-healing** - focused on the detection, diagnose and repair of problems resulting from bugs or failures, on its own.
- **Self-protection** - capable of protecting the system in two ways, either as a whole against malicious attacks or cascading failures, or prevent problems ahead of their occurrence, based on previous information.

That said, a self-adaptive system is usually composed of a **feedback loop** and a **managed system**. The managed system is a component (hardware or software) that can

be managed, such as an application. The feedback loop handles the adaptation of the managed system, which includes, for instance, changing the system's configuration when it is under heavy load [45].

The feedback loop, proposed by IBM, is a structure referred to as **MAPE-K loop** (or just **MAPE-K**). It is composed of the **M**onitor, **A**nalyse, **P**lan and **E**xecute entities, alongside with runtime models (**K**nowledge) of the managed system, its environment and the adaptation objectives provided by the system administrators [45]. A runtime model can be described as an abstraction of a system that allows to reason about it during its execution [14]. An example of a runtime model is software architecture runtime model, which are graph-based data structures that store configurational and operational information. The model is sustained through system monitoring, in order to echo in the model any changes that may occur in the system, and help the system to self-adapt [14].

Each of these entities is described as follows [45, 99, 35]:

- **Monitor (M)** - gathers data from the managed system and the environment, with the help of probes, to update the Knowledge. The data sent to the Knowledge may be subject to preprocessing tasks, such as filtering and aggregating the data received.
- **Analyse (A)** - assesses if there is the need to perform any adaption action, based on the monitored state of the managed system, the environment and the adaption concern of interest.
- **Plan (P)** - is in charge of planning the mitigation actions to adapt the managed system. These actions can range from a single command to a more complex workflow.The Plan component will then notify the Execute in order for the latter to execute the adaptation actions on the system.
- **Execute (E)** - is responsible for performing the adaption actions derived from the mitigation plans.
- **Knowledge (K)** - is intended to manage different knowledge sources for supporting the feedback loop's operation, thus managing the knowledge that other components may need for their operations. Examples of shared knowledge include log files, system metrics and topology information. A knowledge source is an implementation of a repository (*e.g.*, a database) that stores knowledge, which can be shared among autonomic managers and accessed by the entities of the autonomic manager (monitor, analyse, plan and execute) [35]. To clarify, an autonomic manager is a component that manages other components using a control loop. Such control loop includes monitor, analyse, plan and execute functions [35].

As shown in Figure 2.2, the sensors and actuators are connected to the system to be managed. These can also be called touchpoints, since they act as an interface to the managed resource [44]. The sensors are managed by the Monitor entity and are in charge of gathering contextual data which will be later analysed to perform adaptation decisions [99]. In turn, the actuators (or effectors) are used by the Execute entity to perform changes to the managed element.

Besides the overall autonomic functionalities of this systems, they will still depend on humans to provide them **policies**. The policies are the goals and constraints that govern the system's actions. These can be expressed using event-condition-action (ECA) policies, goal policies or utility function policies. ECA policies are usually presented in the format "when an event occurs and condition holds, then execute action". Goal policies

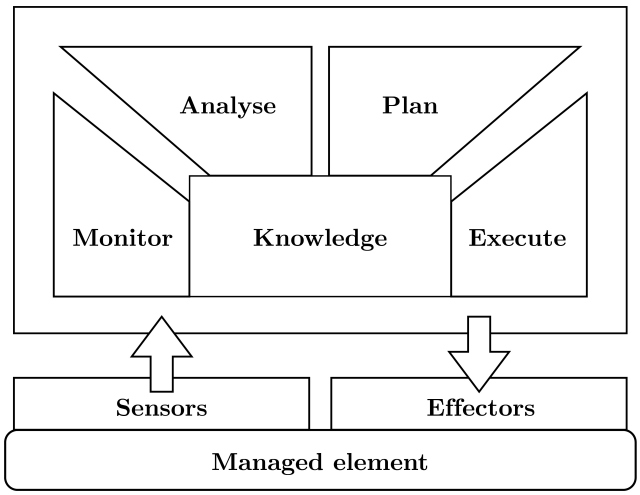


Figure 2.2: MAPE-K loop structure [55]

are described in a higher level since they specify criteria that characterise desirable states, but leave the system in charge of finding how to achieve that state (we can specify that a service should have a response time of under 2 seconds and the autonomic manager will use internal rules to scale resources, in order to attain the desirable state). Lastly, utility functions define a quantitative level of desirability to each state, taking as input a number of parameters and returning the desirability of that state as output [43].

To define such policies, a **manual manager** can be used. This manager provides a system management interface that allows system administrators to perform tasks that these delegate through a management console.They can choose which tasks involve human intervention and which should be performed autonomically, according to policies defined by them [35].

Furthermore, these kind of systems will help reduce human errors, but if a mistake is made while setting up those policies, the consequences can be greater. Thus, two engineering challenges arise: ensuring that goals are specified correctly in the first place and ensuring that systems behave reasonably even when they are not [55]. Facing these two challenges, Kephart [55] suggests that for the first one, a joint work between psychologists and computer scientists to help simplify and clarify the means by which humans express the intended goals shall take place. In which concerns the second one, this would require an additional layer of robustness in the system to protect itself from erroneous input goals provided by humans.

With regard to the approaches that can be followed to integrate autonomic capabilities into an application, two are considered [28]:

- **internal approach** - the autonomic manager is part of the system to manage.
- **external approach** - the autonomic manager does not depend on the application and communicates with it from the outside.

The solution proposed in this document is part of the **Planning** phase. It will choose mitigation plans according to fault information received from the **Analyse** entity, which will then be sent to a publish-subscribe middleware. These plans will be collected afterwards by the **Execute** and applied to the system. The proposed solution is explained in detail in Chapter 3 and the related work is documented in the present Chapter, in Section

2.6. The method that allows to pinpoint the fault location by the **Analyse** component is **distributed tracing** and will be explained in more detail in Section 2.5.

Since the present work is related to providing self-healing actions to the managed system, we can identify five important properties that **self-healing systems** must have [75, 9]:

1. Monitoring, in runtime, of a given system (or part of it).
2. Detection when a given fault occurs in the system.
3. Diagnosis of the fault, including identification of events.
4. Generation or selection of a given recovery plan.
5. Validation and execution of a given recovery plan.

Thus, self-healing is concerned with fault identification and effective recovery, without loss of data or service, and repairing the system whenever possible. Fault prediction techniques might also be used, allowing to perform a re-configuration process to avoid the previously detected faults or reduce the likelihood of their occurrence [87].

Nonetheless, with the evergrowing complexity of microservices, it is mandatory to pay attention to system's **dependability**. This can be defined as the "ability to deliver service that can justifiable be trusted", being service the system's behavior as it is recognized by its users or other systems [8].

The attributes that compose dependability are availability, reliability, safety, integrity and maintainability. **Availability** is our most valuable quality attribute in the context of this work since the framework under development aims at assuring that cloud-native applications remain available even when in the presence of faults. This attribute can be defined as "readiness for correct service" [8]. Alongside with the previously mentioned quality attributes, there are other two mechanisms that are essential for attaining dependability - the **threats to** and the **means for** dependability. The threats that compose the *fundamental chain* (Figure 2.3) are faults, errors and failures. A *fault* is an anomalous condition of the system which may lead to a failure, an *error* is a human action that causes an incorrect result and a *failure* is an event that arises when the service provided deviates from correct service.

$$... \rightarrow fault \xrightarrow{\text{activation}} error \xrightarrow{\text{propagation}} failure \xrightarrow{\text{causation}} fault \rightarrow ...$$

Figure 2.3: Fundamental chain of dependability and security threats [8]

The chain in Figure 2.3 depicts that a *fault* is activated when an input is applied to a component, causing a dormant fault to become active. After being activated, the fault produces an *error*, which is a result from the computation process that, in turn, may create other errors (error propagation). When the errors that were propagated affect the delivered service, a *failure* occurs, which deviates the system from correct service [8]. Finally, a component failure causes a permanent or transient failure to the system that it belongs to. A *permanent* fault is one whose presence is continuous in time and persists indefinitely or until it is repaired. In turn, a *transient* fault (also known as soft fault) is bounded in time.

The last mechanism are the **means to attain dependability**, which are *fault prevention* (avoid the occurrence of faults), *fault tolerance* (avoid service failures in the presence of faults), *fault removal* (remove a fault to avoid damaging the system's correct service) and *fault forecasting* [8]. The latter is performed through an evaluation of the system's behavior whenever a fault occurs or is activated, in order to classify them according to their failure modes (qualitative evaluation) or in terms of the probability of a fault occurrence (quantitative evaluation).

Sterritt et al. [87] perform an interesting study on *how autonomic computing can implement a framework for dependability*. They acknowledge that the mechanisms stated above have a considerable influence on the dependability of a system and resort to the *fundamental chain* (Figure 2.3) to present the need for an improved approach to design fault tolerant systems, to help break this chain and further prevent failure. However, they recognize that the creation of such autonomic systems is a major challenge which will require further research and contributions from the subjects of Software Engineering and Artificial Intelligence. Notwithstanding, two strategies for introducing Autonomic Computing are discussed, either by making individual systems autonomic or achieving autonomic behavior at the system level. The article concludes stating that not only the self-healing property of autonomic systems is concerned with a system's dependability, but that all facets of Autonomic Computing are concerned with it. Hence, with the adoption of this paradigm, it is possible to increase system's dependability.

In a more detailed analysis, an autonomic system will monitor itself, analyse monitoring data, decide whether or not an action must be perform to bring the system to correct service and discover, predict or prevent the system from failures (self-configuration, self-healing, self-optimization and self-protection, respectively). The previously mentioned attributes of dependability will help assess if the monitoring data indicates the existence of any anomaly in the system [41]. To deal with such occurrences we resort to the means of fault prevention, tolerance, removal and/or forecasting. These can be embedded in the self-adaption provided by the autonomic system. The anomalies found consist in the threats to dependability - faults, errors and failures.

In which concerns the faults that may occur when working with microservices, these will be thoroughly explained and analysed in the following Section (2.3).

## 2.3 Faults in Microservices

In addition to the fine-grained nature of microservices, this situation gets worse due to their dynamic nature, since a lot of failures that occur in a microservices environment are due to complex interactions between them. These applications usually have complex invocation chains that involve many microservices' invocations, being most of these asynchronous. Likewise, any failures in configuration or coordination of microservices instances may lead to failures at runtime [101]. Another important consideration is that if a fault is not properly handled, it may increase the system failure rate, which is the probability that a component or system fails at runtime [2].

To assure microservices reliability, it is important that failures and performance issues are detected quickly, as well as the root cause of such failures. However, microservices architectures pose new challenges, due to complex dependencies among microservices, service heterogeneity, the vast amount of metrics generated and frequent updates of services (a company such as Netflix can update their services thousands of times a day) [98]. To allow root cause localization of failures in microservices architectures, several works have been

proposed, such as the one in [98]. This work derives the root cause of performance issues through correlation of application performance symptoms with the corresponding system resource utilization. The application level metrics help detect performance issues. From there, the cause analysis engine constructs the attributed graph to represent the anomaly propagation paths and, finally, extracts the anomalous subgraph and infers which service could have caused the anomaly.

Thus, the importance of detecting and locating the faults that may occur in a system during runtime execution is essential to a system's availability. Likewise, many failures are very difficult to reproduce and this can only be done at runtime, thus the importance of techniques such as chaos engineering, which is explained in the next Section (2.4).

Zhou et al. [100] acknowledge the lack of research about fault analysis and debugging of microservices and provide an industrial survey to learn about common faults in microservices, the techniques used for debugging and the challenges that developers face when trying to debug the systems. Furthermore, they perform a benchmark of a medium-size microservice system, which will then be the starting point for the empirical study to investigate the effectiveness of the debugging practices that are used. The outcome documents that the existing debugging practices can be further improved. The faults considered are arranged based on their **symptoms** (functional or non-functional) and **root causes** (internal, interaction, or environment).

However, in a latter work by the same authors [101], where they build MEPFL, which is an approach for latent error prediction and fault localization in microservice applications that learns from system trace logs, they categorize these fault cases in four types:

- **Monolithic Faults** - faults that can cause failures when the application has all its microservices deployed in a single node, with only one instance per microservice and these interacts in a synchronous way. These are usually a result from internal implementation faults of the services.
- **Multi-Instance Faults** - faults associated with the existence of multiple instances of the same microservice at runtime and are a result from lack of coordination among the different instances.
- **Configuration Faults** - faults related to environmental configurations of microservices, such as resource limits and are usually a result of improper configuration of microservices or the environments where these operate.
- **Asynchronous Interaction Faults** - faults that may occur during asynchronous communication among services. Failures can arise when asynchronous invocations are executed or when these are returned in an unexpected order. These usually result from missing or improper coordination of asynchronous invocations (such as the lack of coordination mechanisms).

In such a dynamic environment, it is hard to pinpoint the causes of every fault that occurs in the system or even detect where the fault occurred or have the capability to detect the faults. However in these work, a set of faults found throughout literature review are considered, as well as a set of planned actions and strategies (see Chapter 4) for error recovery that the framework under development will choose to apply to the system, in order to recover from the detected faults.

Among all the existing faults that can occur in a microservices environment, the following, collected from literature review [100, 19, 6, 1, 86], will be considered in this study:

| Failure mode | Possible Failure Causes | Symptom |
|---|---|---|
| **Incorrect Content** | Service received an invalid response | The incorrect value propagates through the service interface and results in a cascading failure. |
| **Corrupted Output** | Missing asynchronous message delivery sequence control | Messages are displayed in the wrong order |
| | Service received a corrupted response | The invalid response is persisted and results in a cascading failure |
| **Crash** | Improper resource limit configuration | Microservice instance is restarted; Timeout exception; Exceptions related to resource-consumption are thrown |
| | Service allocates memory without freeing it (memory leak) | Service exhausts its memory and returns timeouts upon user requests |
| **Hang** | Service receives an abnormaly high amount of requests, making it unable to process them | The service starts to return timeouts to user requests |
| | External service is not reachable | Attempted connections to an external service are timed out |
| | Service acting as a gateway or proxy did not receive a timely response | Service timed out |

Table 2.1: Table of considered faults

These faults are presented according to the Failure Mode and Effect Analysis (FMEA) format. This is a "systematic technique to explore the possible failure modes of individual components or subsystems and determine their potential effects at the system level" [64]. It gathers the potential failure modes of subsystems, their root causes and their effect in the system. In the first column we are presented with the **failure modes**, which are the different ways that the deviation from correct service can assume [8]. The second column describes possible failure causes for each failure mode. In the third column the failure effect (or symptom) is documented, which is the consequence of the failure in the system. The reference of the identified fault is documented in the last column.

The failure modes considered are [8, 18]:

- **Incorrect output** - a service returns a valid response, but containing a wrong result.
- **Corrupted output** - a service returns a corrupted result.
- **Crash** - a service stops working completely.
- **Hang** - a service does not answer any incoming requests or does not produce any further output.

The main focus in gathering the faults presented in Table 2.1 was finding faults whose failure effect could be an HTTP code that the Fault Detection System could detect and subsequently inform the Mitigation Plan Selector. Likewise, and since HTTP is a widely

used communication protocol among microservices, we focused on status codes in the 5xx range (server-side error codes) [23]:

- **500 Internal Server Error** - server has encountered an unexpected condition, thus preventing it from fulfilling the request. This is a broad response usually adopted when no better response to describe the error can be provided.
- **502 Bad Gateway** - server, while acting as a proxy or gateway, received an invalid response from an upstream server.
- **503 Service Unavailable** - server is not ready to handle the request, which is usually a result from server overload or maintenance.
- **504 Gateway Timeout** - server, while acting as a proxy or gateway, did not receive a timely response from an upstream server.

Although these faults could be identified, the major availability and performance breakdowns observed in the cloud are caused by **gray failures**. Huang et al. [42] define these as "component failures whose manifestations are fairly subtle and thus defy quick and definitive detection" and report that these get more common with the growth in complexity and size of cloud systems. Examples of this type of faults are random packet loss, performance-degradation and non-fatal exceptions. These failures assume a specific feature named **differential observability**, which is when the system's failure detectors do not notice problems when applications are afflicted by them, or, more precisely, when different components notice different behaviors of the same application. An example would be when one component is negatively affected by the failure and another does not perceive that there is a failure. These type of failures can sometimes lead to unwanted situations where recovery actions harm the system. This same article reports a scenario where a failure is not detected upon its occurrence due to a resource-reporting bug. A machine that was operating in degraded mode kept getting overloaded and was being constantly rebooted. When the failure detector noticed this constant rebooting, it shut down the machine. This, alongside another problem in the replication workflow, put pressure in the remaining servers, which led to a cascading failure.

Likewise, the authors of the paper outline potential solutions that could be considered to fight gray failure. A possibility would be to to close the observation gap between the system and the application that it serves and moving single failure detection, such as the one that an heartbeat mechanism provides, to a multi-dimensional health monitoring one. Further possibilities include taking advantage of the diversity of components in the system, in order to complement each other information, to detect the subtleties of gray failure. Another possibility would be to discover the temporal patterns that result in gray failures, to help the systems react before applications are affected.

To evaluate if the proposed solution can, in fact, choose plans to recover from the presented faults, fault and failure injection experiments will be performed. Thus, the following Section (2.4) will describe this technique, as well as the underlying technologies.

## 2.4 Fault Injection

To improve and assure a system's dependability, fault tolerance techniques must exist. However, we need a way to evaluate these fault tolerance mechanisms, to assess if our system is capable of withstand failures that may occur. Thus, fault injection shall be used.

This technique can be defined as the willful insertion of a fault or error in a system, to observe how it behaves and handles the resulting effects and recovers from possible failures that may occur [60].

We can identify fault injection as being of four types [79, 66, 102, 13]:

- **Hardware-based** - injection of faults at the physical level by controlling environmental parameters, such as shutting down virtual machines.
- **Software-based** - faults are injected through their implementation in software. Several examples are setting wrong parameters, incorrect timeouts, inject latency in HTTP traffic or sending messaging with a different format from the one expected. This approach mimics software faults through changes performed in the application's source code.
- **Simulation-based** - injection of faults in high-level model(s) of the system, through the usage of simulation scripts or by modifying the model. It enables an early evaluation of the system's dependability when only a model is available. The meaningfulness of the results obtained deeply depends on the accuracy level of the model of the system.
- **Hybrid** - a combination of two or more of the previously identifies fault injection techniques. This approach provides a more complete exercise of the system under analysis.

Nonetheless, a drawback of software-based fault injection is that it requires the modification of the application source code. Thus, the code that executes during the experiment is not the same as the one that will run in production [102].

Moreover, software injection methods can be categorized according to the timing of the faults injected - compile-time or run-time [102]. When the program's source code is modified before the program image is loaded and executed, in order to inject a fault, we say that the fault is injected at **compile-time**. The modified code helps emulate the fault scenario, generating an erroneous software image. When this image is executed, it activates the injected fault. Since the effect of this fault is hard-coded, it can be used to emulate permanent faults. On the other hand, when a fault is injected during the program execution time, we say that it is injected at **run-time**. To inject such faults, a mechanism is required to trigger fault injection. Examples of triggering mechanisms include time-out, where the injection is triggered based on a timer, and exception/trap, where, for example, a software trap instruction invokes the fault injection.

Furthermore, in recent years, the term **Chaos Engineering** appeared, emerging as a "discipline to enable resiliency in the cloud"[91]. Another definition from [78] describes it as "the discipline of experimenting on a system to build confidence on the system's capability to withstand turbulent conditions in production". When an application is deployed to a production environment, it faces many unpredictable conditions due to the dynamics and complexity of microservices invocations. Thus, it is important to experiment in a system during runtime, in order to assess the system's resilience under real conditions [51]. Chaos engineering can be classified as simulation-based fault injection since the system is tested under a simulation of what the real failure conditions would be in a production environment. [51]. Through such experiments, developers can discover system's weakness which are required to be solved in order to improve its resilience.

One of the most well known examples of this technique is Chaos Monkey, which was created when Netflix moved their data center to Amazon Web Services and wanted to

assess how potential failures in Amazon Web Services (AWS) would affect their ability to provide continuous service [3]. Chaos Monkey then became part of a bigger group, the **Simian Army**. While the Chaos Monkey operates by randomly selecting an instance of a virtual machine and shutting it down, there are other that work at a higher level, such as Chaos Kong, which simulates the failure of an entire Elastic Cloud Computing region, in Amazon Web Services. Currently, the Simian Army is no longer actively maintained, but Chaos Monkey continues to operate as a standalone service [67].

To further express the importance of testing an application's resilience, Heorhiadi et al. [39] present, in Table 1, a set of outages in popular Internet services. Their postmortem reports show missing or faulty failure-recovery logic, which pinpoints that unit or integration tests are insufficient to detect these bugs. They further recommend performing resilience testing (*e.g.*, chaos engineering) to assess if the application can recover from failures that are common in cloud environments.

If we intend to perform an experiment that follows this approach, there are four principles that must be addressed [11, 60]:

- **Build a hypothesis around steady-state behavior** - even complex systems manifest regular behaviors that can be foreseen. Thus, it is important to find metrics that can define a system's steady state behavior so that we can hypothesize and assess the system's health.
- **Vary real-world events** - the design of experiments must consider a sample of all possible inputs that can occur in the real world. To do so, we can inject a failure in a service, simulate a service failure or choose a subset of the service's users to perform an experiment.
- **Run experiments in production** - it is never possible to reproduce real system conditions in test scenarios, thus integration tests must be performed in production to assure that many critical failure scenarios are not skipped.
- **Automate experiments to run continuously** - it is important to automate tests to assure that these are run repeatedly as the system evolves over time, hence preserving confidence in results over time.

There are many chaos engineering tools available in the market. However, some of them are not available for use with a Kubernetes orchestrator, thus those will be excluded from the considered tools. A comparison of the remaining tools that can be used with this orchestrator is performed and presented in Table 2.2 [34, 76].

Although Istio is among the considered tools, it is not a fault injection tool. It is a service mesh used to manage the communication among microservices that also allows to inject faults to the traffic it monitors, such as delay and abort faults.

The comparison performed helped define the fault injection tools to be used in the solution's evaluation. Litmus and Istio were chosen since Litmus has a wide set of experiments available and it is already setup in the cluster where the work will be performed and the same applies to Istio. Although the latter is very limited in the range of experiments, it allows to simulate faults defined in the previous section. The process of fault injection for each of the faults listed in the previous section is documented in Chapter 4.

In the following section, a deeper insight about the methodology used to identify the faults in the system is documented.

| Name | Description | Pros | Cons | Number of attacks |
|---|---|---|---|---|
| Litmus | Cloud-native chaos engineering framework for Kubernetes | Large number of experiments available; WebUI available; public repository to share chaos experiments | Administrative overhead; Complicated to perform experiments | 39 |
| ChaosBlade | Alibaba's experimental injection tool that follows the principles of chaos engineering | Large number of experiments; Fault injection at the application level | Documentation lacks language support; Lacks reporting capabilities | 40 |
| Chaos Mesh | CNCF hosted project recognized as a cloud-native chaos engineering platform | Reasonable amount of experiments; Does not require special dependen--cies or modifica--tion of the deployment logic | No attacks available at the node level | 17 |
| Istio | Open source service mesh to manage communication among microservices | Built into Istio service mesh; Easy to setup experiments | Very limited number of experiments | 2 |

Table 2.2: Fault and Failure Injection Tools

## 2.5 Distributed Tracing

The distributed and decoupled architecture of microservices requires new means to monitor the existing systems. The complexity of the request flow makes it more difficult to isolate the faults and uncover where did they occur and who caused them. Thus, distributed tracing can be defined as an approach where a unique identifier is tagged at the origin and persisted all the way in the flow of information until the end of the request's lifecycle [81]. This technique can be used to study and understand the behavior of distributed systems, as well as monitor their performance. It is also a means to improve **Observability**, which is the "measure of how well internal states of a system can be inferred from knowledge of its external outputs" [95].

In a nutshell, distributed tracing starts by creating a trace when a client request arrives to the system. A trace can be seen as a "visualization of the life of a request as it moves through a distributed system" [71]. It is composed by spans, which are different parts of the workflow that contain information, such as the operation name and a start and finish timestamp, and are the basic units of work. The start timestamp of the child lies with the duration of the parent span. However, the end timestamp of the child spans is not always before the finish timestamp of the parent when the two are asynchronous [61]. Each

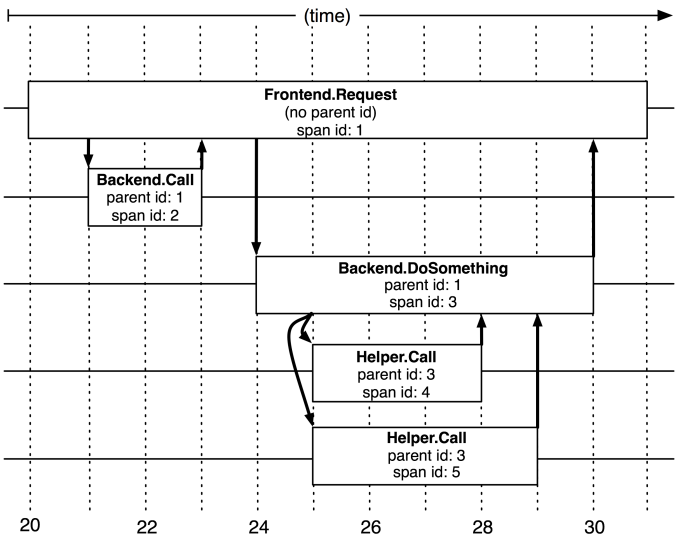


Figure 2.4: Causal and temporal relationships between spans. Figure from [85]

component of the system contributes with a span. With the provided information, spans can be grouped and ordered to model causal relationships. This is crucial to understand which service was the caller and which was the callee, to help define, in a failure scenario, what could be the origin of a certain failure.

Furthermore, among the metadata that each span contains, we can find the *Span Context*, which contains the *span id*, *parent id* and *trace id*. The context provides information for the span to know who its parent is and to each trace it belongs to. The child span creates its id, which it then propagates to the child span, as the parent id, together with the trace id [61]. The span id field consists in a unique identifier used to identify the unit of work. The parent id field is used to reconstruct the causal relationships between the individual spans in a distributed trace (Figure 2.4). For instance, in Figure 2.4, the parent id in the *Backend.Call* span provides enough information to infer that the *Frontend.Request* span is the parent span. A span that is created without a parent id is know as *root span*. If several spans share the same trace id, it means that they belong to the same distributed trace [85]. The *trace id* is a unique identifier used to identify the trace that a span belongs to.

The trace data collected can further be sent to a monitoring agent which will display this information in an interface [82]. The traditional way to instrument an application would require to change its source code. However, recent approaches arose, such as the use of an underlying service mesh like *Istio* or *Traefik*. The first follows a sidecar design pattern that allows to attach lightweight proxies to services. The second one opts for the usage of a proxy endpoint that runs in each node and handles the routing. Among the existing distributed tracing tools, we can identify *Jaeger* [49] and *Zipkin* [73].

One of the first and most relevant works performed in this area is **Dapper** [85]. Google developed Dapper as a tracing infrastructure for their distributed systems. They recognized the importance of having a tool that would allow them to pinpoint errors along their highly distributed infrastructure and set four important design goals - low overhead, application-level transparency, scalability and quick availability of tracing data. Low overhead was desired to avoid that instrumenting their services would have any impact in their infrastructure's performance. Application-level transparency allowed to avoid fur-

ther problems such as errors due to incorrect or missing instrumentation. Scalability was also important since the system would need to handle big clusters such as the one's at Google. The possibility of having tracing data available in little time would allow quicker reactions to runtime failures. The application-level transparency was achieved through a set of control flow, ubiquitous threading and Remote Procedure Call (RPC) libraries. The desire to have a low overhead and scalability was attained with the usage of **adaptive sampling**. This approach would allow to tune the sampling rate to cope with different workloads, increasing their sampling rate when dealing with low traffic workloads and high traffic workloads would require the sampling rate to be decreased to control the overhead.

Initiatives such as OpenTracing API have been created to provide a standard framework to instrument applications. Having a vendor-agnostic API helps in the process of instrumenting tracing into the application's source code without having to worry about the distributed tracing tool being used and having to repeat the instrumentation process when they intend to change tools. Another well-known initiative is OpenCensus, which consists in a set of libraries to help collect application metrics and distributed traces, that can be sent to a backend to be further analyzed. These tools have now been merged to form OpenTelemetry, in order to provide a single place with libraries, Application Programming Interface (API), Software Development Kit (SDK) to capture distributed tracings and metrics from the applications [70].

The following and final section of this Chapter presents related work in the field of autonomic computing, namely its self-managing properties.

## 2.6 Related Work

The research for autonomic systems has been increasing continuously throughout the last years. Several proposals of architectures for autonomic systems or components that can make a system autonomic have been proposed. However, there is a scarcity of implementations and corresponding evaluation, having noticed that several papers found throughout literature review only depict conceptual architectures and components, with the implementation still to be performed.

Subsections 2.6.1, 2.6.2, 2.6.3 showcase relevant work found that hold self-adaptive properties. Subsection 2.6.4 does an overview of existing work and documents the research direction to be taken in the development of our framework.

### 2.6.1 Gru

As documented in Section 2.2, autonomic capabilities can be implemented in an application using an internal approach, where the autonomic manager is part of the managed application, or follow an external approach.

Florio et al. [28] propose an implementation of the MAPE-K loop, where an autonomic manager is attached to a system that was not initially designed to be autonomic, thus following an external and decentralized approach. An **autonomic enabler** is used as an intermediary (link) between the autonomic manager and the system to manage. The authors state that the adoption of a decentralized approach to autonomically manage an application reduces the risk of bottlenecks and single point of failure.

The approach taken is established on a multiagent system, where each agent is an

"intelligent" and independent unit that performs autonomic actions on the managed system, based on their own information and information coming from peers. The implementation was performed on Docker containers, with each Docker node consisting of a Docker Daemon and a **Gru-Agent**. The latter implements the **MAPE-K loop** as follows:

- The **Monitor** interacts with the Docker Daemon to gather container monitoring information.
- The **Analyser** receives the collected data and evaluates the available resources for each microservice.
- The **Planner** decides which action to take according to a partial view of the system provided by the Analyser, based on policies (rules to execute one or more actions) and strategies (algorithms to help choose the appropriate policy).
- The **Executor** interacts with the Docker Daemon to apply to the container the actions from the policy chosen.

The decisions taken are only based on a partial view of the system because the agents only exchange information with a set of their neighbors, which helps reduce the overhead. Besides this inter-agent communication to exchange knowledge, the agents also communicate with a Repository containing configuration information of each agent and $\mu$Service-Descriptors. The latter has information about each microservice running in the system, as well as QoS constraints related to the execution of the microservice. Such constraints are considered by the agent when deciding the actions to actuate.

The experiments performed reveal that Gru is capable of scaling containers according to the constraints defined in the $\mu$Service-Descriptors, but this work is limited to simple self-healing actions, such as scale up and scale down.

### 2.6.2 Autonomic Version Management in self-healing microservices architecture

A considerable amount (16% according to [36]) of the incidents that occur in cloud systems are a result of software updates.

Wang [94] proposes an architecture with self-healing capabilities and autonomic resource management. However, the initial effort consists in a component of a version manager to autonomically manage version upgrades in microservices. The author noticed that papers only focused on the impact of service upgrades in a single application. Nonetheless, different teams from a company tend to reuse microservices from each other. If a microservice that another team depends on is upgraded, it can break the entire application. Thus, autonomic version management at the application and system level must be considered. By dealing with unexpected changes across system boundaries, it may be possible to avoid or minimize the impact of partial outages of the system.

The proposed architecture allows hot swapping and automated version upgrading of services, without impacting the other services that depend on them. It containerizes microservices as business nodes that run within a group of virtual machine instances. The manager nodes that help microservices working together and easing system changes are allocated in another set of virtual machines. To enable communication among instances of a node, a message broker is used. The manager node takes care of orchestration, scheduling

and control of the microservices in the cluster, and is composed of the four components that follow.

The component for managing version upgrades at application and company level is the **version manager**. It is in charge of defining how and to where the traffic should be redirected according to the type of changes (minor or major). The **service registry and discovery service** allows services to locate each other. The **schedule resolver** acts as a load balancer and performs resource usage optimization. Lastly, the **health monitor and fault manager** are intended to detect, correct and prevent failures in the system. It collects microservices monitoring information at run time, to provide information to the score engine to attribute a score according to their health status. This component also contains strategies to help minimize the impact of partial outages. With regard to the communication between components, approaches using message formats such as JavaScript Object Notation (JSON) or Protocol Buffers are considered.

Despite the fact that the initial focus of the proposed solution is focused on autonomic version management, it is something that must be considered due to the fact that microservices are constantly changing (*e.g.*, a company, such as Netflix, updates their services thousands of times a day). Besides that, it is common for different autonomous teams to be working on the same product. Addressing such issue can help us walk a step closer in avoiding failures due to software updates. The solution is planned to be further tested in a production environment, through chaos engineering experiments.

### 2.6.3 An architecture for self-managing microservices

The dynamic environment of microservices makes it difficult for developers and system administrators to be aware of everything that happens in a system. Thus, the development of autonomic applications or externally attaching autonomic capabilities to applications is an important step to help manage applications and to assure important quality attributes, such as their availability.

Toffetti et al. [88] recognize that cloud applications need to be continuously managed to accommodate the resource usage to the workload and to ensure that correct service is provided even when subject to failures. They defend that the actual management functionalities have intrinsic limits since these functionalities are provided as infrastructural or third party services. This external approach inhibits their natural adaptation to the managed application and requires additional management efforts.

Hence, an architecture for self-managing microservices is proposed. Through the usage of a consensus algorithm, a leader can be elected to assign management functionalities to nodes, which allows for a node in the cluster to take over the management logic of another node that eventually fails. In order to provide service discovery and automatically update components' configuration upon changes, *Etcd* is used. *Raft* is the chosen consensus algorithm used for fault-tolerance and consistency of the keystore (etcd). Configuration management helps implement self-management functionalities, such as self-healing. Components are distributed across different failure domains to create a resilient architecture and ensure that components that fail are quickly restarted. Furthermore, sharing monitoring data in *etcd* enables health monitoring and auto-scaling components to become stateless.

This architecture works both on atomic services, as well as in microservices applications. However, to enable self-managing microservives applications, multiple key-value stores are required - a microservices application has its own key-value store and there is a global

key-value store that works at a global applicational level, which is in charge of endpoint discovery and leader election. The leader handles monitoring, auto-scaling and health-management functionalities.

In [89], an experimental evaluation of the proposed architecture was performed. The target application was a typical 3-tier monolith application, which was subject to architectural changes, in order follow the principles of cloud-native applications. It was then split in Docker containers, deployed among several virtual machines.

The authors wanted to demonstrate that the self-managing architecture proposed could address the requirements of *elasticity* and *resilience*. A load generator was used to exercise the system over an increasing load and test its elasticity. It was possible to observe that the application was able to automatically scale up and down, according to CPU utilization and application response time, and maintain the desired response time. With regard to resilience, a set of IaaS failures were emulated by killing containers and virtual machines. When testing the system behavior upon container failures, new services were instantiated appropriately, but there was a significant impact on their response time. With respect to virtual machine failures, the effects observed varied according to the type of virtual machines running and the type of containers running on those machines. If the machines were running stateless containers, the effects were small and transient. However, if the virtual machine is running stateful containers, such as a database, an increase in response time of the database was observed, which subsequently affected the application's response time.

### 2.6.4 Research notes

IBM proposed the autonomic computing concept two decades ago and it is still a challenge to develop a system with autonomic capabilities that will not disrupt its correct service.

Throughout the extensive literature review performed about autonomic systems that display self-healing properties, it is still notorious the lack of research about this subject. Even though some works were found on this topic, only some of them document implementations and the results from experiments, while others only present the proposed solutions.

Nonetheless, the research performed, and particularly [28] and [88, 89], helped endorse the fact that it is possible to attain autonomic capabilities in an application. The implementation from [28] uses a feedback loop (the MAPE-K loop), which is run periodically. However, the proposed solution of the present work consists in a continuous feedback loop. This can be seen as an improvement, which would allow faster reaction to failure events. The solution proposed by [88, 89] takes on a hierarchical approach, with the management logic running inside the microservices. This is different from the solution currently proposed in this framework, where the management logic is provided externally.

Furthermore, it was possible to realize that, to the best of my knowledge, no work similar to the one proposed in this thesis, where a publish-subscribe middleware is used to share information about faults and recovery actions, has already been performed. Besides the fact that the present approach is documented specifically concerning a Kubernetes cluster, it can still be used with other orchestrators due to the decoupling provided where only the message format needs to be agreed.

Since the orchestrator under consideration (Kubernetes) already provides some self-

healing capabilities, such as the ability of automatically restarting failed pods or migrating deployments if any of the cluster nodes fail, the proposed solution aims to complement such capabilities, in order to fullfil existing gaps and improve system's dependability.

The following Chapter documents the proposed solution to implement the autonomic framework.

# Chapter 3

# Proposed Solution

This Chapter presents the proposed solution, which is an autonomic framework intended to keep a cloud-native application available. Its self-healing properties will provide error recovery plans and strategies according to the fault information provided by a Fault Detection System. The latter resorts to distributed tracing to gather important fault information. This information contains the name of the affected components, the category of the fault and a brief description of the fault, the communication protocol used among components, the causal relationship between them and the timestamp of when the fault was detected. All this information is placed in a JSON object to be sent to the Mitigation Plan Selector. According to the received fault information, the recovery plans will be issued. These consist in JSON objects that are composed by a set of fields, such as the description of the action, the affected components and the recommended actions, as documented in Section 3.3. These error recovery plans are intended to bring the system back to correct service.

An overview of the solution is presented in figure 3.1.

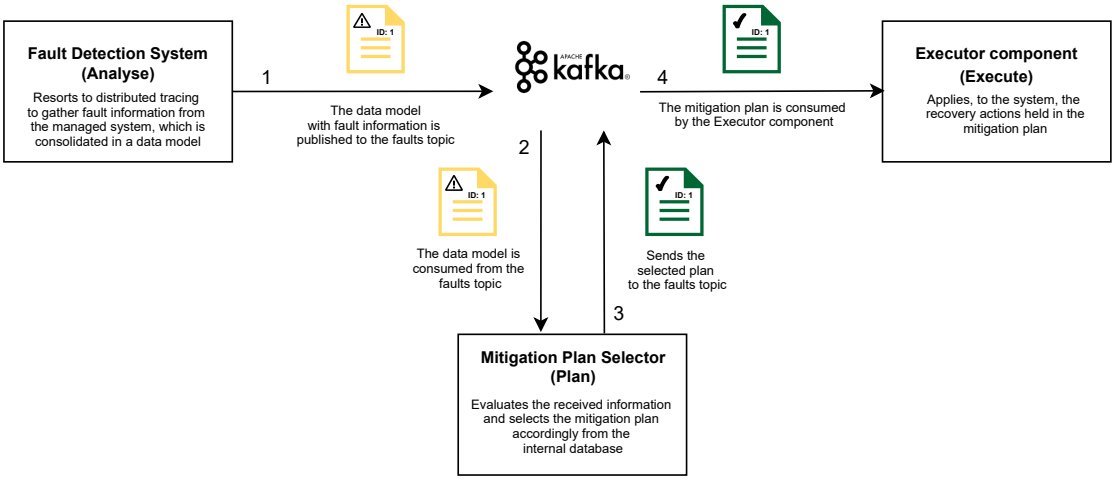


Figure 3.1: Proposed solution overview

As shown in Figure 3.1, the components will send information (the JSON objects) to a topic in the Kafka's Publish-Subscribe middleware. Publish-Subscribe was used because this type of middleware allows for an asynchronous, fast and reliable communication, which helps both speed up and decouple the communication among the different components. Although the Monitor entity is not shown in the figure, it is handled by the underlying

service mesh belonging to the cloud-native application's technology stack.

The managed application used in the present work is Stan's Robot Shop [46]. This is one of several well-known sample microservices applications, used to perform tests and learn more about this type of architecture. It is intended to mimic an e-commerce application that sells robots. It includes a catalogue of products, a cart to store products and the possibility to order them. The system is composed of twelve microservices and has a diverse technology stack comprised of different programming languages such as Java, Python and Golang, thus showing the heterogeneity of this type of architecture.

Throughout the present chapter, examples of commands and the validation of the requirements are directly related to Kubernetes since, in the context of the project, the cluster that will be used as a testbed has this this orchestrator deployed.

Nonetheless, as with any software engineering project, we must start by defining the functional and non-functional requirements of the system (Section 3.1). These are followed by the architecture definition in Section 3.2. Section 3.3 showcases the Mitigation Plan Data Model, which is the data model used in the mitigation plan. Finally, Section 3.4 and Section 3.5 document the recovery actions and strategies, respectively, applied by the framework.

## 3.1 Requirements

The requirements of the system help understand what the framework does and how it must behave to fulfill the system's goals. Functional requirements describe what the system must do and how it must react to runtime stimuli [12]. On the other hand, the non-functional requirements, also known as quality attributes, define the properties of the system that help realize the goals of the system [12]. The functional requirements are documented in Subsection 3.1.1, followed by the non-functional requirements in Subsection 3.1.2. The latter are thoroughly validated in Subsection 3.1.4.

### 3.1.1 Functional Requirements

Starting by the functional requirements, these are documented with use cases, in the fully dressed format, allowing a more in-depth understanding of each of the requirements. Two fields were removed from this format - **Technology and Data Variations List** - since there are no deviations either in the technology or data being used due to the technical constraint described in Section 3.1.3 and the **Special Requirements** since in each use case scenario the three existing non-functional requirements would be repeated. These use cases are documented in Subsubsections 3.1.1.1, 3.1.1.2, 3.1.1.3 and 3.1.1.4. Their prioritization is performed after the description of the four use cases.

#### 3.1.1.1 Use Case 1: Collect the fault information from the Publish-Subscribe

**Primary Actor:** Fault Reader

**Stakeholders and Interests:**

- Fault Reader: Wants to obtain information about the detected faults, parse and send it to the Fault Analyser

- Publish-Subscribe: Wants to provide topics to host the shared information about the detected faults

**Preconditions:**

- There must be information in the faults topic of the Publish-Subscribe
- The information in the topic must be in JSON Format

**Success Guarantee (Postconditions):** The fault information is collected from the faults topic, the offset of the message read is committed and the information is sent to the Fault Analyser

**Main Success Scenario (Basic Flow):**

1. Information about a detected fault in the system is placed in the topic of faults of the Publish-Subscribe middleware
2. The Fault Reader collects the information contained in the faults topic
3. The offset of the message, which contains the fault information, is committed
4. The JSON object is deserialized and sent to the Fault Analyser

**Extensions (Alternative Flows):**

1.a The Publish-Subscribe is not properly configured, which means that the information placed on the topic is not renewed

2.a The format in which the fault is documented is not in accordance with what the Fault Reader expects to receive

**Frequency of Occurrence:** As soon as information is available in the faults topic

#### 3.1.1.2 Use Case 2: Parse the fault information collected

**Primary Actor:** Fault Analyser

**Stakeholders and Interests:**

- Fault Reader: Wants to send the collected information about the fault to the Fault Analyser
- Fault Analyser: Wants to analyse the information received and provide the fault's description, where did it occur and which components were affected, as well as the protocol being used for communication between the affected components

**Preconditions:** Information about a fault must be received from the Fault Reader

**Success Guarantee (Postconditions):** The fault information is analysed and the information resulting from the analysis is sent to the Plan Selector. This information will contain the description of the fault, the order in which the command(s) must be applied to the affected component(s)

**Main Success Scenario (Basic Flow):**

1. Fault information is received from the Fault Reader
2. The information is properly analysed in order to assess the fault's type, description and affected components
3. Once the analysis is complete, the resulting information is sent to the Plan Selector

**Extensions (Alternative Flows):**

1.a No information is received from the Fault Reader

**Frequency of Occurrence:** Whenever information is collected from the faults topic and read by the Fault Reader

#### 3.1.1.3 Use Case 3: Select a mitigation plan to be applied upon the fault detected

**Primary Actor:** Plan Selector

**Stakeholders and Interests:**

- Plan Selector: Wants to select a mitigation plan to apply after receiving the fault information from the Fault Analyser
- Fault Analyser: Wants to send to the Plan Selector the assessment of the received fault information
- Database: Wants to provide the mitigation plans that it stores

**Preconditions:**

- There must be a predefined set of mitigation plans in the database
- Analysis information of a fault must be received from the Fault Analyser

**Success Guarantee (Postconditions):** A mitigation plan is selected after receiving the fault information from the Fault Analyser and it is sent to the Plan Writer.

**Main Success Scenario (Basic Flow):**

1. The fault analysis information is received from the Fault Analyser
2. A search is made in the database where the mitigation plans can be found
3. A mitigation plan is found to mitigate the perceived fault
4. The mitigation plan found is complemented with the fault information received and sent to the Plan Writer

**Extensions (Alternative Flows):**

2.a The database does not respond when the search is carried out

1. A retry of the request is sent to the database

3.a No match is found after the search

**Frequency of Occurrence:** Whenever fault assessment information is received from the Fault Analyser

#### 3.1.1.4 Use Case 4: Send the selected mitigation plan to the Publish-Subscribe

**Primary Actor:** Plan Writer

**Stakeholders and Interests:**

- Plan Writer: Wants to send the selected mitigation plan to the faults topic
- Publish-Subscribe: Wants to store the mitigation plans received and make them available to the Executor component

**Preconditions:**

- A mitigation plan must have been selected by the Plan Selector
- There must be enough space in the Publish-Subscribe to store the selected mitigation plan

**Success Guarantee (Postconditions):** The chosen mitigation plan is placed in the Publish-Subscribe

**Main Success Scenario (Basic Flow):**

1. The mitigation plan received from the Plan Selector is serialized to JSON format
2. The mitigation plan is sent to the Publish-Subscribe
3. The Publish-Subscribe receives the plan

**Extensions (Alternative Flows):**

3.a The plan received is in the wrong format

**Frequency of Occurrence:** Whenever there is a mitigation plan to send to the Publish-Subscribe

Regarding the **priority** of the presented functional requirements, three values were considered - **High (H)**, **Medium (M)** or **Low (L)**. These were attributed to the different requirements according to their relevance to the proposed solution. A high level of priority was attributed to the four use cases defined since all of them are crucial to the implementation of the framework. Notwithstanding, we should be aware of the risks associated with the prioritization defined since all the risks have high priority. This could mean that, if the requirements are not fulfilled, the proposed solution could run the risk of not being finished. However, the estimated implementation difficulty is low, thus the defined prioritization does not pose any major threat to accomplish the final solution.

### 3.1.2 Non-Functional Requirements

In which concerns the non-functional requirements (also known as Quality Attributes), these are documented using the scenarios of quality attributes, which gather a set of important characteristics to help describe these requirements. Three non-functional requirements were considered for this work and their priorities are assigned in the utility tree directly after the following scenarios.

#### 3.1.2.1 Scenario 1 - Performance

**Stimulus:** All the fault information sent to the Publish-Subscribe needs to be processed by the Mitigation Plan Selector

**Source of the stimulus:** Fault Detection System

**Artifact:** System

**Environment:** System runtime (normal operation)

**Response:**

- The Mitigation Plan Selector processes the fault information received
- The information received about the fault is analysed to find out the fault's type, where did it occur and which components were affected, as well as the protocol being used and the causal relationship
- The Mitigation Plan Selector determines which mitigation plan to apply
- The selected plan is sent to the faults topic in the Publish-Subscribe

**Response Measure:** During the overloaded period, the Mitigation Plan Selector must be capable of providing service without losing any data.

#### 3.1.2.2 Scenario 2 - Availability

**Stimulus:** The Mitigation Plan Selector is waiting for information to be sent to the Publish-Subscribe and crashes

**Source of the stimulus:** Internal

**Artifact:** System

**Environment:** System runtime

**Response:** The failed Mitigation Plan Selector replica is shutdown and a new one is started. During this period no information will be consumed from the topics, thus no data is lost.

**Response Measure:** The time needed for repair must not be longer than 5 minutes

#### 3.1.2.3 Scenario 3 - Scalability

**Stimulus:** All the fault information that arrives at the Publish-Subscribe middleware needs to be processed.

**Source of the stimulus:** Fault Detection System

**Artifact:** Mitigation Plan Selector, System

**Environment:** System runtime

**Response:** The number of replicas of the framework are increased to cope with the increase in load

**Response Measure:** Number of additional replicas added to face the increase in load

In scenario 3 (Scalability), only the scale up action is being considered since, in this scenario, we are assessing an increase in load. Nonetheless, whenever the load decreases, the Kubernetes auto-scaling mechanism is capable of scaling down and reduce the number of existing replicas.

The utility tree shown in Figure 3.2 gathers the quality attributes considered. The root node only contains the word "utility" which is used as an expression of the overall "goodness of the system" [12]. The children of this node contain the quality attributes, which have, as their children, their refinement, i.e., a decomposition to help better understand that requirement. Finally, the leaf nodes are the scenarios of the quality attributes and the edges that lead to them contain their priorities, according to two different factors:

- Impact on the architecture - High (H), Medium (M), Low (L)
- Value for mission or business - High (H), Medium (M), Low (L)

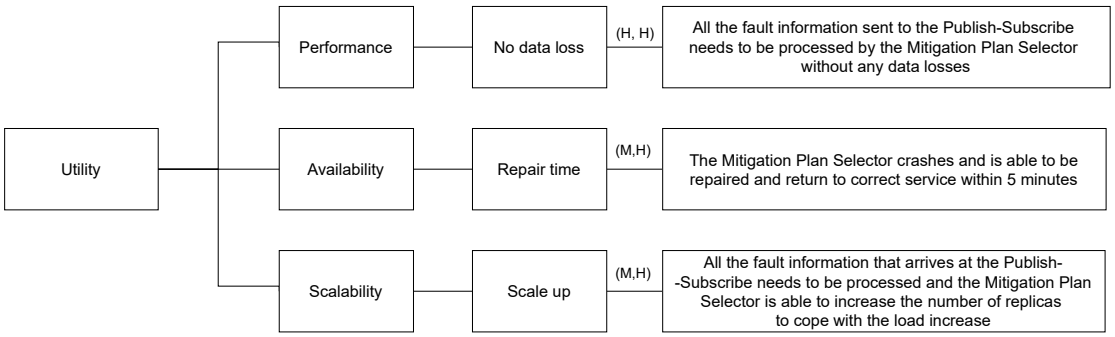


Figure 3.2: Utility Tree

The three attributes have a high priority in which concerns the value for business since they are must-have requirements for the framework. With regard to the impact in the architecture, it is set as high for performance because it requires additional considerations in the architecture to assure that no data is lost. Availability and scalability are assigned medium priority since both require the existence of an orchestrator to assure the responses desired.

### 3.1.3 Technical Constraints

According to Len Bass et al.[12], a constraint is a design decision with zero degrees of freedom. Likewise, technical constraints are restrictions that influence the architecture and can be related to several aspects such as the data format or communication protocol to be used in a system. In the development of this component there is one technical constraint that must be considered:

- **Data Format:** The data format used for data exchange between the component being developed and the external components must be JSON [53].

### 3.1.4 Validation of Non-Functional Requirements

The non-functional requirements documented in the previous subsection need to be thoroughly validated to assure that this can be met, which is the purpose of the present subsection.

Regarding **performance**, it is claimed that during a period with high volume of fault information, the Mitigation Plan Selector must be capable of providing service without losing any data. Since Apache Kafka is the technology being used as the Publish-Subscribe middleware, it has the capability of storing data until it is consumed from the topics by the Mitigation Plan Selector. Hence, even if a high load of requests is received, they can be held in the topics and not get lost while the framework is handling other requests, thus fulfilling the performance requirement.

From an **availability** standpoint, the repair time after a crash occurs is considered, which is the time taken between the detection of the failure and the repair of the failed component. Vayghan et al. [92] performed a set of experiments to evaluate the service outage of Kubernetes and concluded that, with the default configuration of Kubernetes, the worst case scenario for the repair time after a pod failure is about 33 seconds and for a node failure is about 263 seconds, which is less than 5 minutes. This experiment helps validate the availability requirement defined.

Finally, regarding **scalability**, which can be defined as "the measure of a system's ability to increase or decrease in performance and cost in response to changes in application and system processing demands" [33], we need to consider the type of scaling that we are going to perform (horizontal or vertical) and the response metric we are using. In which concerns the type of scaling, we are considering only horizontal scaling, which can be defined as "scaling by adding or removing machines/replicas from our pool of resources" [12]. About the metric, we are considering that the component must be able to scale up or down according to load fluctuations.

Since we are using the Kubernetes orchestrator, and considering that the auto-scaling feature is enabled, the default configuration measures the relative CPU utilization of each active pod and uses the formula in Figure 3.3 to evaluate the number of pods (P) that are needed to maintain the CPU utilization under the defined target, being $U_{target}$ the target relative CPU utilization [17].

$$P = \left\lceil \frac{\sum_{i \in ActivePods} U_i}{U_{target}} \right\rceil$$

Figure 3.3: Formula of the Kubernetes' Horizontal Pods Auto-scaling algorithm [17]

Since this feature of the Kubernetes orchestrator is capable of scaling the pods up or down according to some specified metric [17], we can conclude that the scalability required is fulfilled.

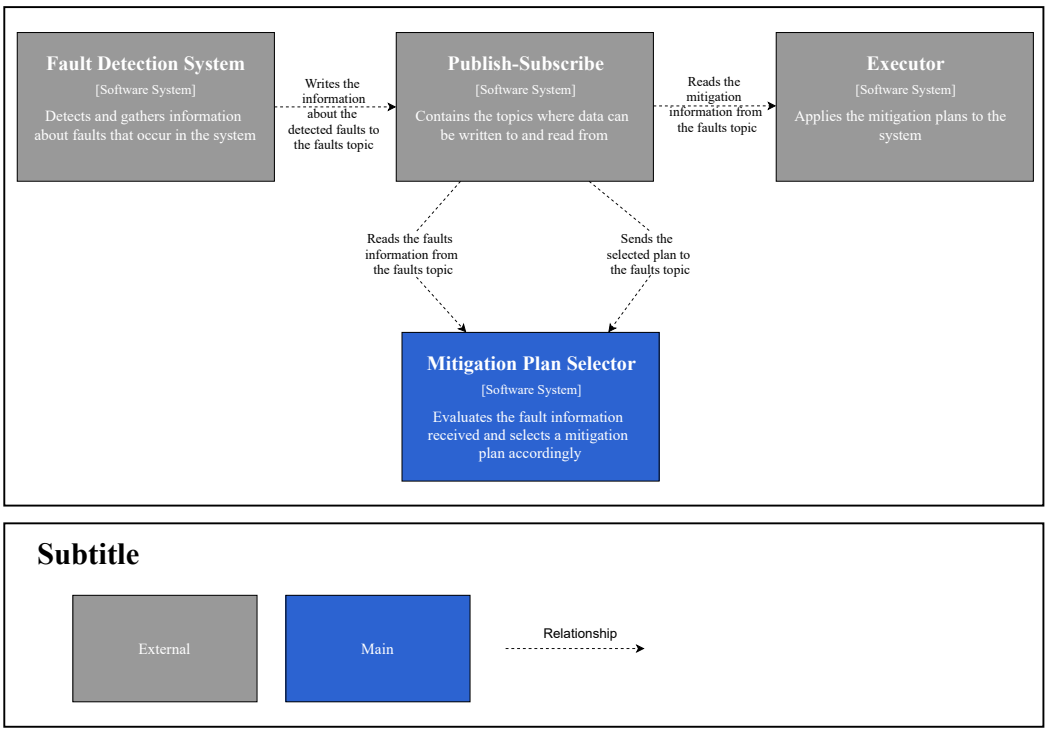


Figure 3.4: Context Diagram

## 3.2 Architecture

This section is meant to showcase the architecture of the component which will be developed. The architecture is detailed using the C4 Model, created by Simon Brown [15]. This was the chosen model due to the fact that it has all the relevant aspects needed to document the architecture of my system, in a clear and non ambiguous way.

### 3.2.1 Level 1 - Context Diagram

Starting by the first level (C1), we create the **context diagram** (Figure 3.4), which presents the component to be developed (Mitigation Plan Selector) and the systems with whom it interacts, allowing us to behold a higher level representation of the component's environment.

On the left side we have the **Fault Detection System**. It is in charge of detecting the faults that may occur in the system resorting to distributed tracing and gather information about these faults. This includes the affected components, a description about the fault and detection timestamp, thus collecting crucial information that will be essential in the process of generating the mitigation plans. This information is then sent to the **Publish-Subscribe**, which is the middleware that allows asynchronous communication between the different components and from where the information will be retrieved by the **Mitigation Plan Selector**. This component will fetch one of the available mitigation plans from the database. The chosen plan will then be published to the Publish-Subscribe Model, from where the data will be consumed by the **Executor**, which is the component in charge of applying the mitigation plans to the system, in order to recover from the existing failure.

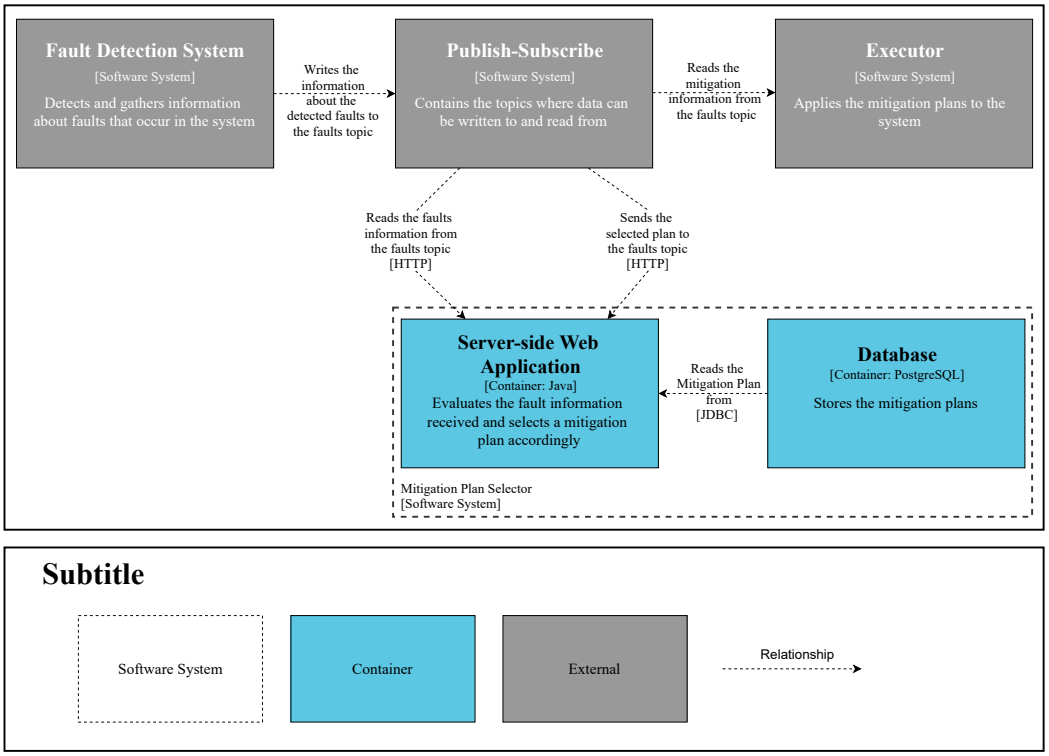


Figure 3.5: Container Diagram

### 3.2.2 Level 2 - Container Diagram

Drilling down in the architecture, we reach the second level (C2), where we present the **container diagram** (Figure 3.5), that allows a more in-depth understanding of the framework.

This allows to split our component in two different containers, the **Server-side Web Application** and the **Database**. The Web Application will be developed using the Java programming language, where the analysis of the fault information will take place, as well as the mitigation plan selection. The Database will store the mitigation plans, which will be fetched by the Web Application. Despite the fact that the data exchanged (including the mitigation plans) will be in JSON format, the plans are stored in the schema defined for the PostgreSQL table. Then, whenever information needs to be collected to compose the mitigation plan, it is selected and saved to the mitigation plan, in JSON format.

### 3.2.3 Level 3 - Component Diagram

Reaching the third level (C3) of the architecture, we present the **component diagram** (Figure 3.6), where the inner components of the web application are known. Hence, this is composed of four components, the **Fault Reader**, **Fault Analyser**, **Plan Selector** and **Plan Writer**. The first one reads the fault information gathered by the Fault Detection System, which is held in a topic of the Publish-Subscribe middleware. After parsing and deserializing the data, which is in JSON format, the information is sent to the Fault Analyser. This evaluates the fault information received, allowing to assess the affected components, the fault description and detection timestamp. The fault analysis data generated will be sent to the Plan Selector. This component will select, from the database, a mitigation plan to be applied to the existing failure. When the plan is chosen, it will

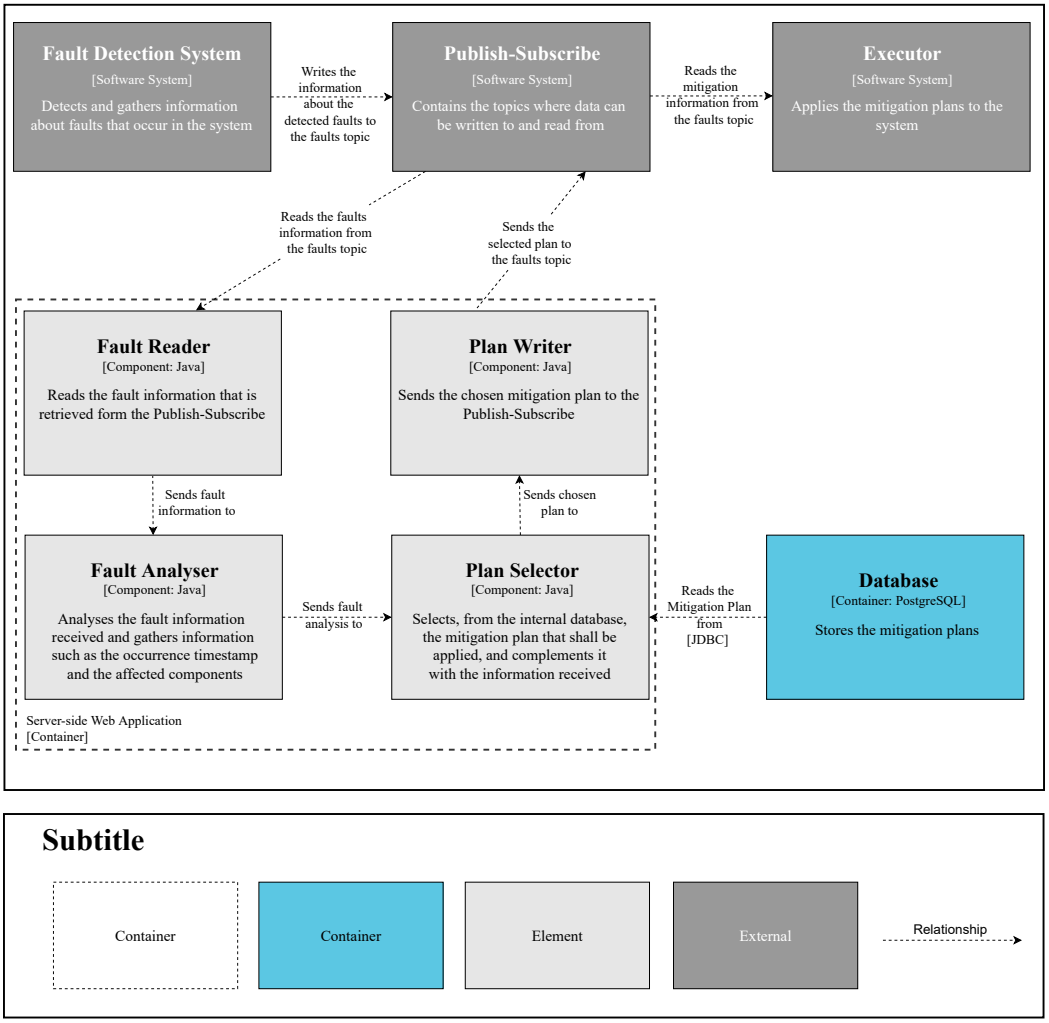


Figure 3.6: Component Diagram

be sent to the Plan Writer, which serializes the plan to JSON format and writes it to the **Faults Topic**. This topic of the Publish-Subscribe will store the mitigation plans to be consumed and applied to the system by the Executor.

The following subsection presents the data model to be used for the mitigation plans provided by the framework.

## 3.3 Mitigation Plan Data Model

Since the mitigation plan data will be sent to a Publish-Subscribe topic, where it can be consumed by different components of the system, there is the need to define a **data model**. This will allow the other components to know how the data of the mitigation plan is structured, as well as help them define how to parse the received information.

This mitigation plan data model will consist in a JSON object containing detailed information about the mitigation plan to be applied, according to the detected fault.

The data model will be structured as follows:

- **faultDescription** [string] - a brief description of the fault that originated the mitigation plan
- **mitigationPlanDetails** [object] - a composed object containing detailed information about the mitigation plan to be applied, containing the **actionDescription**, **faultyComponents** and **recommendedAction** fields
- **actionDescription** [string] - an explanation of what can be done to recover from the fault
- **faultyComponents** [string] - a list of component identifiers from the components that were affected by the fault
- **recommendedAction** [string] - a list of commands to be applied to the system
- **occurrenceTimestamp** [long] - timestamp of when the failure occurred, in milliseconds
- **detectionTimestamp** [long] - timestamp of when the failure when the failure was detected by the framework, in miliseconds

Listing 3.1 presents an example of a mitigation plan for a specific fault to be considered.

```
{
  "faultDescription": "Attempted connections to an external service are timed out
      since the external service is not reachable",
  "mitigationPlanDetails": {
      "actionDescription": "To recover from the present fault, the affected
          component(s) must be restarted",
      "faultyComponents": [
          {"name": "pod-checkout-567"}
    ],
    "recommendedActions": [
        {"command": "kubectl rollout restart deployment pod-checkout-567"}
    ],
    "occurrenceTimestamp": 1278678127781,
    "detectionTimestamp": 1278678127790
  }
}
```

Listing 3.1: JSON example of the Mitigation Plan Data Model

Besides this example being directly related to Kubernetes [56], it is possible to modify the recommendedActions field with commands from other orchestrators, thus making this component "orchestrator-agnostic".

An important concern that must be addressed is the insertion of new plans in the database or any repository considered. If a considerable amount of plans needs to be added manually, this task can become unsustainable. Nonetheless, as observed in Section 3.4 and as seen throughout literature review [97, 28, 88, 89, 36], the amount of recovery actions that could be considered is not something that a human being could not handle. Furthermore, whenever a new recovery action may be considered, it needs to be thoroughly evaluated, to search for possible risks of its usage, in order to avoid damaging the system when trying to repair it. Thus, the new recovery actions can be inserted manually in the database by a system administrator.

The subsection that follows contains the recovery actions that are considered in the proposed solution.

## 3.4 Recovery Actions

Developing autonomic fault responses is a challenging task since there is no single "perfect" fault mitigation action to a given failure. The objectives and state of the applications have a significant impact on the design of the recovery actions. The system's autonomic behavior should also respect the non-functional requirements and service-level objectives imposed and not damage the application's correct service [24]. A service-level objective (SLO) is an agreement within an SLA regarding metrics such as uptime or response time. On the other hand, SLA is an agreement between the service provider and the client concerning measurable metrics such as uptime [7].

That being said, we must be aware of the underlying risks of performing recovery actions in the system [90]:

- **Costly unavailability** - May be a result of incorrect autonomic actuation performed by the autonomic manager. Performing improper recovery actions can damage the system and lead to a worse scenario than if no action would be performed.
- **Unexpected expenses** - Since the cloud follows the pay-per-use pricing model, where users pay for the resources they use, an incorrect recovery action may trigger high resource use, leading to unwanted expenses.

Despite the importance of having appropriate fault diagnosis mechanisms, in order to provide accurate root-cause localization of failures, fault mitigation is another important challenge that must be addressed. In the industry, it is common to give humans the task of solving problems that arise. This is advantageous in the sense that is safer than trying to approximate the appropriate recovery action to apply to the system. However, the human approach is only useful in systems that do not require quick reactions to failures [24]. Nonetheless, it is utterly important to develop appropriate recovery actions and mechanisms for their approximation [97]. This will be of most importance in critical systems that require time-bounded responses, which could take advantage of autonomic actuation [24].

In the context of this thesis, the following recovery actions were considered [90]:

- **restart** - reboots the failed component, in order to start it from a clean state. This

is a common practice in the industry, which is proven to be effective in recovering from failure scenarios [63, 97, 36].

- **version downgrade** - replaces the current service version with the previous one, if one exists. Since a considerable amount of failures that occur in a cloud environment are due to faulty software updates (16%, according to [36]), it is mandatory to consider such recovery action. Even tough developers perform a vast amount of tests before deploying code in production, some failures only manifest in this environment.

Further recovery actions were initially evaluated, but ended up not being considered. For instance, in several papers, recovery actions such as scale up/down [28, 97] and virtual machine or pod migration [97] are presented. However, the orchestrator used (*Kubernetes*) already provides several self-healing actions. Instead of developing a scaling mechanism from scratch, we can take advantage of Kubernetes HPA (Horizontal Pod Autoscaler), which automatically scales services up or down, according to the resource utilization defined. Moreover, whenever one of the worker nodes (virtual machines) where an application is running stops working, the Kubernetes master automatically reschedules and migrates the resources to the available nodes. Whenever a Kubernetes cluster is configured, it is important to guarantee that redundancy is in place. As previously stated, if a worker node fails, its resources can be rescheduled to other worker nodes. However, the same does not apply to the master node. This node is the one in charge of managing the worker nodes and is where the Kubernetes API server and *etcd* run. Thus, a failure on a master node with a single replica can be catastrophic. This requires the existence of additional master nodes to assure correct service in a failure scenario.

Despite the fact that the mentioned recovery actions can help improve the availability of a system, redundancy remains utterly important, ensuring that spare components are ready for service in case of failure [93].

## 3.5 Recovery Strategies

A recovery strategy can be described as an algorithm that combines a set of recovery actions to be applied to a system in a failure scenario. The core motivation for defining recovery strategies is to enhance and combine the strengths of single recovery actions into more robust recovery workflows, that can, for instance, try a different recovery action if the previous one was not successful in mitigating an existing failure.

That being said, two recovery strategies were considered:

- **Global Restart** - this strategy employs the restart recovery action to every service upon the detection of a failure.
- **Iterative Recovery** - this strategy takes advantage of both of the recovery actions considered in this work (restart and version downgrade). However, it uses the algorithm described in Algorithm 1. Upon the detection of a failure, the first action to be applied is a restart. If after a defined period, information about the same failure is received, the version downgrade action is chosen. Finally, if the failure persists after applying the previous action, a notification is sent to the operator. This will provide context to help identify and manually solve the failure detected.

Reflecting on the strategies considered, the *global restart* can become a viable approach if the experiments exhibit a low restart time for the services deployed in Kubernetes. On the

**Algorithm 1:** Iterative recovery strategy algorithm

**Input:** Record containing fault information
**Output:** Mitigation plan to be applied to the managed application
**Data:** Period = $x$ seconds

```
1  Collect status code, name of the components and occurrence timestamp from fault
     information record;
2  if database is not empty then
3      if (status code is equal to status code stored in database) and (components in
         components stored in database) and (occurrence timestamp is less or equal
         than period + last timestamp stored in database) then
4          if version downgrade count is equal to 1 then
5              Create mitigation plan to notify the operator of the unsuccessful
                 recovery attempts;
6          else
7              Create mitigation plan with version downgrade recovery action;
8      else
9          Create mitigation plan with restart recovery action;
10 else
11     Create mitigation plan with restart recovery action;
```

other hand, *iterative recovery* can provide a basic automation workflow where the strategy applies more than one recovery action to try and recover the system. Nonetheless, and as mentioned in the previous section, the risk of actuation must be carefully studied to avoid damaging the system when trying to recover from a failure.

# Chapter 4

# Evaluation of the Proposed Solution

The present chapter documents the methods used to evaluate the proposed solution, in order to assess if it can apply the appropriate recovery actions, as well as the time required to recover the system. For this purpose, fault injection shall be used. It will be performed through code changes at compile-time (software-implemented fault injection), as well as resorting to chaos engineering, using Litmus.

This chapter is organised as follows. Section 4.1 describes the details of the experiments performed, namely the different types of runs executed and the metrics collected, as well as the different components that take part in the fault injection campaign and the workload used. Section 4.2 contains the different faults considered, each of them consisting in a fault identified in Table 2.1, in Section 2.3, and the corresponding description of the method to inject the desired fault. Lastly, Section 4.3 contains a description of the experimental setup where the experiments were performed.

## 4.1 Experimental runs

The experimental runs performed to evaluate the proposed architecture encompass two phases, the **golden run** and the **faulty run**. The golden run is intended to gather metrics from the system and evaluate its normal state. Throughout a golden run no fault is injected. On the other hand, the faulty runs are those where the faults listed in Section 4.2 are injected into the system. Only one fault is injected per run, to assure that we can study the effect of each scenario individually, otherwise we could face undesired aftermath, such as interaction between different faults. A faulty run consists of the following three steps [18]:

- **warmup** - step intended to stabilize the system and taking place in the first 30 seconds of each run.
- **peak** - period where faults are injected into the system. This is the only period when faults are injected and occurs during the 90 seconds that follow the warmup step. This duration considers additional maintenance operations required, such as starting up pods.
- **cooldown** - step with the objective of assuring that there is enough time for the fault to propagate in the system and become a failure. This way we can achieve a fault effect in every run. This step lasts for the final 30 seconds of the faulty run, which also provides enough time for the system to stabilize.

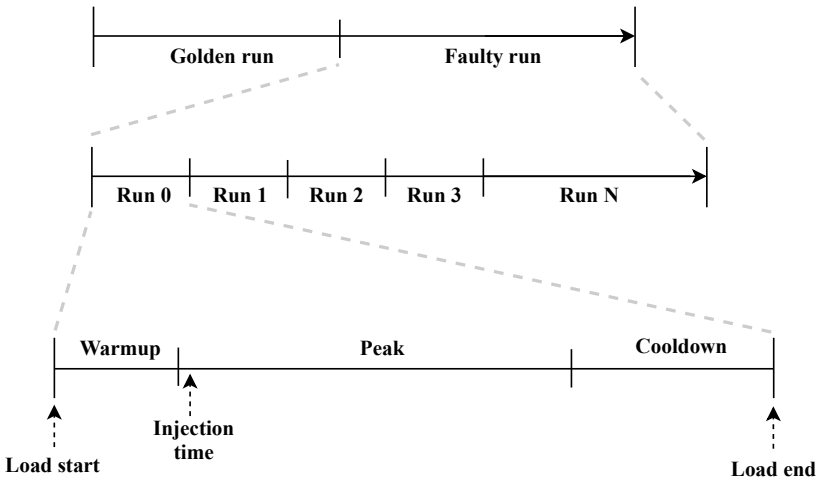


Figure 4.1: Experiments execution profile.

A representation of the different experimental runs can be observed in Figure 4.1

During each run, a set of measurements are collected from the system to help analyse its performance and behavior. The metrics collected are:

- **service logs** - every container running inside a pod produces logs, which are collected and saved for post-run analysis.
- **spans and traces** - the spans produced during each run are saved to the Elasticsearch database, which is the backend storage for Jaeger. After each run, this database is queried to retrieve spans that contain fault information within the time period of the run.
- **prometheus metrics** - *Prometheus* is a tool for metrics collection. It uses metrics from *Node exporter* (*Prometheus* exporter which provides host-related metrics) and *Kube-state-metrics* (service that listens for the Kubernetes API server and generates cluster level metrics). The metrics collected from Prometheus are container memory usage, container CPU utilization, number of running containers, number of available containers and container start time.

An overview of the workflow of an experimental run is depicted in Figure 4.2.

The **controller** is the component which iterates over the experimental runs of a fault injection campaign and stores the results for later analysis [66]. The **monitor** entity gathers raw data (*e.g.*, measurements) from the target system (Stan's Robot Shop). These components consist in Python scripts, which interact with the **fault injector** and the **load generator**, in order to manage the workflow of the experiments.

The **fault injector** is in charge of injecting the desired faults in the system and it is also a Python script. Moreover, the **load generator** that provides the workload for the experiments is the Robot-Shop load generator. The **workload** are the inputs submitted to the application. The load generator uses *Locust*, which is a load testing framework. It runs for 150 seconds, with 10 concurrent clients and a spawn rate of 1 client per second. These values were chosen according to the performance of the system and the available resources, having tested different combinations until no erroneous monitoring data was present in the golden runs.

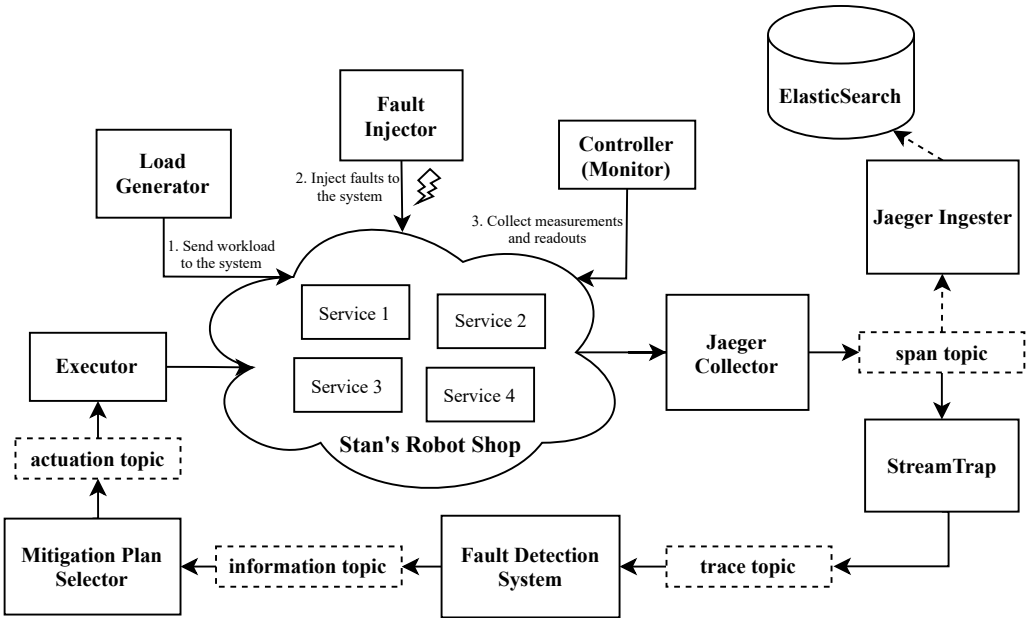


Figure 4.2: System and Experimentation overview

At the end of each run, all the metrics previously mentioned are stored for further analysis. Additionally, the remaining messages that are stored in the Kafka topics, as well as the topics, are deleted and recreated. This is done to ensure that no data from the current experimental run affects the results of the forthcoming runs.

## 4.2 Faultload

The *faultload* are the different faults that are injected into the target system. These faults are performed in a *fault injection campaign*, which consists in the several experiments that are performed [66]. This faultload was chosen according to several considerations - they represent typical faults that are common in microservices applications or cloud systems; they have considerable impact on the service provided by the system, thus affecting its availability; they can be applied to more than one service, thus allowing to generalize the conclusions obtained [20].

The remainder of this section provides a detailed description of the faults presented in Table 2.1, as well as the fault injection method used in each scenario.

### 4.2.1 Crash

A crash failure occurs when an application or a system stops working. This is a very common scenario in cloud applications [36], having direct impact on the service availability, thus the importance of addressing this failure scenario. To simulate this type of failure, an instruction is added to the source code of the catalogue service, at compile time. The instruction considered is *throw new Error("error")*, in Javascript, which throws a Javascript error, leading to a crash. Initially, the payment service was the target of the experiments for the crash failure. However, this service is deployed using uWSGI, which is an HTTP server, where the payment service, implemented in Python, runs. When a crash was injected using *sys.exit()*, in Python, it would throw a *SystemExit* exception that kills the process running

inside the container. However, this server automatically restarted the process when this was terminated, which would influence the experimental results.

### 4.2.2 Hang

A hang happens when one service stops answering requests and does not produce any further output. One possible cause could be a missing response from an upstream service, from which the present service hangs while waiting for the response. This scenario can lead to timeouts, if such are configured. Since this scenario can have a significant impact on system availability, it must also be considered.

To perform the simulation of this failure scenario, a code change was implemented. A *sleep()* instruction was inserted into the source code of the payment service, written in Python. This enables the representation of an hang scenario where a service would be overloaded with requests or would be performing a longstanding operation and not answering to requests.

### 4.2.3 Wrong result

In this scenario, as the name states, the service produces syntactically correct content, but with wrong values. Similar to the previous scenarios, a code change was performed in order to replicate this failure. In this specific scenario, the ratings service, instead of returning a rating value between 0 and 5, returns the value 6, which is outside the expected range. Despite being a valid value, its content is incorrect and can affect further operations that use the computed value.

This type of errors in the interaction between services or call statements are called *interface errors* [25]. Since it is through the system's interfaces that failures can propagate, such interface faults can result in cascading failures that affect downstream services that rely in this service's response.

As stated in [66], interface error injection enables the injection of errors that corrupt the input and output values which a component exchanges with other software components, the hardware and the environment. Error injection at the output values is used to emulate the outputs of faulty components, as well as assess their impact on the rest of the system [66]. This form of fault injection, which is sometimes referred to as interface error injection or failure injection, is also referred to as fault injection since the corrupted values can be seen as external faults, according to the definitions of Avizienis [66].

The criteria used for interface error injection answers to the following questions [66, 52]:

- **What to inject?** - The invalid value to be injected can be generated using several methods, such as fuzzing (replacing a correct value with another that is generated randomly), bit-flipping (inverting one of the bits of a value in order to corrupt it) and data-type based injection (replacing a valid value with an invalid one, based on the type of the parameter being corrupted).
- **Where to inject?** - Errors can be injected at the input or output parameters of the software interface.
- **When to inject?** - Injection can occur when a service of the target application is invoked.

### 4.2.4 Corrupted output

A corrupted output scenario can occur when a service returns a corrupted result, which does not comply with the expected data type. This scenario is another type of interface error injection.

In order to reproduce such scenario, a code change is performed. Instead of returning the total value of a purchase, which would be an integer, a string is returned by the cart service. This service starts by returning an invalid output value, which is stored in the Redis database, as well as in the session variable that stores user data. The following service, which is the shipping service, retrieves the cart object to store the shipping information. However, since the total order value is not used, this fault is not yet detected. Finally, the payment service will retrieve the cart from the request and uses the total value to perform an operation. When this occurs, the fault is triggered, thus reaching a cascading failure scenario.

### 4.2.5 Memory Leak

In this final scenario, a memory leak is considered. As stated in [54], this is one of the major software bugs which threatens the availability of software systems. A *memory leak* exhausts system resources, which can lead to system crashes [54].

To perform this scenario, the Memory Hog experiment from Litmus was used. This experiment exhausts the memory resources of a pod in Kubernetes, according to a configurable amount of memory to be exhausted.

Litmus, the selected chaos engineering experiment, has two main components, the *control plane* and the *execution plane*. The first one contains the components that enable the operation of the Chaos Center, which is the website-based portal for Litmus. This component also performs the post-processing and analysis of the experiment results. The execution plane is the component that manages the experiments and holds the different components which perform the chaos injection in the target resources.

When a chaos experiment is performed [62], the *control plane* sends the workflow manifest to the *execution plane*. This manifest is then used to create the *chaos workflow* custom resource. A custom resource can be seen as a custom object that allows to extend the capabilities of the Kubernetes API. When the *workflow controller* finds the new workflow custom resource, it creates the *chaos experiment* and *chaos engine custom resources*. The *chaos operator* proceeds to choose the *chaos engine* that is ready to perform the chaos experiment and creates the chaos runner. The latter reads the experiment data, constructs the jobs that will inject chaos in the target service(s) and monitors them until completion. Finally, the *chaos engine* gets the experiment result and sends it to the control plane.

## 4.3 Experimental setup

The experimental setup was deployed in a virtual private cloud on Openstack, composed of six virtual machines. Four of these machines sustain the Kubernetes cluster - one VM runs the master node, which manages and coordinates the worker nodes; the workers run in three other machines and it is where the managed application and the remaining services of the proposed solution run. In which concerns the two remaining machines, one runs the ElasticSearch database, to where spans are persisted. The final one acts as a bastion host.

| | CPU (vCPU) | RAM (GB) | Disk (GB) | Operating system |
|---|---|---|---|---|
| Kubernetes Node (x4) | 8 (each) | 16(each) | 100(each) | Debian 10 |
| ElasticSearch | 4 | 8 | 100 | Debian 10 |
| Bastion | 4 | 8 | 100 | Debian 10 |

Table 4.1: Infrastructure configuration

This VM is directly exposed to the Internet and, in a normal scenario, has the purpose of withstand attacks. However, in this scenario, it is used to access the worker nodes that run on the internal network of the virtual private cloud.

In which concerns the specifications of the machines, all of the four VM's that composed the Kubernetes cluster have 8vCPUs, 16GB of RAM, 100GB of storage and run Debian 10 operating system. The ElasticSearch and bastion host machines have 4vCPUs, 8GB of RAM, 100GB of storage and also runs Debian 10 operating system. Table 4.1 provides an overview of the specifications of the experimental infrastructure.

# Chapter 5

# Implementation

The present Chapter documents the implementation of the solution proposed in Chapter 3. The following sections contain a detailed description of the architecture's components, as well as other underlying tools and data that help sustain the desired functionalities of the solution.

The proposed solution focuses on the *Planning* phase of the MAPE-K loop. Nonetheless, it was necessary to develop additional components, in order to obtain a fully functional feedback loop. These components are *StreamTrap*, *Fault Detection System* and *Executor*, as well as the streaming deployment strategy of *Jaeger*. All of these components use Kafka as their middleware, helping them exchange data between each other, and sustain the required information for the Mitigation Plan Selector to analyse the existing fault information and provide the appropriate recovery actions to be applied to the managed system.

Jaeger, alongside Istio service mesh, helps monitor the system, gathering spans from the target microservices application (Stan's Robot-Shop). Spans contain metadata from each microservice in the invocation chain, helping collect relevant information to monitor the system. The loop starts with spans being sent to the tracing backend by the sidecar attached to each service. Once the tracing backend receives the traces, it batches the desired amount of spans before publishing them to the span topic. Once the desired amount is reached, the spans are sent in a batch of messages to the defined topic, to be consumed by the StreamTrap component. The latter is in charge of sorting the received spans and reconstructing the traces that these spans belong to. The resulting traces are then sent to the trace topic and read by the Fault Detection System. This component is in charge of analysing the traces received and identifying the ones which contain HTTP codes in the 400 and 500 ranges. When identified, a record with fault information is composed. This record collects the name of services present in the traces and other important metadata, such as the communication protocol and the occurrence and detection timestamps. The resulting record is then sent to the Mitigation Plan Selector, which will select a recovery action for the failure identified. Lastly, this plan is sent to the Executor, which applies the selected recovery action to the services present in the mitigation plan.

An overview of the implementation described is presented in Figure 5.1.

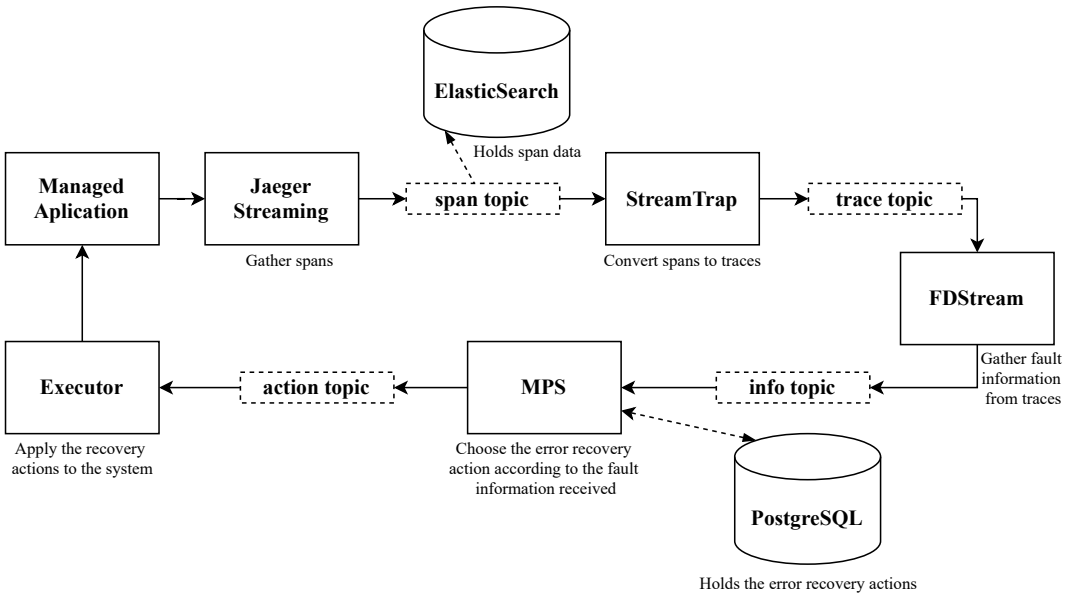


Figure 5.1: MAPE-K Loop Implementation

## 5.1 Components

This section presents a detailed description of each component, to help better understand how the elements of the architecture implement the desired functionalities.

### 5.1.1 Istio

*Istio* is the service mesh used to provide observability of the managed application. It works by attaching a sidecar to each pod of the managed application. The Envoy proxy from each sidecar enables reliable message delivery and can implement a set of relevant mechanisms such as service discovery and circuit-breaking. This proxy also helps instrument the requests with the required headers to enable distributed tracing.

Istio provides two different ways to integrate with tracing backends: Envoy or Mixed based. The chosen way was to use Envoy since it is already compatible with Jaeger, which is a distributed tracing backend compatible with Zipkin-API.

With this approach, the sidecar proxy sends the required tracing information to the tracing backend, which runs on port 9411. As stated in Istio documentation [47], Envoy is in charge of:

- generating request IDs and trace headers (for example, x-B3-TraceId) for the requests that go though the proxy. The trace headers contain the trace context, which has references to other spans and traces.
- generating trace spans for each request according to its request and response metadata.
- sending the generated trace spans to the tracing backend.
- forwarding the trace headers to the proxied application.

The initial headers are only generated by Istio if they are not provided by the request.

Such headers are useful to propagate trace context across service boundaries. The trace context contains references to other spans and traces, such as their SpanId, TraceId and ParentSpanId, which are used to refer to spans and traces across a process boundary. This allows to gather individual spans and stitch them together to be able to have an overview of the traffic flow [47].

### 5.1.2 Jaeger

*Jaeger*, a software to trace the communication among distributed services, plays an important role as a way to monitor the managed application, alongside *Istio* service mesh.

Jaeger provides several deployment strategies in Kubernetes - *allInOne*, production and streaming. The one that suits the present architecture is **streaming**, since it enables continuous data streaming to the StreamTrap component. It is composed by several components - agent, collector, ingester and query - decoupled from each other, which allows each component to be scaled independently:

- **Agent -** network daemon, which is placed in every node of the cluster. It listens for spans that are sent via UDP, which are then sent to the collector.
- **Collector -** receives traces from all the Jaeger agents and runs several processing tasks, such as trace validation and indexing, before storing them. As backend storage, either Cassandra, ElasticSearch or Kafka can be used.
- **Ingester -** reads data from a Kafka topic, which it then stores in another storage backend (Cassandra or ElasticSearch).
- **Query -** is a service intended to retrieve traces from storage and display them in the UI that it provides.

With this approach, Kafka acts as an intermediate buffer to help reduce the pressure from the backend storage (Figure 5.2). Instead of having Jaeger Collector writing spans directly to the backend database, they are first sent to the Kafka buffer. From this topic, data is consumed by Jaeger Ingester and further stored in ElasticSearch.

Despite the streaming deployment using Jaeger Agent to listen for spans, in the proposed architecture Jaeger runs agentless. This is intended since the Istio sidecar, which is attached to each pod of the managed application, already sends tracing information to the Jaeger Collector.

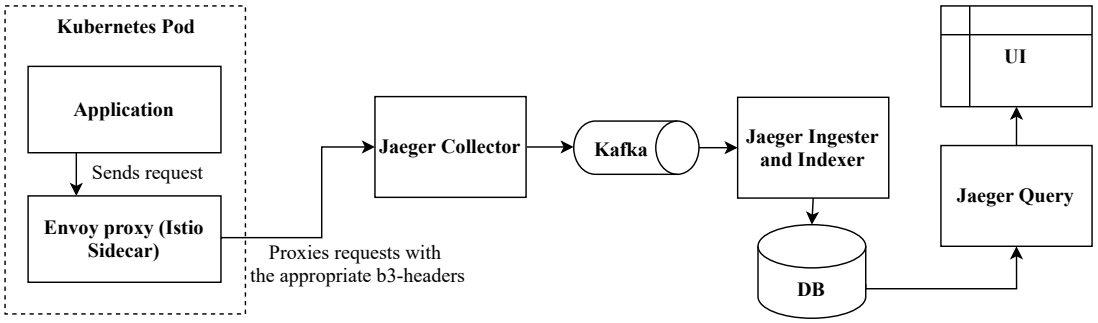


Figure 5.2: Jaeger Streaming Deployment Strategy. Figure from [48]

Since Kafka is the chosen message-oriented middleware, we must define a Consumer Group for the StreamTrap deployment different from the one chosen for Jaeger Ingester. With this configuration, both consumers can read from the Kafka topic independently, without consuming each other's messages.

### 5.1.3 StreamTrap

This component handles the conversion of spans into traces. It starts by collecting a batch of messages from the Kafka topic where spans are produced to, with each message containing a span. The batch contains a minimum of 30 records and a maximum of 75 records, to assure that there is enough data (spans) to reconstruct the traces. Different batch sizes were considered and the present one was chosen as the most appropriate trade-off between standby latency and complete traces. The standby latency is the time it takes to batch the referenced number of records (spans). A complete trace is a fully reconstructed trace with all spans in its invocation chain, without losing any spans.

The batch of spans initially collected is saved to a file in jsonl format, containing a span per line. Then, the sorting process takes place. It receives as input the span file and scans its content, in order to sort the spans by their traceId, which are saved to a new file. This output file is browsed in order to reconstruct the traces with the spans contained in the file. It then gathers all the spans which have the same traceId and uses them to reconstruct the trace. To perform this reassembly, we must keep in mind that a trace, according to OpenTracing [72], can be seen as a **directed acyclic graph (DAG)** of spans (Figure 5.3). To elucidate, a DAG is a graph in which all edges have a direction, such that it is not possible to find a path that leads back to the same node.

The graph is then iterated in order to reconstruct the traces. The component starts by searching the parentIdentifiers within each span. If the span has no parentIdentifiers, it means that it is a root span. Otherwise, we iterate over the parentIdentifiers and, if they are not null, we add them as neighbors of the parent of the current span (node). If, once again, the parent of the current node is null, we add it as root. If the array of roots is null, we consider that there are missing nodes, since the graph always needs to have a root node.

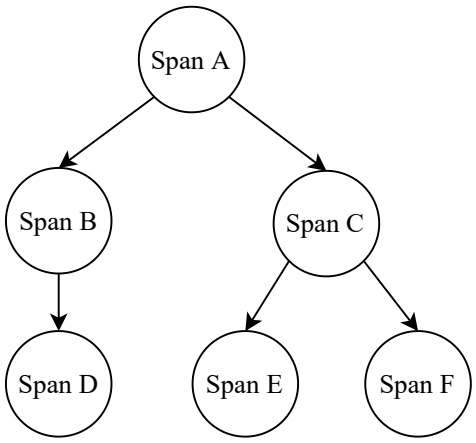


Figure 5.3: Directed Acyclic Graph

Finally, the reconstructed traces are published, one by one, to the Kafka topic from where the Fault Detection System will consume.

```
{
  "faultCategory": "Server-side error",
  "faultDetails": {
    "description": "503",
    "protocol": "HTTP",
    "causalRelationships": [
      "serv1->serv2"
    ]
  },
  "faultyComponents": [
    "payment"
  ],
  "detectionTimestamp": "1623024581053479",
  "timestamp": "1623024599683"
}
```

Listing 5.1: JSON example of the Fault Information Object

| Field | Description |
|---|---|
| traceId | unique 64 or 128 bits unique identifier for a trace. Is set in every span belonging to the trace. |
| spanId | unique 64bit identifier for the operation within the trace. |
| parentId | identifier of the parent span. Null if the it is the root span. |
| duration | duration of the operation associated with the span, in microseconds. |
| timestamp | Epoch microseconds that mark the start time of the span. |
| annotations | Describes events that justify latency with a timestamp. |
| tags | Adds contextual information to a span. Examples include the HTTP status code of the request, the service that made the request and the one to whom the request was made. |

Table 5.1: Span specification [103]

### 5.1.4 Fault Detection System

The present component starts by collecting traces from the traces topic. These are then parsed and a search for HTTP status codes that represents an error code, either in the 400 or 500 range, is performed. Whenever a match is found, the required information about the existing failure is retrieved from the trace. This information contains the name of the services involved, the status code identified, the timestamps of occurrence and detection and the communication protocol. Since a trace is composed of one or more spans and each span contains information as exhibited in Table 5.1, the data is collected from each span. The only data that is not collected from the span is the detection time, which is registered by the framework whenever the present component finds a match in a span. With this data, a record with fault information is created and sent to the topic, to be further consumed by the Mitigation Plan Selector. An example of the information that the record contains is shown in Listing 5.1.

This component only considers distributed tracing data as monitoring data, but can further be extended to accept other information, such as logs from services or other metrics collected from the system. This is possible due to the decoupling that the message oriented middleware provides and the fact that the monitoring information comes from external sources.

### 5.1.5 Mitigation Plan Selector

This component reads the existing fault information records from the info topic. This information provides a description of the failure, the names of the affected components and the detection and occurrence timestamps. After parsing this data, this component fetches a mitigation plan from the Postgres database. The mitigation plan is then completed with the fault information, prior to being sent to the action topic. The mitigation plan also contains the recovery action that must be applied to the affected components identified in the fault information record. A plan similar to the one in Listing 5.2 is then sent to a topic, from where the Executor will read the plan.

This component can apply both the recovery actions and strategies previously described in Sections 3.4 and 3.5.

```
{
  "faultDescription": "503",
  "planTimestamp": "1626169711688",
  "mitigationPlanDetails": {
    "actionDescription": "To recover from the present fault,
    the affected component (s) must be restarted",
    "recommendedActions": [
      {
        "action": "restart",
        "command": "kubectl rollout restart deployment payment"
      }
    ],
    "faultyComponents": [
      {
        "name": "payment"
      }
    ]
  },
  "detectionTimestamp": "1626169711573675",
  "timestamp": "1626169711668"
}
```

Listing 5.2: Mitigation Plan to apply a restart after payment service crashed

### 5.1.6 Executor

The last component of the feedback loop is the Executor, which reads the mitigation plan from the topic and applies the recovery actions to the system.

To apply the recovery actions, the Executor communicates with the Kubernetes API Server. This server exposes an HTTP API that allows users, different parts of the cluster and external components to communicate. It also enables querying and manipulation of the state of API objects in Kubernetes. Taking advantage of several Java libraries presented in the Kubernetes documentation, the component communicates with the API Server to apply the restart and version downgrade actions to the system when needed.

The **restart** action is performed by scaling down to zero the number of replicas and scaling up to one the number of existing replicas of the intended deployment.

In which concerns the **downgrade version** action, it is done in two steps. First, we verify if there is any previous revision of the target deployment, *i.e.*, we verify if there is any previous version of the target service. If so, we change the current container version to the previous one. To clarify, a revision is created when a new rollout occurs. This happens when the *.spec.template* of a pod is changed, for example, by replacing the container's image. This template specification describes the data that a pod should contain if it is

created from a template.

If a recovery action cannot be applied to the system, a notification record is sent to the notification topic. This record allows to inform the system administrators and developers that an action could not be applied and what prevented it from being applied, such as not finding the service or that the service does not have a previous version to downgrade to.

As previously shown in Figure 5.1, these components compose the MAPE-K loop, thus enabling to perform autonomic actions to the system.

### 5.1.7 Apache Kafka

Apache Kafka [4] is the middleware that enables the communication among the different components of the architecture. It is a distributed event streaming platform where clients and servers communicate over a high-performance TCP network protocol. It usually runs as a cluster, consisting of one or more servers, that can be deployed in different datacenters. Each of these servers is a Kafka broker. The existence of multiple brokers allow for redundancy, failover and load balancing.

Furthermore, it helps guarantee that no data is lost, which is crucial since we need to receive all the information about failures in the system, to be able to perform the appropriate recovery actions. To assure that no data is lost, we take advantage of *Kafka* retention policies, which can keep records for the amount of time or the maximum size of data configured.

*Kafka*, by default, provides an *at-least-once* delivery semantic through the appropriate management of consumer offsets. With this semantic, a *Kafka* consumer uses an automatic commit policy, which triggers an offset commit periodically. Whenever an application, acting as a *Kafka* consumer, successfully consumes a message from a topic, it updates the consumer offset, which is saved and stored in a separate topic called "__consumer_offsets". This ensures that, if there is a failure and the consumer does not receive the message, it can consume it again since the offset was not updated. With this approach, no messages will be lost, but duplicates are possible.

However, this delivery semantic does not meet the requirements of the proposed solution. If we receive duplicated messages containing mitigation plans, we can subject the system to unnecessary actions which can lead to downtime. Thus, **exactly-once delivery** must be implemented. The first step is to configure the producer to be *idempotent*. By turning on idempotence, each Kafka message will contain two additional fields, the producer id (PID) and the sequence number (seq). This way, upon a client or broker failure, if a message is sent another time, the topic will only accept messages with a new sequence number and producer id. To guarantee that each message is processed exactly-once, we resort to *transactions*. With this approach, either the full operation is successful or a rollback is issued. Additionally, the consumer must be configured to be transactional. This is possible by setting the *isolation level* to *read commited*. This way, the consumer will only read messages that were already commited. Finally, we need to commit the offsets of the messages consumed back to the Kafka topic and send them to the producer transaction. The operation finishes when we commit the transaction [10, 26].

Zookeeper is deployed alongside Kafka because it manages and coordinates the Kafka cluster. It keeps track of the status of the nodes in the Kafka cluster, as well as the messages and topics in Kafka.

Each topic has two partitions, but it is possible to increase the number of partitions to

be able to have several consumers, in parallel, handling data consumption from the topic. However, for the current experimental scenario, a single consumer was sufficient.

To read from the span topic, the two existing consumers (one from the StreamTrap and the other from the Jaeger Ingester) must belong to different ConsumerGroups. Having each consumer in a different ConsumerGroup allows each of the two components to read the same messages from the topic, instead of parallelizing data consumption. This is the intended scenario since we want each of the two components to perform different operations on the data consumed: Jaeger Ingester will read data from the Kafka topic and persist it to the backend storage (ElasticSearch) and StreamTrap will read a batch of spans, which will be parsed and composed into traces.

Further configurations to Kafka include enabling topic deletion to help clean data from topics during the experiments. The topic retention time was set to one day instead of the default time of seven days. The topic retention policy was changed from compact to delete. What this does is that, in the default Kafka configuration, log compaction is performed, where the last known value for each message key within the log data for a topic partition is always retained. With the delete policy, the old segments will be discarded when their retention time or size limit is reached.

### 5.1.8 PostgreSQL

PostgreSQL is the relational database that hosts the error recovery data. A relational database was chosen since the schema would always be the same, thus the option to choose a NoSQL database was discarded.

This database contains a table which holds the mitigation plans and is composed of six columns:

- **id** - auto-incremental integer to distinguish the different table records
- **action description** - description of the recovery action to be performed
- **recommended action** - command to be applied to the system
- **action** - recovery action to be applied
- **fault category** - category of the detected fault
- **service type** - type of service that the recovery action will be applied to

This database is queried by the Mitigation Plan Selector whenever it needs to create a mitigation plan.

### 5.1.9 ElasticSearch

*Elasticsearch* [27] is a widely used distributed search and analytics engine. It provides storage, as well as querying and analyzing vast amounts of data in near real-time.

In the proposed architecture, this component is the backend database where the spans will be stored. The Jaeger Ingester will fetch the spans from the Kafka topic and persist them to this database. From this storage, Jaeger Query will fetch the data to be displayed in the Jaeger UI, available in the browser.

### 5.1.10 MongoDB

MongoDB is a document-oriented NoSQL database. This type of databases are known for not having strict schemas as the ones found in relational databases, such as PostgreSQL and MySQL. And this was also the main reason for choosing this database to store the information related to the **iterative recovery** strategy. Since different approaches were being tested and the structure of data was constantly changing, it was more suitable to use MongoDB.

In MongoDB data is stored as JSON. Inside a database in MongoDB, data is stored in collections, which is the equivalent to a table in SQL. However, this does not have a strict schema as observed in relational databases. A collection is a group of documents. Each document can be compared to a row in a table.

This database stores information related to the failures that occur in the system. This helps provide the required information to the algorithm of recovery strategy previously defined in Section 3.5. The information required, which is stored in a document, is the following:

- **id** - unique identifier used to identify each document stored in the collection.
- **last_ts** - the epoch timestamp, in microseconds, of when the failure occurred. This timestamp is collected from the trace.
- **status code** - the HTTP status code from the failure that was identified. This data is collected from the trace with information about the fault.
- **components** - the components from the managed application that were considered as culprits. This information is gathered from the trace collected.
- **applied actions** - object that stores the different actions that were applied to the system and how many times they were applied.

### 5.1.11 Kubernetes

Containers by themselves cannot assure that they are running as intended or that the load has increased and it is required that the number of running containers to accommodate the exceeding traffic. As such, a technology to manage the deployment of containers, their scalability and management is required. For this purpose, the proposed solution resorts to *Kubernetes* [56].

In a Kubernetes cluster, the pods are hosted in the worker nodes. Every node in the cluster contains two components that enable the corresponding node to run pods - the **kubelet** and the **kubernetes proxy**. The first communicates with the Kubernetes API server to receive pod specifications, which provide information to assess if they are running as intended. It also reports to the *control plane* with information about the pod's health and status. The latter (Kubernetes proxy) is the network proxy that is located in each node in the cluster and is in charge of handling the communication to the pods running on these nodes.

In the master node(s) we find the **control plane**, which is the "brain of the operations" in Kubernetes, making decisions about the cluster and responding to any events, such as creating new resources. It is composed by the following five components. The **API server** exposes the Kubernetes API, which is used by all the resources in the cluster to

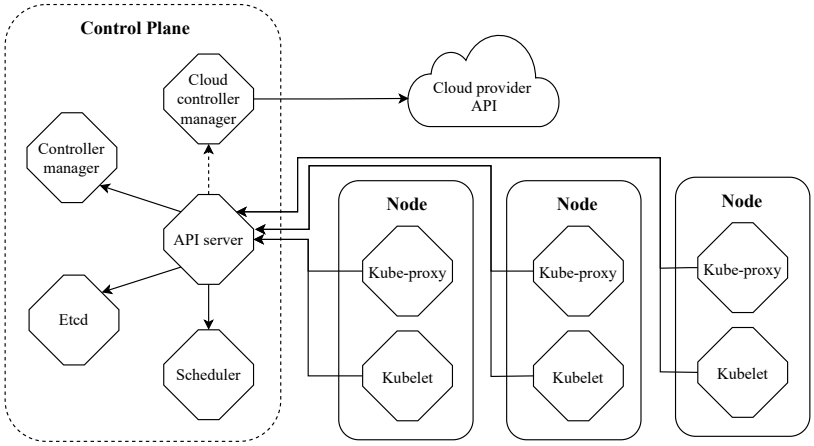


Figure 5.4: Kubernetes Architecture Overview [58].

communicate. **Etcd** contains all the data related to the cluster and can be used as the single source of truth. **Scheduler** designates the nodes with newly created pods, thus handling the scheduling logic. **Controller manager** runs the controller processes which monitor the cluster state to assure that the state of the cluster matches the desired state. Finally, the **cloud controller manager** runs the controllers that provide the link between the cluster and the underlying cloud providers.

Figure 5.4 provides an overview of the Kubernetes architecture.

# Chapter 6

# Results and Analysis

This chapter documents the results and analysis over the data gathered from the fault injection campaign in Chapter 4. The present chapter is split in four subsections. The first one provides the results from the performance of the recovery actions. The second one describes the analysis of the recovery actions' effectiveness. The third one describes the effectiveness of the recovery strategies considered. The fourth and final one provides the answer to the research questions from Section 1.4 in Chapter 1.

## 6.1 Performance of the recovery actions

The autonomic architecture developed is intended to assure the high availability of cloud-native applications. Thus, and as stated in Research Question 2, we want to assess if we can achieve an availability of 99.99% with the recovery actions selected. This can be attained if the solution presented provides a relatively low recovery time, that allows to perform recovery actions upon existing failures, and still remain under the threshold of high availability.

To compute the **Mean Time To Recovery (MTTR)** of the present solution, we only require the timestamp of the failure occurrence and the start up time of the failed Kubernetes resources upon applying the recovery actions. However, to evaluate the performance of the solution, five timestamps were collected, to provide a more granular view of each phase's duration:

- **Injection time** - timestamp when the fault is injected into the system by the controller of the experiment.
- **Occurrence time** - timestamp when the fault occurred, which is provided in the trace record that is collected from the Kafka topic.
- **Detection time** - timestamp when the fault is detected by the Fault Detection System. Since this is the component which analyses the traces to search for error codes, the detection timestamp is collected when an error code is found.
- **Plan time** - timestamp when the mitigation plan is composed, after choosing the recovery action to be applied to the system.
- **Action time** - timestamp when the recovery action is applied to the managed application.

- **Container start time** - timestamp when the container started, after being submitted to the chosen recovery action.

With the timestamps collected, additional metrics were considered. The *detection time (MTTD)* which is the average time required to detect and diagnose failures in the microservices application. The *wait time* is the period when spans are batched before they are fed to the StreamTrap component. It is measured as the difference between the occurrence timestamp and the timestamp of the arrival in the StreamTrap component. The *execution time* is the execution time of the framework, from the moment that the fault is detected until the recovery action is applied to the system.

The occurrence, detection, plan and action timestamps collected are collected by the architecture components and saved to the PostgreSQL database by the Executor component, after applying the recovery actions. This is done to avoid any additional overhead from database communication when collecting these timestamps. The injection and container start time are collected by the experiment controller. The injection timestamp is saved upon fault injection. To save the container start time, the controller uses *kubectl* to fetch the container start time. This data is saved to a Comma-Separated Values (CSV) file, alongside other experiment information, such as system metrics (CPU and memory usage), spans and traces.

In order to gather enough experimental data, 100 experimental runs were performed for each of the failure scenarios considered. The information collected was analysed, in order to remove any existing outliers, and is displayed in the graphics presented in the following subsections.

An important aspect that must be considered is that the services that compose the managed application are distributed among several virtual machines in a datacenter. Thus, we need to ensure that the local clocks of each machine are synchronized, otherwise we risk affecting the results of the time measurements performed in the different experiments. This synchronization works to minimize clock skew (time difference between two clocks) as much as possible. To perform this task, the local clock of each machine is synchronized with the NTP (Network Time Protocol) server provided by the datacenter where the virtual machines are hosted. This is the standard protocol used to synchronize computer clock times in a network.

Briefly explaining, when a client, such as an host machine, synchronizes with an NTP server, a set of requests and responses are exchanged [69]. The client starts by sending an NTP query to the server with the *origin timestamp*, which is the client's system time when the query is sent. When the server receives the request, it stamps the *receive timestamp*. It then sends the response with this timestamp and the *transmit timestamp*. Finally, the client receives the response and registers the *destination timestamp*, which is the timestamp when the response is received. The timestamps gathered help determine the difference between the internal clock and the time provided by the server, taking into account the round trip delay (the time it takes for the message to travel to the server and back), thus maintaining synchronization.

### 6.1.1 Crash

The first scenario considered when evaluating the performance of the framework is a crash in the catalogue service. The observation that strikes us at first when performing an analysis of Figure 6.1 is that the MTTR of the **restart** action is lower than the one

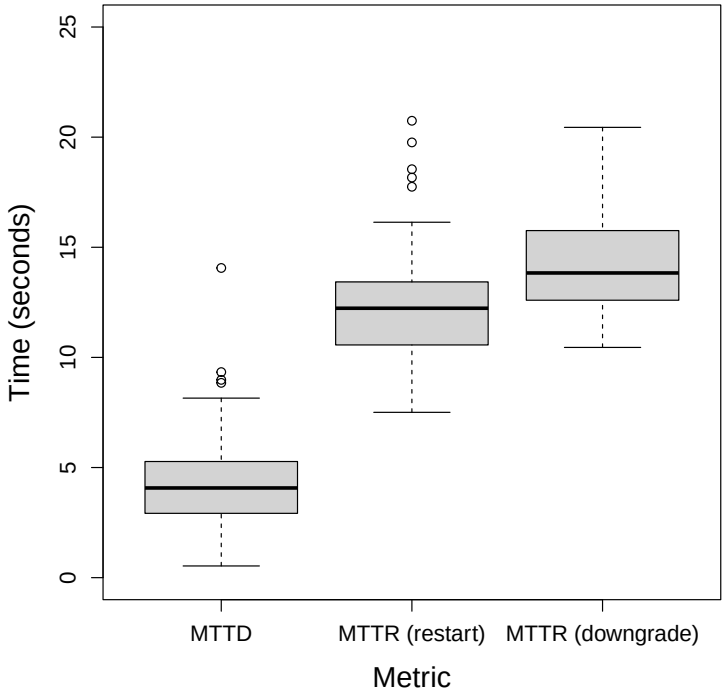


Figure 6.1: Performance in the crash catalogue scenario

| **Metrics** | **Mean** | **Median** | **Q1** | **Q3** | **Min** | **Max** |
|---|---|---|---|---|---|---|
| MTTD | 4.287 | 4.072 | 2.920 | 5.273 | 0.531 | 14.060 |
| MTTR (restart) | 12.150 | 12.141 | 10.450 | 13.255 | 7.506 | 19.761 |
| Execution time (restart) | 0.614 | 0.603 | 0.510 | 0.667 | 0.426 | 1.088 |
| MTTR (version downgrade) | 14.19 | 13.830 | 12.620 | 15.760 | 10.450 | 20.440 |
| Execution time (version downgrade) | 1.239 | 1.208 | 1.159 | 1.277 | 0.963 | 1.673 |
| Restart time | 7.375 | 7.307 | 6.602 | 8.101 | 5.434 | 10.483 |

Table 6.1: Standard statistics of the crash catalogue experiment

from the **version downgrade action**. Since the version downgrade action starts by checking if there is any previous deployment revision to rollback to and only then applies the downgrade action, the difference in recovery time is justified. Nonetheless, both recovery actions present similar performance with restart taking 12.150 seconds, on average and version downgrade taking 15.890 seconds, on average. However, observing Table 6.1, we can see that, in the *Min* column, which contains the minimum values, the restart can recover in 7.506 seconds and the version downgrade in 10.450 seconds. The use of boxplots provides a better view of how the data is distributed. Since the median splits the box of the MTTD in half, we can observe that the data is roughly symmetric. However, in the other two plots, some skew is observed. This shows that, for instance, in the box concerning the restart data, the data has positive skew, which means that the values in the upper part of the box are closer together, and not that the sample size is smaller in that area. Finally, regarding the variability of the data, since the size of the interquartile range (grey box which contains the middle 50% of the data) is narrow in all of the boxes boxes, it shows that the data does not vary too much.

Taking a closer look at the standby time, execution time and restart time, we can observe that:

- The Mean Time to Detect a failure in this scenario is 4.287. seconds. This metric measures the time between the occurrence of the failure in the managed application and the timestamp of when the framework detected it. This value is mostly conditioned by the process of batching spans. However, this operation is required in order to reconstruct traces and have the full service invocation call. This was the defined approach for the detection process, *i.e.*, to consider all spans in a trace as culprits. Nonetheless, in a future implementation, this task can be replaced with a more advanced root-cause localization and identification of the culprits. To define the number of spans that should be used for a single batch, experiments were performed with different batch sizes and the corresponding standby time was evaluated. It started with higher values (minimum of 250 and maximum of 500 spans), where the average standby time was 29.891 seconds. Then this range was decreased to 100-250, which resulted in an average of 12.771 seconds. Since this time was still not acceptable, the final experiments were performed with 30-75, reaching an average time of 4.154 seconds. Additionally, in the different experiments similar slicing of the traces was observed. Thus, the smallest batch size was chosen to reduce the overhead, which had a noticeable influence in reducing the Mean Time To Detect. It is possible to conclude that the biggest overhead for the mean time to detect comes from the span process and the remaining time corresponds to the time that the framework takes to notice the failure.

- The time required for a container to start is 7.375 seconds, on average. This low restart time presents one of the benefits of the utilization of container-based virtualization. However, the restart value depends on the infrastructure where the managed application runs. A more powerful infrastructure would result in even lower start times, which, in turn, would help decrease the MTTR. Nonetheless, an improvement in the restart time was possible by modifying, in Kubernetes, the *pullPolicy* of the container images from Always to IfNotPresent. What this configuration does is that, instead of always pulling the container image from the container registry when the service is started, Kubernetes verifies if the image already exists locally. If it already exists, we can save the time required to download the image from the repository.

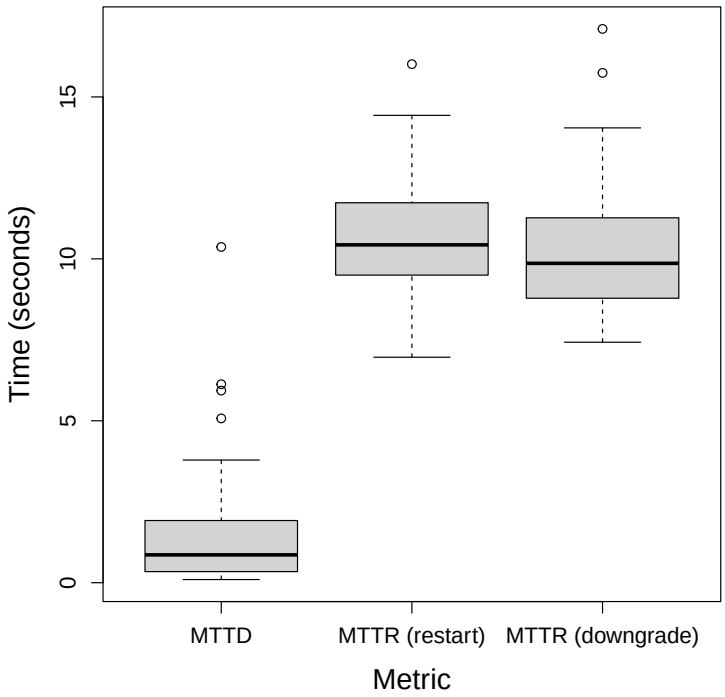


Figure 6.2: Performance in the wrong output scenario

### 6.1.2 Wrong output

Observing the results obtained in the wrong output scenario (Figure 6.2), we notice that the results are identical to the ones from the previous experiment. However, in this scenario, there is almost no difference between the performance of the recovery actions.

Additionally, the Mean Time To Detect presents even lower values, of 1.550 seconds, when compared to the previous experiment. Since all the experiments were performed under the same conditions, one may conclude that the number of spans considered per batch was the lower end of the batch (30 spans). Considering that the MTTD is mostly influenced by the batch time, this could explain the results observed. Another possible explanation lies in the fact that the fault could also manifest earlier when compared to the other scenarios.

### 6.1.3 Memory Leak

Contemplating the results for the wrong output scenario, a similar result from the one of the crash scenario (Figure 6.1) can be seen in Figure 6.2. This helps reinforce the idea that the mean time to repair does not differ much even when dealing with different services, written in different programming languages and with different image sizes.

On the other hand, the detection time is not influenced by the service, but by the time it takes for the required amount of spans to be produced and batched before being sent to the service in charge of fault detection.

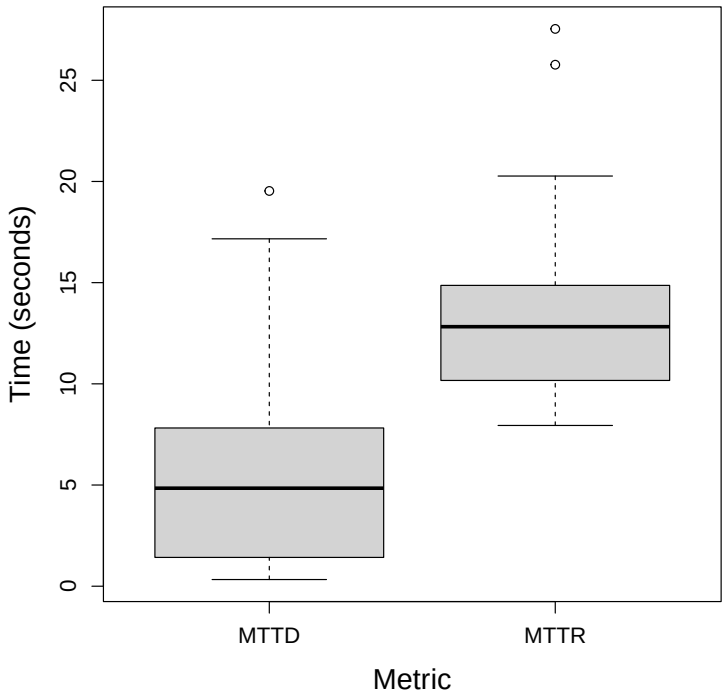


Figure 6.3: Performance in the memory leak scenario

### 6.1.4 Hang and corrupted output

Regarding the last two scenarios - hang and corrupted output - that were initially considered and prepared for evaluation, these ended up not providing the expected results that would be worth evaluating. The reason for discarding the **hang scenario** was due to a limitation in the detection phase. The hang did not manifest as an HTTP status code and the only way to detect it would be to consider the request duration contained in each span. Then, this value could be compared to an average request duration that could be stored in a database. Additionally, a threshold to be added on top of the average request duration could also be used to avoid any false positives. However, the time for developing and testing the solution is limited. Thus, this solution was discarded and the time was used to implement and evaluate the recovery strategies.

In which concerns the **corrupted output scenario**, it was discarded after the results obtained in the experiments. The way the fault was injected resulted in a cascading failure which incurred, once again, in a detection limitation of the present solution. The fault was injected in the cart service, which would return a corrupted output (wrong variable data type) upon confirming the cart information and proceeding to shipping. This corrupted value was persisted in a session variable and not used when defining the shipping information. Finally, when one would arrive to the checkout, which was handled by the payment service, this value was used and resulted in a failure. Despite the fact that the HTTP status code, which appeared in the trace data, would point out that the culprit would be the payment service, the true culprit was not present in the invocation chain, since the fault was injected in the cart. The way to discover the root cause of the present failure would not be so trivial. The reason behind this is that, for the best of my knowledge, it is not possible to correlate two individual traces. This makes it impossible

to find the true culprit.

In a final consideration regarding the performance of the proposed solution, one can observe that the MTTR in the different scenarios is similar. Only in the wrong output scenario a different value is observed, but still close to the times of the remaining scenarios. However, since the three scenarios are evaluated with different services, each written in a distinct programming language, and having different container sizes, it is recognizable that some differences may be observed. Since the container size only impacts the download time that would be required to fetch the image, but the pullPolicy is set to *IfNotPresent*, the image size can be discarded as the reason behind the different times observed. Thus, the main reason lies in the different programming languages and frameworks used for the services considered. The catalogue service is written in nodejs, shipping is written in Java with Springboot and ratings is written in PHP. In summary, we can conclude that the type of failure does not affect the mean time to recover of the proposed approach. In an additional consideration, the Mean Time to Detect in the wrong output scenario is considerably lower. However, this result is not related with the performance of the recovery action, but instead, with this specific failure scenario.

## 6.2 Effectiveness of the recovery actions

The performance experiments from the previous section show that the proposed solution can perform recovery actions in a short period of time. This is mainly due to the fast publish-subscribe middleware, the execution time of the framework and the usage of container-based virtualization, which allows low container startup times when compared to virtual machine virtualization. However, an analysis should be performed to uncover if these recovery actions are effective and able to recover the affected services, bringing them back to providing correct service.

To verify the effectiveness of the solution, a set of metrics that are usually considered when executing performance tests are used. This type of tests allow to verify the behavior of the system under load and uncover any possible limitations of the target application.

The metrics considered are the ones provided by the load generator *Locust*:

- **Throughput** - using 10 simultaneous clients, the number of requests that the system handles, per second, is measured.
- **Response time** - the time, in milliseconds, that it takes for the service to respond to requests is also considered. The data from this metric that is present in the plot corresponds to the 95th percentile. This percentile describes the response time required for 95% or less of the requests.
- **Number of failures per second** - the HTTP status codes from the requests performed are considered. This helps to know when an anomaly in the system might have occurred.

With these metrics we can investigate the impact of the different failures in the managed application. In order to get a baseline from the system behavior, a golden run was performed, which followed the guidelines provided in Chapter 4. The load generated using Locust was sent directly from the browser to simulate, in a more realistic manner, the client requests while using an e-commerce application.

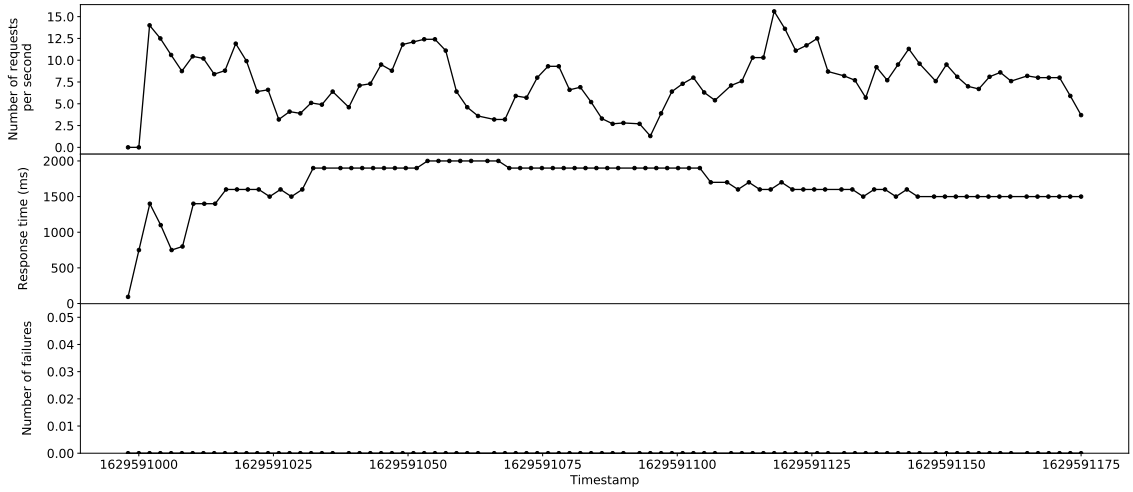


Figure 6.4: Golden run

The results from the golden run (Figure 6.4) do not exhibit any failure as expected. We can observe that the maximum number of requests that the system handles, per second, tops at around 15. And finally, the 95th percentile of response times is 2 seconds, which means that 95% of the requests are answered in 2 or less seconds. Using the 95th percentile has the advantage of considering the vast majority of the results while still discarding some outliers. The 75th percentile could also be a viable option, but the 95th percentile was more appropriate since it considers more data.

### 6.2.1 Crash

In which concerns the crash scenario, the first recovery action evaluated was the restart action (Figure 6.5). When comparing with the baseline run (Figure 6.4), it is possible to observe that the number of requests to which the system responds decreases to one third from the timestamp where the failure is injected until the end of the experiment. Moreover, after applying the restart, the response time starts to decrease considerably. This does not mean that the system was able to recover from the failure. Instead, it shows that the restart action impacts the response time and that, after the actions is applied, the response time decreases again. Since the fault is injected at compile-time and persists throughout the experiment, it was expected that this action would not be effective in this specific scenario.

Regarding the version downgrade action, a different behavior is observed (Figure 6.6). The number of requests that the system can handle is similar to the ones from the golden run. The response times also decrease slightly after the recovery action is applied to the system. Furthermore, in the third subplot, it is noticeable that, shortly after the action is performed, the number of requests with HTTP codes goes down to zero. Upon the results obtained, one may conclude that the version downgrade action is effective in this scenario.

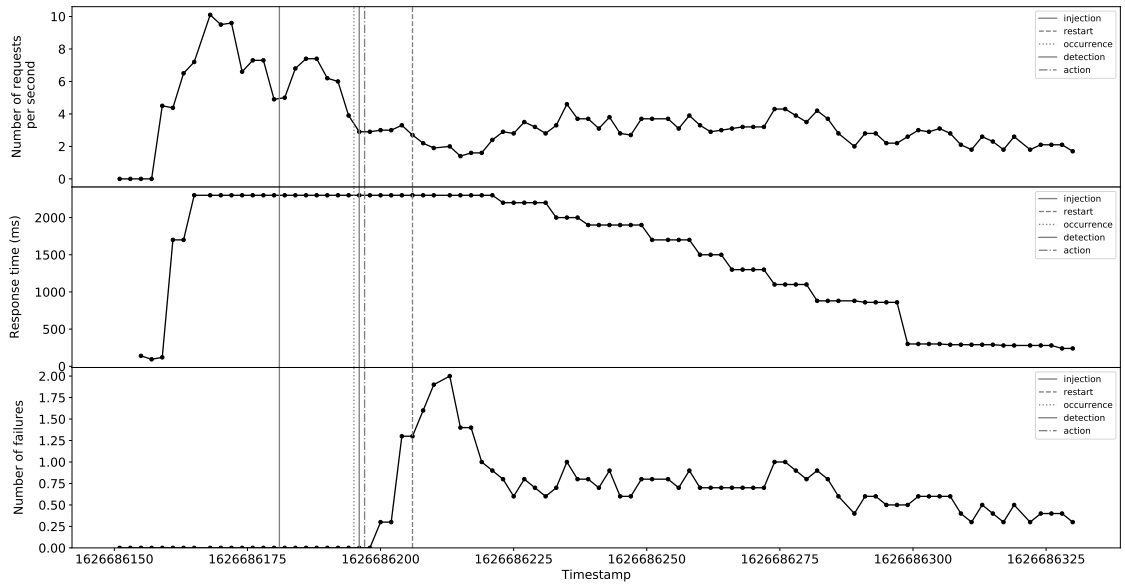


Figure 6.5: Crash catalogue scenario with restart

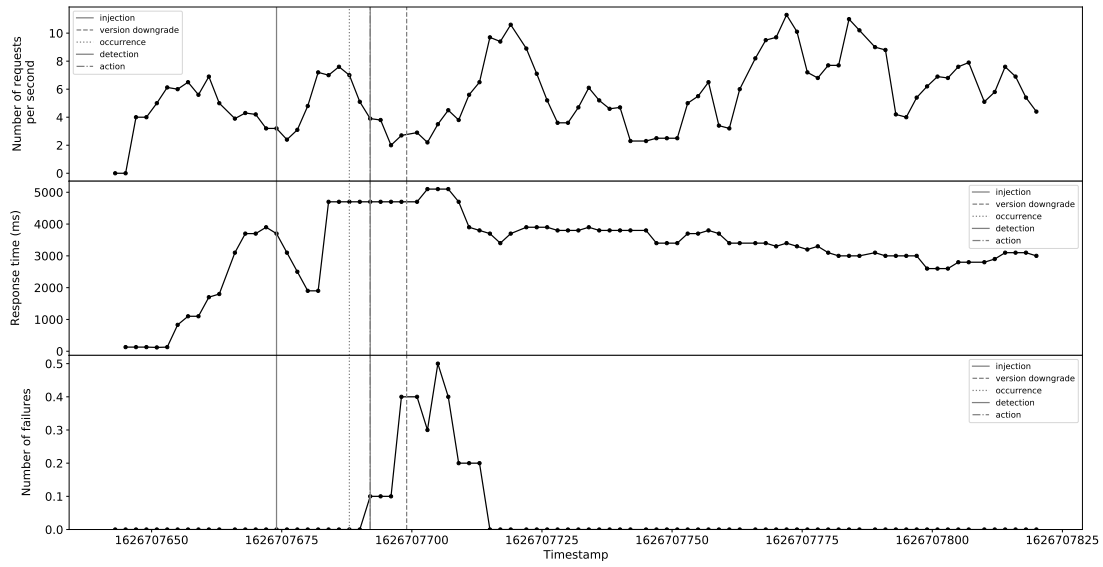


Figure 6.6: Crash catalogue scenario with version downgrade

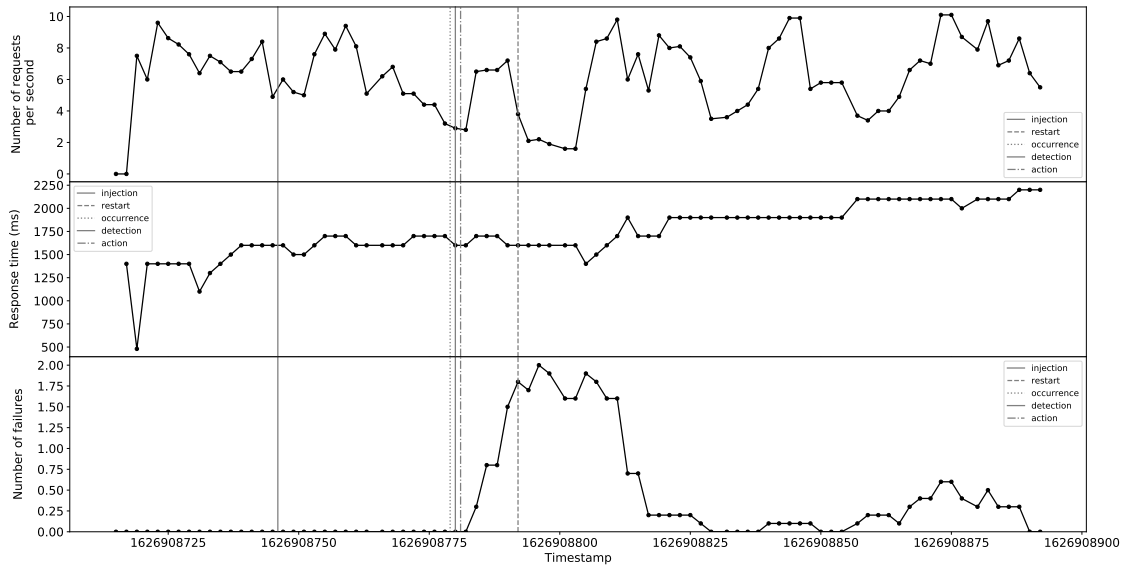


Figure 6.7: Wrong output scenario with restart

### 6.2.2 Wrong output

This scenario presents similar results to the ones from the crash catalogue scenario. Starting with the restart action (Figure 6.7), it presents that the number of requests does not decrease as much as in the crash scenario. This may be a result from a crash being a more severe scenario, which has a more meaningful impact in the service provided by the application. On the other hand, the type of interface error analysed in this scenario shows a lighter impact. Nonetheless, the two remaining subplots in Figure 6.7 present similar scenarios to the ones from the previous scenario, which are evidences that the recovery action was not successful. Since this fault is also permanent and lasts throughout the experiment, the version downgrade once again demonstrates the ability to recover the system from the failure scenario (Figure 6.8). It is possible to observe that after applying the recovery action, the metric values return back to the normal values expected, which match with the ones from the baseline run.

### 6.2.3 Memory leak

In this final scenario, the restart action is the only action considered. This is done since this is a transient fault, *i.e.*, it is bounded in time, and it would not be appropriate to try to downgrade the service version. With this scenario, the core idea was to evaluate if the restart action could improve a mechanism already used by Kubernetes, which is the OOMKiller.

As stated in Kubernetes documentation [57], a container can have as much memory as it requests, as long as the memory usage does not exceed the limit defined. Whenever a container tries to allocate more memory than its limit, it becomes a candidate for termination. If it keeps consuming memory past its limit, the *kubelet* will terminate the pod. The kubelet consists in an agent that runs on each node and assures that containers are running in a pod [58]. To terminate pods, the kubelet resorts to the "OOM Killer". The OOM Killer works by observing the node memory usage to search for memory exhaustion. If such exhaustion is detected, the OOM Killer will choose which processes should

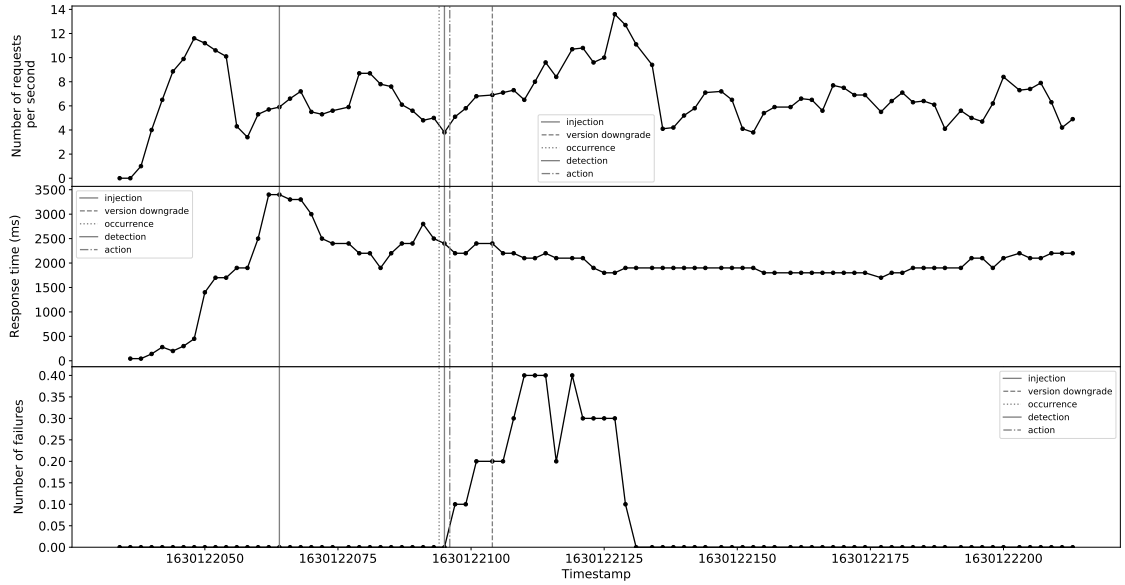


Figure 6.8: Wrong output scenario with version downgrade

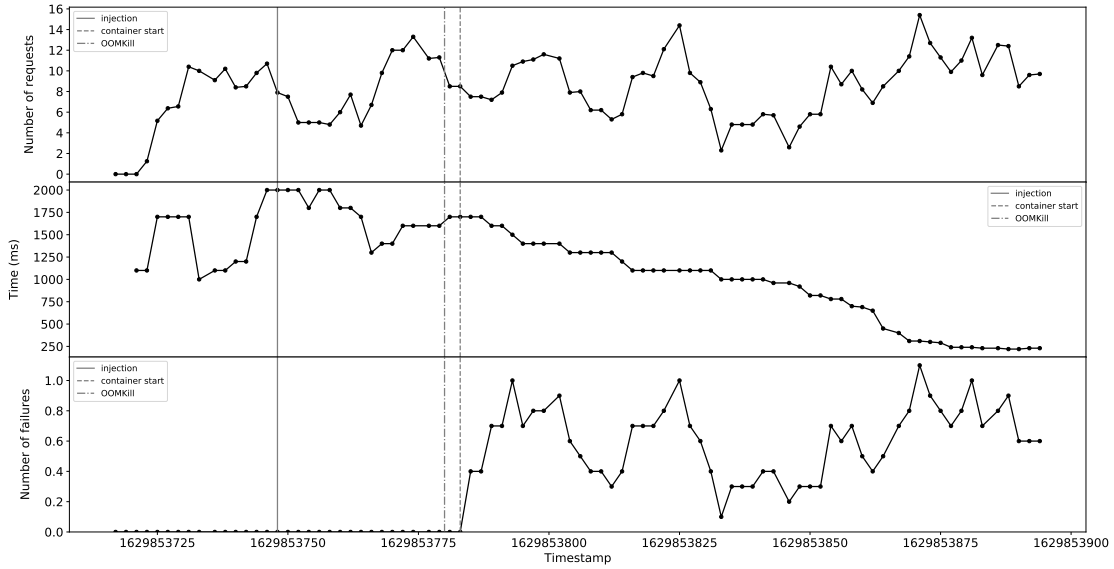


Figure 6.9: Behavior of Kubernetes OOMKiller

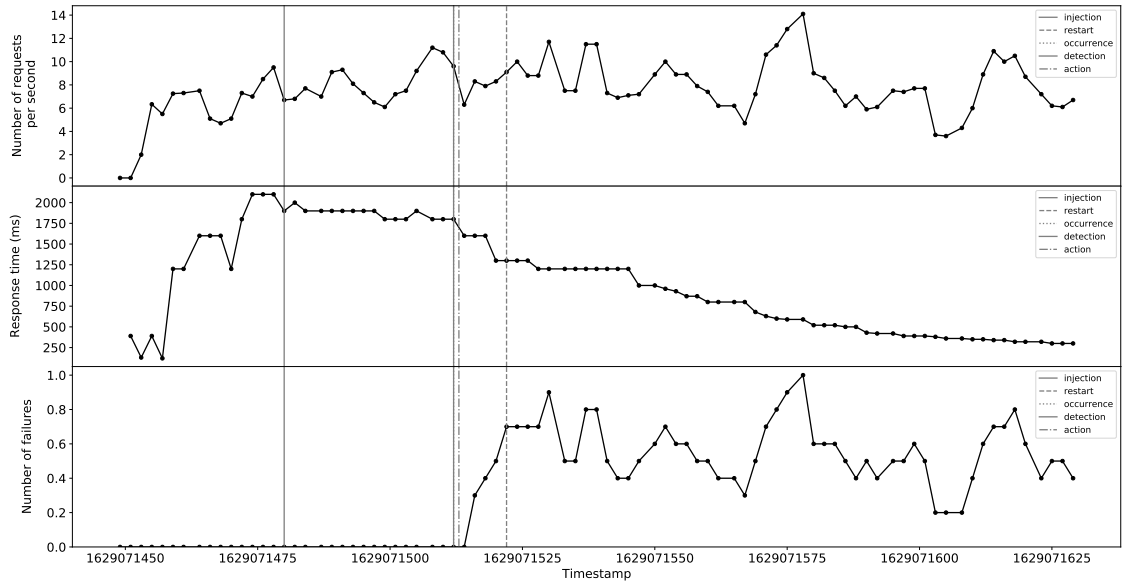


Figure 6.10: Memory Leak scenario with restart

be killed. The core objective is to kill the least amount of processes while recovering the maximum amount of the node's memory. For this, the kernel holds an oom_score for each process and the process with higher oom_score is more likely to be killed. The oom_score is calculated with regard to the percentage of memory that is being used by the process.

When performing the chaos experiment, we can observe, in Figure 6.9, that Kubernetes uses the OOM Killer to terminate the service which is exhausting the memory resources. Comparing the results from the OOM Killer with the ones from the restart action (Figure 6.10), we can observe that the results obtained are very similar from one another in every aspect. This result is not a complete surprise since both approaches terminate the service and start it again at a clean state. However, the difference lies in the detection method. While the OOM Killer relies on the memory usage to actuate, the present solution monitors the spans provided by distributed tracing. And, as observed throughout the experiments, the service provided started to become affected and unable to answer some requests before OOM Killer was triggered. Thus, the approach used by the framework can complement the mechanism already provided by Kubernetes, in order to provide error recovery sooner. Nonetheless, the recommended approach would be to have the Horizontal Pod Autoscaler enabled, which would scale pods according to resource usage and avoid any failures due to the increase in resource consumption.

## 6.3 Effectiveness of the recovery strategies

The performance and effectiveness of isolated recovery actions was already evaluated. However, an interesting approach would be to also consider recovery strategies. Such strategy encompasses a more elaborated workflow, composed of more than one recovery actions, which would provide some sort of simple automation to bring additional benefits for the system's dependability. Hence, the recovery strategies mentioned in Section 3.5 - **global restart** and **iterative recovery** - will be evaluated. Due to the conclusions obtained in the previous section, the global restart strategy will only be evaluated in the memory leak scenario. The same applies to the iterative recovery strategy, which will only

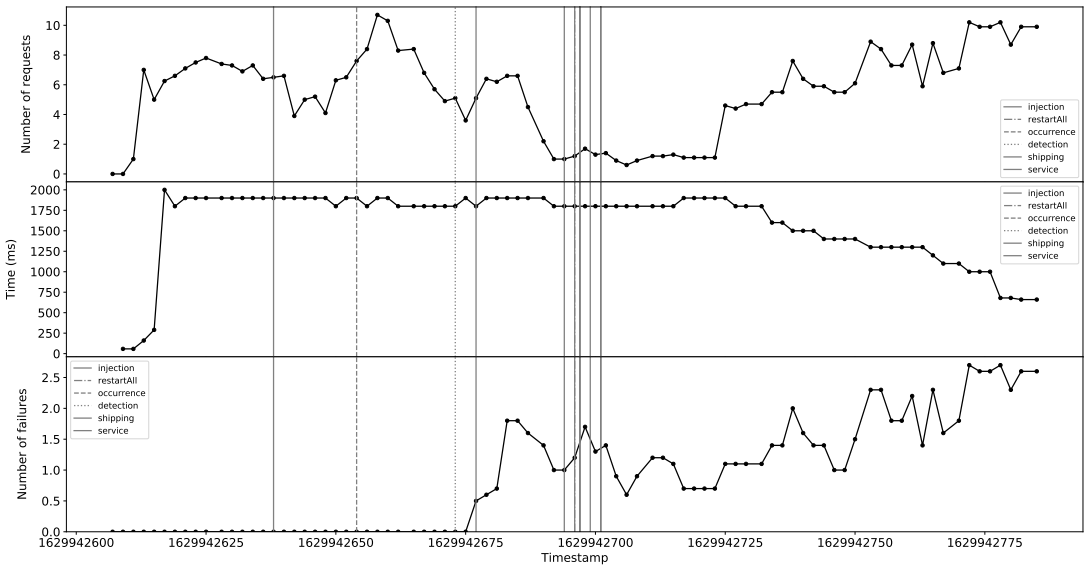


Figure 6.11: Memory Leak scenario with global restart strategy

be applied to the crash catalogue scenario due to the similar conclusions between the crash catalogue and memory leak scenarios.

Figure 6.11 displays the results from the global restart strategy. The leftmost vertical line is always the injection timestamp. Additionally, the lines at the line all have the label "service" since represent the start time of each service and keeps the legend more clean. When comparing the plot with the one from the single recovery action (Figure 6.10), it is possible to observe similarities in the response time and failure subplots. However, there is a noticeable and important observation in the first subplot in Figure 6.11. This exhibits that the requests per second are severely affected since all the services are restarted, which can briefly affect the application's availability. Nonetheless, this strategy may be applied when no fault detection and analysis mechanism is in place, to provide the culprits of an existing failure, and the application can sustain momentary downtime. Notwithstanding, in a system's normal operation, the single restart option is preferable due to the lower actuation risk associated.

Regarding the *iterative recovery* strategy, it combines both recovery actions (restart and version downgrade) to further improve the recovery phase. Despite knowing that the *Mean Time Between Failures (MTBF)* of every system is different and there is no reference value, it is recognizable that if a failure with the same characteristhics arises within a short amount of time, such as a day or less, we can conclude that we are dealing with the same failure and that it was not repaired. Thus, it is possible to define a certain period, which could be equal or lower than the MTBF, and that would be the timeframe considered. If failure data containing the same HTTP status code, the same components and within the defined timeframe was found, one could consider that the previous recovery action was not effective and a new one should be applied. For the sake of the current experiment, the timeframe considered was 120 seconds, which is the duration of the experiment. Furthermore, upon applying the first recovery action, a standby time of 60 seconds was considered, to help present the results in the plot and their subsequent analysis.

When observing the experimental results (Figure 6.12), it is noticeable that the recovery strategy was effective. After applying the first recovery action (a restart), this was not effective and, once information regarding the same failure appeared, the version downgrade

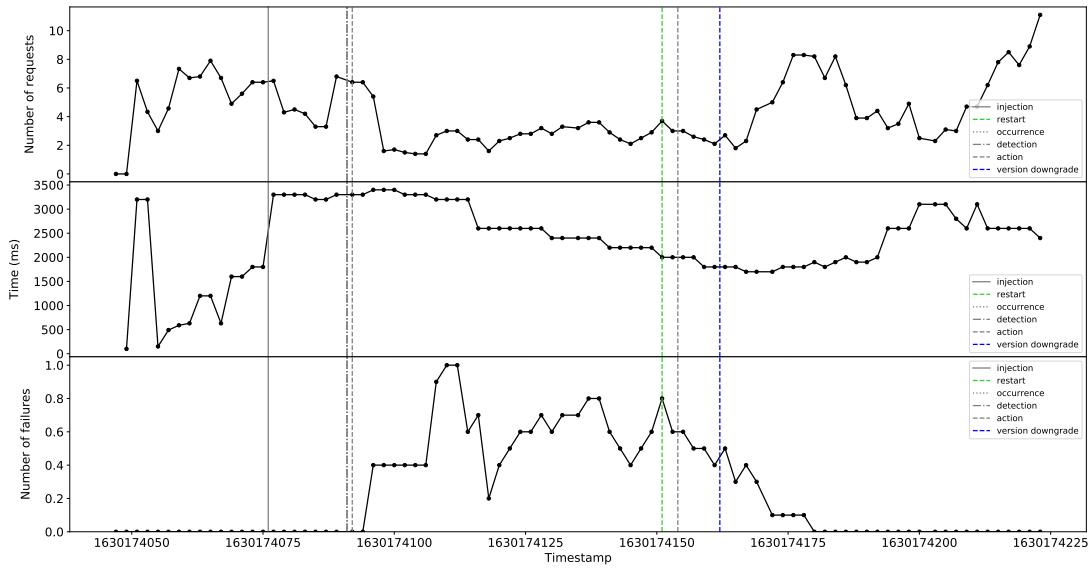


Figure 6.12: Crash catalogue scenario with iterative recovery strategy

was applied and restore the system's normal operation.

## 6.4 Answers to the research questions

The present section provides the answers to the research questions defined in Chapter 1. It is split in three subsections, each one corresponding to one of the research questions.

**RQ1. Study the feasibility of building a MAPE-K loop using a Publish-Subscribe middleware.**

Through the experiments performed, it was possible to observe that *Apache Kafka* fulfilled the desired requirements and enabled the feedback loop. It helped guarantee exactly-once delivery semantic of messages. This has a core importance in guaranteeing that the recovery actions are only applied once, even if a failure occurs in Kafka, Zookeeper or any of the components of the architecture. The durability provided by the topics, which can store information until it is consumed, helps guarantee that no data is lost. Moreover, it enabled fast information exchange among the different components. This high-performance from Kafka is very important to help guarantee the fast recovery times that the proposed solution provides.

**RQ2. Evaluate the possibility of achieving an availability of 99.99% for a cloud-native application with the self-healing capabilities provided by our framework.**

Availability is one of the most important quality attributes for companies nowadays. The underlying idea for the current research question was to evaluate how the proposed solution could be a framework to assure high availability of cloud native application. Since the Mean Time Between Failures (MTBF) is something that we cannot control and differs from system to system, what we can do is evaluate, taking into consideration the MTTR obtained, what MTBF would allow us to achieve the desired availability.

Analysing the results displayed in Table 6.2, it is possible to observe that, with the

| Failure scenario | Recovery action | MTTR (s) | Availability (%) | MTBF (h) |
|---|---|---|---|---|
| Crash catalogue | Restart | 12.150 | 99.9% | 3.371 |
| | | | 99.99% | 33.747 |
| | Version downgrade | 15.890 | 99.9% | 4.409 |
| | | | 99.99% | 44.134 |
| Wrong output | Restart | 11.500 | 99.9% | 3.191 |
| | | | 99.99% | 31.941 |
| | Version downgrade | 9.358 | 99.9% | 2.597 |
| | | | 99.99% | 25.992 |
| Memory leak | Restart | 14.083 | 99.9% | 3.908 |
| | | | 99.99% | 39.116 |

Table 6.2: Mean Time Between Failures (MTBF)

lowest MTTR, which belongs to the version downgrade in the wrong output scenario, it is possible to withstand almost one failure per day while still guaranteeing high availability. In which concerns the worst case scenario, related to the version downgrade in the crash catalogue, the system can handle a failure every two days.

As previously stated, there is no reference value to be used as the Mean Time Between Failures (MTBF), since every system is different, and as a consequence, it behaves and fails differently. However, the recovery times provided by the framework enable the system to accommodate failures occurring within a short period of time between each other, while still guaranteeing high availability.

Notwithstanding, to further improve the availability of an application, additional mechanisms and considerations should take place. A common example is the existence of spare replicas that must coexist with the running instances in order to provide redundancy in case of failure.

**RQ3. Evaluate the effectiveness of different recovery strategies as a means to complement single recovery actions.**

Regarding the recovery strategies, these can be seen as a means to improve single recovery actions. With the experiments performed and documented in Section 6.3, one could notice that even with simple recovery strategies, it is possible to recover the system's correct service.

The **global restart** action takes advantage of the fast restart time to try to mitigate any existing anomaly. This is a non-intrusive approach when compared to the version downgrade one. However, it must be carefully applied since the throughput and response time of the application may be affected. An additional downside is that this strategy does not scale. If we are working with a small microservices application, it could be feasible to apply this recovery strategy. However, the same does not apply to a medium to large-scale application.

On the other hand, the **iterative recovery** is effective against permanent failures. Since these are common failures, which can result from software updates that introduce bugs in the production environment, it is positive to observe this outcome for the present recovery strategy.

Notwithstanding, the risks of actuation must be carefully studied, both for the recovery actions and strategies, to avoid damaging the service when trying to mitigate failures.

# Chapter 7

# Conclusion and Future Work

This final Chapter presents an overview of the work performed, the conclusions attained and possible research directions. The overall work of this thesis resulted in an architecture of a feedback loop which is capable of autonomically apply recovery actions and strategies to a microservices application. Both goals initially defined in Subsection 1.3 have been fulfilled.

The first goal, regarding the development of a framework that could apply recovery actions to a microservices application was achieved. The solution monitors a cloud-native application using data from distributed tracing. This data is then analysed to search for any anomaly in the system and different recovery actions or strategies can be applied to help mitigate the existing failure. The second goal was also met as shown in Chapter 6. Applying the failures defined in Chapter 4 to the framework helped analyse the performance and effectiveness of the recovery actions and strategies considered. The restart action provides a non-intrusive and fast method to actuate in the system. Taking advantage of this commonly used recovery action, it was possible to observe that, in a failure scenario which was the result of a transient fault, it can help restore the normal system service. However, one may conclude that scaling up the resources would be the appropriate action in this scenario. On the other hand, the version downgrade action proves to be effective in scenarios where the fault injected is permanent. Since permanent failures, such as the one's resulting from software updates, are not unusual, it is interesting to observe the fast performance and proven effectiveness of this action. Moreover, the usage of recovery strategies to enhance the recovery phase and provide automation capabilities when initial recovery actions are not effective also revealed that this is a path that can be followed and further enhanced. The global restart action has the advantage of helping the system when no detection method is available. However, there are risks attached, such as affecting the application's throughput and the fact that it can not be used in application composed of many microservices. Regarding the iterative recovery strategy, it exhibited the capability of a simple automation workflow of choosing a different recovery action when the current one was not effective.

Reflecting about the work performed, the following conclusions can be withdrawn:

1. It is possible to actuate autonomically and perform recovery actions and strategies in a microservices application to recover the system in failure scenarios.
2. The complex and dynamic nature of microservice architectures makes it harder to detect and actuate upon failures.
3. There is still a lack of research in this field that should be addressed.

In which concerns the first conclusion, the proposed framework enables the actuation of recovery actions and strategies in cloud-native applications. It has the capability of working with any microservices application and container orchestrator due to the existing decoupling. Since the framework gathers system metrics from external sources, such as distributed tracing, it can work with any system as long as it complies with the message format defined. Additionally, the application may work with different orchestrators, but the commands that are applied would need to be modified since each orchestrator may have its own commands.

Regarding the second point, the experiments performed, namely the one where the output provided by the application is corrupted, show scenarios that make it difficult to detect the existing failure, which, in turn, makes it impossible to actuate in that concrete failure scenario.

The third and final point reports something that was observed while performing literature review. The trend towards adopting cloud computing and developing cloud-native applications is something new, which justifies the lack of research in fields such as recovering from failures in microservices, detecting faults in microservices and the different types of faults that may occur in this kind of environments.

From the work performed, the following research directions may be followed:

- Develop and integrate, in the current framework, a mechanism capable of choosing the most appropriate recovery action.
- Integrate a proper fault detection mechanism in the architecture to improve the detection of the culprits.
- Assess the risk of autonomic actuation in a microservices application.
- Develop and evaluate more complex recovery strategies.
- Consider additional metrics to evaluate the failure scenarios at runtime.

Regarding the first consideration, in the work performed, only one recovery action was considered per run. However, an interesting approach would be to develop a solution to choose the most appropriate action to apply in a concrete failure scenario. To enable this mechanism, it would be utterly important to improve the detection mechanism. This should provide as much information as possible about the failure to the component that chooses the recovery actions, so that the latter could analyse and choose the most appropriate..

The second item is related to the fact that, in the current implementation, the detection method considers as culprits all the services that are part of a trace which an HTTP status code in the 400 and 500 range. This approach should be replaced with one where a proper fault detection mechanism analyses metrics and data collected from the system to assess the root cause of a failure. This would also improve the work performed by the proposed solution since it could help reduce the risk of actuation.

In third place, the risk of actuating in a system should be carefully analysed. Since the availability is the core quality attribute that should be respected, one should be careful to avoid damaging the system even more when trying to restore the system's service. An additional consideration would be to evaluate and compare the impact of recovery actions in stateless and stateful services.

The fourth point is related to study and implement additional recovery strategies that could be embedded in the proposed solution. These could benefit from improved detection mechanisms to help provide richer information upon which the system could deliberate when choosing a recovery action.

In fifth and final place, more metrics from the system, besides distributed tracing, should be considered to assist the framework in having a better overview of the managed application. This, in turn, could improve the detection and planning phases.

Reflecting upon the work performed, it is possible to conclude that it was successful. The goals defined were accomplished and relevant conclusions and considerations could be withdrawn. Finally, the acceptance of the research paper that was submitted to the *DREAMS Workshop 2021* helped enhance the importance and relevance of the work performed.

# Bibliography


[1] *500 internal server error · Issue #2 · paulc4/microservices-demo*. URL: https://github.com/paulc4/microservices-demo/issues/2 (visited on 5th Dec. 2020).

[2] Fatemeh Afsharnia. 'Failure Rate Analysis'. In: Dec. 2017. ISBN: 978-953-51-3713-9. DOI: 10.5772/intechopen.71849.

[3] Khaled Alnawasreh et al. 'Online robustness testing of distributed embedded systems: An industrial approach'. In: 2017. DOI: 10.1109/ICSE-SEIP.2017.17.

[4] Apache. *Apache Kafka*. URL: https://kafka.apache.org/ (visited on 10th Dec. 2020).

[5] Apache. *Apache Mesos*. URL: http://mesos.apache.org/ (visited on 2nd Jan. 2021).

[6] Apigee. *502 Bad Gateway Unexpected EOF | Apigee Docs*. URL: https://docs.apigee.com/api-platform/troubleshoot/runtime/502-bad-gateway (visited on 2nd Jan. 2021).

[7] Atlassian. *SLA vs. SLO vs. SLI - Differences | Atlassian*. URL: https://www.atlassian.com/incident-management/kpis/sla-vs-slo-vs-sli (visited on 12th July 2021).

[8] Algirdas Avižienis et al. 'Basic concepts and taxonomy of dependable and secure computing'. In: *IEEE Transactions on Dependable and Secure Computing* 1 (1 2004). ISSN: 15455971. DOI: 10.1109/TDSC.2004.2.

[9] Nagwa Lotfy Badr. 'An investigation into autonomic middleware control services to support distributed self-adaptive software'. PhD thesis. Liverpool John Moores University, 2003.

[10] Baeldung. *Exactly Once Processing in Kafka with Java*. URL: https://www.baeldung.com/kafka-exactly-once (visited on 15th Aug. 2021).

[11] A Basiri et al. 'Chaos Engineering'. In: *IEEE Software* 33 (3 2016), pp. 35–41. DOI: 10.1109/MS.2016.60.

[12] Len Bass, Paul Clements and Rick Kazman. *Software Architecture in Practice*. Third. Addison-Wesley Professional, 2012.

[13] Alfredo Benso and Stefano Di Carlo. 'The Art of Fault Injection'. In: *Control Engineering and Applied Informatics* 13 (4 2011). ISSN: 14548658.

[14] Thomas Brand and Holger Giese. 'Towards software architecture runtime models for continuous adaptive monitoring'. In: vol. 2245. 2018.

[15] Simon Brown. *The C4 model for visualising software architecture*. URL: https://c4model.com/ (visited on 15th Dec. 2020).

[16] Thanh Bui. 'Analysis of Docker Security'. In: *ArXiv* abs/1501.02967 (2015).

[17] Emiliano Casalicchio. 'A study on performance measures for auto-scaling CPU-intensive containerized applications'. In: *Cluster Computing* 22 (3 2019). ISSN: 15737543. DOI: 10.1007/s10586-018-02890-1.

[18] Frederico Cerveira et al. 'Evaluation of restful frameworks under soft errors'. In: vol. 2020-October. 2020. DOI: 10.1109/ISSRE5003.2020.00042.

[19] Marcello Cinque, Raffaele Della Corte and Antonio Pecchia. 'Microservices Monitoring with Event Logs and Black Box Execution Tracing'. In: *IEEE Transactions on Services Computing* (2019). ISSN: 19391374. DOI: 10.1109/TSC.2019.2940009.

[20] Mariana Cunha and Nuno Laranjeiro. 'Assessing Containerized REST Services Performance in the Presence of Operator Faults'. In: 2018. DOI: 10.1109/EDCC.2018.00025.

[21] Piotr Karwatka - Divante. *Monolithic architecture vs microservices: Which is better? | Divante*. URL: https://divante.com/blog/monolithic-architecture-vs-microservices/ (visited on 4th Jan. 2021).

[22] Docker. *Swarm mode overview | Docker Documentation*. URL: https://docs.docker.com/engine/swarm/ (visited on 3rd Jan. 2021).

[23] MDN Web Docs. *HTTP response status codes*. URL: https://developer.mozilla.org/pt-PT/docs/Web/HTTP/Status (visited on 5th Jan. 2021).

[24] Abhishek Dubey et al. 'Towards a verifiable real-time, autonomic, fault mitigation framework for large scale real-time systems'. In: *Innovations in Systems and Software Engineering* 3 (1 2007). ISSN: 16145046. DOI: 10.1007/s11334-006-0015-7.

[25] João A. Durães and Henrique S. Madeira. 'Emulation of software faults: A field data study and a practical approach'. In: *IEEE Transactions on Software Engineering* 32 (11 2006). ISSN: 00985589. DOI: 10.1109/TSE.2006.113.

[26] DZone. *Interpreting Kafka's Exactly-Once Semantics*. URL: https://dzone.com/articles/interpreting-kafkas-exactly-once-semantics (visited on 15th Aug. 2021).

[27] Elasticsearch. *Elasticsearch: The Official Distributed Search & Analytics Engine*. URL: https://www.elastic.co/elasticsearch (visited on 29th July 2021).

[28] Luca Florio and Elisabetta Di Nitto. 'Gru: An approach to introduce decentralized autonomic behavior in microservices architectures'. In: 2016. DOI: 10.1109/ICAC.2016.25.

[29] CNCF (Cloud Native Computing Foundation). *CNCF Cloud Native Definition*. URL: https://github.com/cncf/toc/blob/master/DEFINITION.md#%E4%B8%AD%E6%96%87%E7%89%88%E6%9C%AC (visited on 10th Jan. 2020).

[30] Martin Fowler. *Microservices*. URL: https://martinfowler.com/articles/microservices.html (visited on 10th Dec. 2020).

[31] Peter Garraghan et al. 'Emergent Failures: Rethinking Cloud Reliability at Scale'. In: *IEEE Cloud Computing* 5 (Sept. 2018), pp. 12–21. DOI: 10.1109/MCC.2018.053711662.

[32] Martin Garriga. 'Towards a taxonomy of microservices architectures'. In: vol. 10729 LNCS. 2018. DOI: 10.1007/978-3-319-74781-1_15.

[33] Gartner. *Definition of Scalability - Gartner Information Technology Glossary.* URL: https://www.gartner.com/en/information-technology/glossary/scalability (visited on 1st Dec. 2020).

[34] Gremlin. *Chaos Engineering tools comparison.* URL: https://www.gremlin.com/community/tutorials/chaos-engineering-tools-comparison/.

[35] IBM Group. 'Autonomic Computing White Paper: An Architectural Blueprint for Autonomic Computing'. In: *IBM White Paper* (June 2005). ISSN: 1944-8244.

[36] Haryadi S. Gunawi et al. 'Why does the cloud stop computing? Lessons from hundreds of service outages'. In: 2016. DOI: 10.1145/2987550.2987583.

[37] Red Hat. *Understanding cloud-native apps.* URL: https://www.redhat.com/en/topics/cloud-native-apps (visited on 15th Dec. 2020).

[38] Robert Heinrich et al. 'Performance engineering for microservices: Research challenges & directions'. In: 2017. DOI: 10.1145/3053600.3053653.

[39] Victor Heorhiadi et al. 'Gremlin: Systematic Resilience Testing of Microservices'. In: vol. 2016-August. 2016. DOI: 10.1109/ICDCS.2016.11.

[40] P. Horn. 'Autonomic Computing: IBM's Perspective on the State of Information Technology'. In: 2001.

[41] Gang Huang, Xuanzhe Liu and Hong Mei. 'SOAR: Towards dependable Service-Oriented Architecture via reflective middleware'. In: *International Journal of Simulation and Process Modelling* 3 (1-2 2007). ISSN: 17402131. DOI: 10.1504/IJSPM.2007.014715.

[42] Peng Huang et al. 'Gray Failure: The Achilles' Heel of Cloud-Scale Systems'. In: vol. Part F129307. 2017. DOI: 10.1145/3102980.3103005.

[43] Markus C. Huebscher and Julie A. McCann. 'A survey of Autonomic Computing - Degrees, models, and applications'. In: *ACM Computing Surveys* 40 (3 2008). ISSN: 03600300. DOI: 10.1145/1380584.1380585.

[44] IBM. 'An architectural blueprint for autonomic computing'. In: *IBM White Paper* 36 (June 2006). ISSN: 19448244.

[45] Didac Gil De La Iglesia and Danny Weyns. 'MAPE-K formal templates to rigorously design behaviors for self-adaptive systems'. In: *ACM Transactions on Autonomous and Adaptive Systems* 10 (3 2015). ISSN: 15564703. DOI: 10.1145/2724719.

[46] Instana. *Robot Shop: Sample Microservices Application.* URL: https://github.com/instana/robot-shop (visited on 4th Feb. 2021).

[47] Istio. *Istio / Distributed Tracing FAQ.* URL: https://istio.io/latest/about/faq/distributed-tracing/ (visited on 20th June 2021).

[48] Jaeger. *Architecture - Jaeger documentation.* URL: https://www.jaegertracing.io/docs/1.21/architecture/ (visited on 28th May 2021).

[49] *Jaeger: open source, end-to-end distributed tracing.* URL: https://www.jaegertracing.io/.

[50] Lalita J. Jagadeesan and Veena B. Mendiratta. 'When Failure is (Not) an Option: Reliability Models for Microservices Architectures'. In: 2020. DOI: 10.1109/ISSREW51248.2020.00031.

[51] Hugo Jernberg, Per Runeson and Emelie Engström. 'Getting started with chaos engineering - Design of an implementation framework in practice'. In: 2020. DOI: 10.1145/3382494.3421464.

[52] Andréas Johansson et al. 'On enhancing the robustness of commercial operating systems'. In: vol. 3335. 2005. DOI: 10.1007/978-3-540-30225-4_11.

[53] *JSON*. URL: https://www.json.org/json-en.html (visited on 10th Nov. 2020).

[54] Hui Kang, Haifeng Chen and Guofei Jiang. 'PeerWatch: A fault detection and diagnosis tool for virtualized consolidation systems'. In: 2010. DOI: 10.1145/1809049.1809070.

[55] Jeffrey O Kephart and David M Chess. 'The Vision of Autonomic Computing'. In: *Computer* 36 (1 Jan. 2003), pp. 41–50. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1160055. URL: https://doi.org/10.1109/MC.2003.1160055.

[56] *Kubernetes*. URL: https://kubernetes.io/pt/ (visited on 23rd Nov. 2020).

[57] Kubernetes. *Assign Memory Resources to Containers and Pods*. URL: https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource/ (visited on 20th Aug. 2021).

[58] *Kubernetes Components | Kubernetes*. URL: https://kubernetes.io/docs/concepts/overview/components/ (visited on 10th Dec. 2020).

[59] Lorenzo De Lauretis. 'From monolithic architecture to microservices architecture'. In: 2019. DOI: 10.1109/ISSREW.2019.00050.

[60] R. K. Lenka, S. Padhi and K. M. Nayak. 'Fault Injection Techniques - A Brief Review'. In: *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*. 2018, pp. 832–837. DOI: 10.1109/ICACCCN.2018.8748585.

[61] Lightstep. *Understand Distributed Tracing | Lightstep Learning Portal*. URL: https://docs.lightstep.com/docs/understand-distributed-tracing#attributestags (visited on 28th June 2021).

[62] Litmus. *Architecture Summary | Litmus Docs*. URL: https://docs.litmuschaos.io/docs/architecture/architecture-summary (visited on 10th Aug. 2021).

[63] Haopeng Liu et al. 'What bugs cause production cloud incidents?' In: 2019. DOI: 10.1145/3317550.3321438.

[64] Vince Molnár and István Majzik. 'Model checking-based Software-FMEA: Assessment of fault tolerance and error detection mechanisms'. In: *Periodica polytechnica Electrical engineering and computer science* 61 (2 2017). ISSN: 20645279. DOI: 10.3311/PPee.9755.

[65] Fabrizio Montesi and Janine Weber. *Circuit Breakers, Discovery, and API Gateways in Microservices*. 2016.

[66] Roberto Natella, Domenico Cotroneo and Henrique S. Madeira. 'Assessing dependability with software fault injection: A survey'. In: *ACM Computing Surveys* 48 (3 2016). ISSN: 15577341. DOI: 10.1145/2841425.

[67] *Netflix/chaosmonkey: Chaos Monkey is a resiliency tool that helps applications tolerate random instance failures*. URL: https://github.com/netflix/chaosmonkey (visited on 10th Dec. 2020).

[68] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. 1st. O'Reilly Media, Feb. 2015, p. 280. ISBN: 978-1491950357.

[69] NTP. *How does it work?* URL: http://www.ntp.org/ntpfaq/NTP-s-algo.htm#Q-ALGO-BASIC-SYNC (visited on 20th Aug. 2021).

[70] OpenTelemetry. *What is OpenTelemetry? | OpenTelemetry.* URL: https://opentelemetry.io/docs/concepts/what-is-opentelemetry/ (visited on 20th Dec. 2020).

[71] OpenTracing. *Spans.* URL: https://opentracing.io/docs/overview/spans/ (visited on 22nd Dec. 2020).

[72] *OpenTracing specification.* URL: https://opentracing.io/specification/ (visited on 10th May 2021).

[73] *OpenZipkin · A distributed tracing system.* URL: https://zipkin.io/.

[74] René Peinl, Florian Holzschuher and Florian Pfitzer. 'Docker Cluster Management for the Cloud - Survey Results and Own Solution'. In: *Journal of Grid Computing* 14 (2 2016). ISSN: 15729184. DOI: 10.1007/s10723-016-9366-y.

[75] E. Grishikashvili Pereira and R. Pereira. 'Simulation of fault monitoring and detection of distributed services'. In: *Simulation Modelling Practice and Theory* 15 (4 2007). ISSN: 1569190X. DOI: 10.1016/j.simpat.2006.11.012.

[76] PingCAP. *Chaos Mesh - Your Chaos Engineering Solution for System Resiliency on Kubernetes | PingCAP.* URL: https://pingcap.com/blog/chaos-mesh-your-chaos-engineering-solution-for-system-resiliency-on-kubernetes.

[77] *Pods | Kubernetes.* URL: https://kubernetes.io/docs/concepts/workloads/pods/ (visited on 10th Dec. 2020).

[78] *Principles of chaos engineering.* URL: https://principlesofchaos.org/ (visited on 4th Feb. 2021).

[79] Rakesh Rana et al. 'Improving fault injection in automotive model based development using fault bypass modeling'. In: *INFORMATIK 2013 – Informatik angepasst an Mensch, Organisation und Umwelt.* Ed. by Matthias Horbach. Bonn: Gesellschaft für Informatik e.V., 2013, pp. 2577–2591.

[80] Daniel Richter et al. 'Highly-Available Applications on Unreliable Infrastructure: Microservice Architectures in Practice'. In: 2017. DOI: 10.1109/QRS-C.2017.28.

[81] Mallanna S.D. and Devika M. 'Distributed Request Tracing using Zipkin and Spring Boot Sleuth'. In: *International Journal of Computer Applications* 175 (12 2020). DOI: 10.5120/ijca2020920617.

[82] Matheus Santana et al. 'Transparent tracing of microservice-based applications'. In: vol. Part F147772. 2019. DOI: 10.1145/3297280.3297403.

[83] *Service mesh: A critical component of the cloud native stack | Cloud Native Computing Foundation.* URL: https://www.cncf.io/blog/2017/04/26/service-mesh-critical-component-cloud-native-stack/ (visited on 20th Dec. 2020).

[84] Amazon Web Services. *Amazon ECS | Container Orchestration Service | Amazon Web Services.* URL: https://aws.amazon.com/ecs/ (visited on 2nd Jan. 2021).

[85] Benjamin H Sigelman et al. 'Dapper , a Large-Scale Distributed Systems Tracing Infrastructure'. In: *Google Research* (April 2010). ISSN: <null>.

[86] *State change updates sending after shutdown and throwing exceptions · Issue #726 · sitewhere/sitewhere.* URL: https://github.com/sitewhere/sitewhere/issues/726 (visited on 3rd Dec. 2020).

[87] R. Sterritt and D. Bustard. 'Autonomic Computing - A means of achieving dependability?' In: 2003. DOI: 10.1109/ECBS.2003.1194805.

[88] Giovanni Toffetti et al. 'An architecture for self-managing microservices'. In: 2015. DOI: 10.1145/2747470.2747474.

[89] Giovanni Toffetti et al. 'Self-managing cloud-native applications: Design, implementation, and experience'. In: *Future Generation Computer Systems* 72 (2017). ISSN: 0167739X. DOI: 10.1016/j.future.2016.09.002.

[90] João Tomás et al. 'Autonomic service operation for cloud applications: Safe actuation and risk management'. In: *Dependable Computing - EDCC 2021 Workshops*. Springer International Publishing, 2021. DOI: 10.1007/978-3-030-86507-8.

[91] Kennedy A. Torkura et al. 'CloudStrike: Chaos Engineering for Security and Resiliency in Cloud Infrastructure'. In: *IEEE Access* 8 (2020). ISSN: 21693536. DOI: 10.1109/ACCESS.2020.3007338.

[92] Leila Abdollahi Vayghan et al. 'Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned'. In: vol. 2018-July. 2018. DOI: 10.1109/CLOUD.2018.00148.

[93] Leila Abdollahi Vayghan et al. 'Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes'. In: 2019. DOI: 10.1109/QRS.2019.00034.

[94] Yuwei Wang. 'Towards service discovery and autonomic version management in self-healing microservices architecture'. In: vol. 2. 2019. DOI: 10.1145/3344948.3344952.

[95] Peter Waterhouse. *Monitoring and Observability — What's the Difference and Why Does It Matter? – The New Stack*. URL: https://thenewstack.io/monitoring-and-observability-whats-the-difference-and-why-does-it-matter/ (visited on 10th Dec. 2020).

[96] *Why Docker? | Docker*. URL: https://www.docker.com/why-docker (visited on 12th Dec. 2020).

[97] Li Wu et al. 'MicroRAS: Automatic recovery in the absence of historical failure data for microservice systems'. In: 2020. DOI: 10.1109/UCC48980.2020.00041.

[98] Li Wu et al. 'MicroRCA: Root Cause Localization of Performance Issues in Microservices'. In: 2020. DOI: 10.1109/NOMS47738.2020.9110353.

[99] Edith Zavala et al. 'HAFLoop: An architecture for supporting Highly Adaptive Feedback Loops in self-adaptive systems'. In: *Future Generation Computer Systems* 105 (2020). ISSN: 0167739X. DOI: 10.1016/j.future.2019.12.026.

[100] Xiang Zhou et al. 'Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study'. In: *IEEE Transactions on Software Engineering* (2018). ISSN: 19393520. DOI: 10.1109/TSE.2018.2887384.

[101] Xiang Zhou et al. 'Latent error prediction and fault localization for microservice applications by learning from system trace logs'. In: 2019. DOI: 10.1145/3338906.3338961.

[102] Haissam Ziade, Rafic Ayoubi and Raoul Velazco. 'A survey on fault injection techniques'. In: *Int. Arab J. Inf. Technol.* 1 (2 2004).

[103] *Zipkin API - Swagger*. URL: https://zipkin.io/zipkin-api/#/ (visited on 20th Aug. 2021).