INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Engenharia do Caos - Trazer ordem ao Caos

HUGO ANDRÉ DE FREITAS RIBEIRO
Setembro de 2024

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Chaos Engineering - Bring Order to Chaos

Hugo Ribeiro

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Computer Systems

Supervisor:
Dr. Paulo Gandra de Sousa, Professor, DEI/ISEP

Evaluation Committee:
President:

Members:

Porto, September 14, 2024

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore the work presented in this document is original and authored by me, having not previously been used for any other end.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, September 14, 2024

Abstract

Chaos Engineering is one of the hottest topics at the moment. Everyone in the field knows its great driving force, Netflix, which chose early on to develop this topic in an *open-source* way to support the community and allow others to get involved. However, having access to the tools and knowing how they are implemented in large companies is not enough for everyone to succeed in implementing Chaos Engineering. The research needed to determine where to start and the integration of this new concept into the project lifecycle can take a long time, a time that companies are often unwilling to give up due to other goals already set.

This thesis aims to help those who are interested in getting a good starting point by explaining the various concepts involved in the area and clarifying what can be done before starting to talk about experiments in production environments. Like everything in Information Technology, all the pieces are interconnected and Chaos Engineering is not an isolated science. Monitoring is one example of the key elements without which the implementation of Chaos Engineering practices would not make sense. Visualizing the state of the system before, during and after experiments is necessary to identify, verify and evaluate the system with or without the implementation of Chaos Engineering techniques.

The different levels of the chaos maturity model will also be presented, demonstrating the various ways in which it can be implemented in a system and the life cycle of a project.

Following the theoretical presentation, experiments will be carried out using chaos tools in an *open-source* project to demonstrate the use of good practices and their usefulness. The choice of tools will be made taking into account the specifications of the project in use in order to increase the deficiencies found with regard to its behaviour under stress. The respective hypotheses will be presented in a formulation, implementation and results presentation format to demonstrate the entire process.

Keywords: Chaos Engineering, Resilience, Chaos Hypotheses

Resumo

Chaos Engineering é um dos tópicos mais atuais do momento. Toda a gente na área conhece o seu grande impulsionador, Netflix, que desde cedo escolheu desenvolver este tópico de uma forma *open-source* para apoiar a comunidade e permitir outros de se envolverem. Porém ter acesso às ferramentas e saber como é implementado nas grandes empresas não é o suficiente para que todos consigam suceder na implementação de Chaos Engineering. A investigação necessária para conseguir determinar por onde começar bem como a integração deste novo conceito no ciclo de vida do projeto podem demorar bastante tempo, tempo esse que geralmente as empresas não estão dispostas a ceder devido a outras metas já estabelecidas.

Esta tese tenciona ajudar os mais interessados a ter um bom ponto de partida explicando os vários conceitos envolvidos com a área e esclarecendo o que pode ser feito antes de começar a falar de experiências nos ambientes de produção. Como tudo na informática todas as peças se interligam e Chaos Engineering não é uma ciência isolada. Monitorização é um exemplo das peças fulcrais sem a qual a implementação de práticas de Chaos Engineering não fará sentido. A visualização do estado do sistema antes, durante e depois da realização das experiências é necessário para identificar, comprovar e avaliar o sistema com ou sem a implementação de tecnicas de Chaos Engineering.

Será também apresentado os diferentes niveis do modelo de maturidade do caos, *Chaos Maturity Model*, onde serão demonstradas as diversas formas como este pode ser implementado num sistema e no ciclo de vida de um projeto.

Após a apresentação teórica serão realizadas experiências utilizando ferramentas do caos num projeto *open-source* de forma a demonstrar a utilização das boas práticas e a sua utilidade. A escolha das ferramentas será feita tendo em conta as especificações do projeto em uso de forma a aumentar as deficiências encontradas no que toca ao comportamento do mesmo aquando sobre stress. As respetivas hipoteses serão apresentadas num formato formulação, implementação e apresentação de resultados de forma a demonstrar todo o processo.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

API	**A**pplication **P**rogamming **I**nterface
AWS	**A**mazon **W**eb **S**ervices
ChAP	**C**haos **A**utomation **P**latform
CLI	**C**ommand **L**ine **I**nterface
CMM	**C**haos **M**aturity **M**odel
CPU	**C**entral **P**rocessing **U**nit
DiRT	**Di**saster **R**ecovery **T**esting
FIS	**F**ault **I**njection **S**ervice
FIT	**F**ault **I**njection **T**echnique
GCP	**G**oogle **C**loud **P**latform
IT	**I**nformation **T**echnology
NAS	**N**ational **A**cademy (of) **S**cience
OSP	**O**pen-**S**ource **P**roject
RAM	**R**andom-**A**ccess **M**emory
TCP	**T**ransmission **C**ontrol **P**rotocol

Chapter 1

Introduction

This chapter explains the problem and how it impacts those interested in getting involved with Chaos Engineering, it describes the main objectives, along with the methodology used during the investigation. The chapter concludes with an overview of the limitations and the structure of this work.

1.1 Problem

Traditional tests have limitations when testing a complex system as they often fail in measuring resilience. Adopting Cloud environments helped this tendency to grow even more. Within every company there are multiple systems, and they differ from each other, meaning that resilience can not be tested and measured in the same way. Guaranteeing the resilience of a system is hard to accomplish, and many surprises emerge only when the production environment is already deployed. Chaos Engineering (CE) has been seen as part of the answer allowing you to raise havoc in the system through experiments.

Despite the potential benefits, embracing CE comes with challenges. There are already several tools available, but it can be overwhelming and time-consuming for a company to integrate CE into their daily practices.

- Where should we start?
- What are the good practices of Chaos Engineering?
- Are we and our systems mature enough to start with Chaos Engineering?

These questions are only a few of the ones that discourage organizations from embarking on their CE journey.

Before implementing CE solutions, other components that will already improve the understanding of the system can be implemented/configured, such as Failure Thresholds, Monitoring, Time to Recover and Knowledge center regarding past issues. Having a good understanding and knowledge about how your system works is crucial when it comes to applying experiments correctly, guaranteeing their validity and allowing them to stop when major issues occur.

Upon research, the available tools will be filtered by the system specifications. In this way it will be possible to carry out experiments in the various areas of the system, such as Infrastructure and Platform.

The Chaos Maturity Model (CMM) can be used to measure the maturity of CE practices within an organization. CMM has two key metrics to accomplish this assessment, **Sophistication** and **Adoption** [1] [2]. Sophistication represents the level of expertise and effectiveness in conducting chaos experiments, offering insights into their validity and safety. The model categorizes sophistication into four levels, ranging from elementary to advanced, with each level indicating the organization's proficiency in chaos experimentation. On the other hand, Adoption provides your system with experimentation coverage by considering both depth and breadth measurements. It is categorized into four levels as well, each delineating the confidence in the system and the exposure to vulnerabilities.

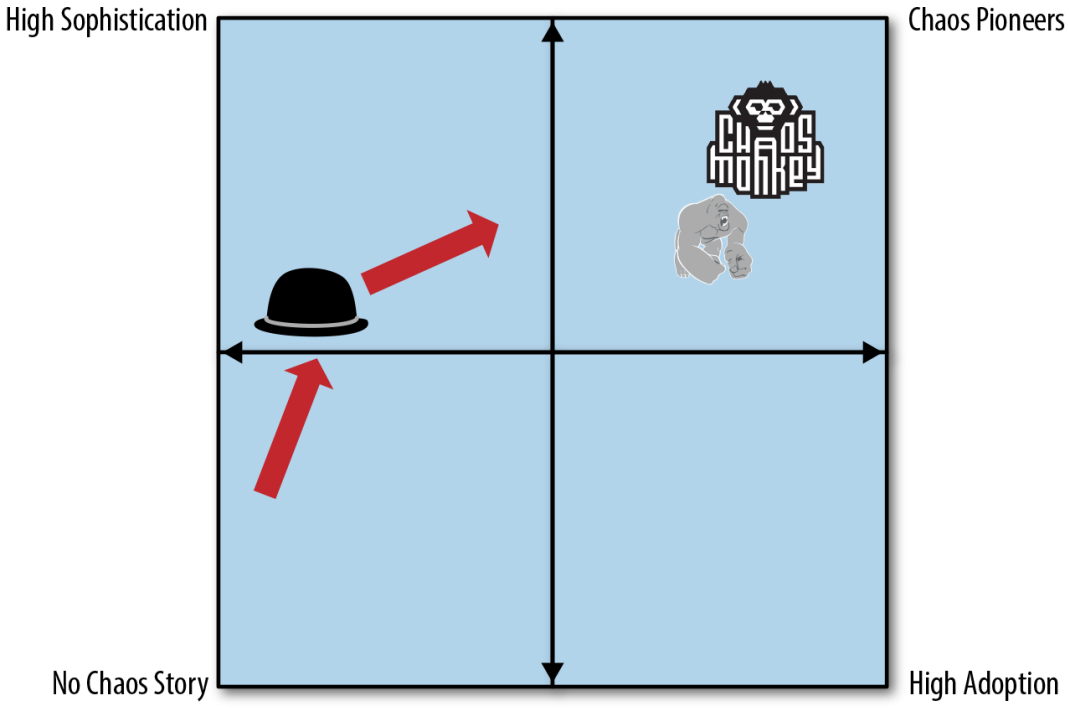

Figure 1.1: Chaos Maturity Model [2]

Before designing a CE solution and applying the CMM framework, other components that are able to improve your understanding of the system can be implemented: among these, Failure Thresholds, Monitoring, Time to Recover and Knowledge center. These will be of help later when starting experiments with CE.

1.2 Objectives

The primary objectives of this project are to comprehensively explore CE and its application, as well as how chaos can be introduced in a project. This work aims to answer critical questions and address challenges associated with the integration of CE into the software development life cycle and the types of chaos that can be introduced.

- **Understanding Chaos Engineering:** Defining Chaos Engineering and its relevance to software development, exploring its integration across different phases of the project life cycle.

- **Challenges of Traditional Testing:** Investigating the limitations of traditional testing methodologies, particularly in the context of complex systems, laying the groundwork for the necessity of Chaos Engineering.
- **Applicability of Chaos Engineering:** Exploring how a system can improve with CE and which are the pre-requirements before implementing a solution.
- **Improving Resilience Maturity:** Examining how CE can contribute to enhance the resilience of the systems, understanding its potential impact on overall system robustness.
- **Implementation Strategies:** Investigating effective strategies to implement CE across various projects, considering the differences in tools and methodologies that may arise.
- **Tool Assessment and Use Cases:** Evaluating a range of CE tools in order to identify the most suitable one according to each specific use case.

These objectives collectively form the framework for the project, aiming to provide valuable insights into CE, to assess its applicability, and to establish practical implementation strategies. The second component will be the application of the presented methods using an open-source project where it is intended to cause chaos. CE has two major focuses: the first one consists in provoking the chaos and it will be investigated in this project; the second deals with observing how the people involved behave. Since an open-source project will be used, the second focus will not be studied.

1.3 Methodology

This chapter will present the research questions that this project aims to answer and the systematic approach, which is the methodology mainly used to carry out this research. It will also include an explanation of the snowball approach. This second approach was implemented to support the first one increasing the number of research papers retrieved and gathering more detailed information. In order to retrieve papers, a simple search was performed using keywords that were chosen based on the current topic and the problem statement.

1.3.1 Research Questions

The research questions below were defined to help the initial gathering of essential information about the topic to be studied.

RQ1: What is Chaos Engineering?
As Chaos Engineering is the main focus of this work, it is important to have a broad and accurate knowledge of this topic. Answering this question involves studying the topic in order to learn the technical terms, the characteristics and the best practices to adopt.

RQ2: Why Chaos is not part of the system/project life cycle?
Although it has been around for a few years, it is only recently that Chaos Engineering has gained more attention and it has also become a goal within companies. However, the vast majority of companies that are already adopting Chaos Engineering are well-known companies, such as Netflix and Google. It is therefore necessary to understand what difficulties exist when entering this phase.

RQ3: How can Chaos be implemented?
After acquiring knowledge about the main topic, which is Chaos Engineering, it is necessary

to understand how it can be implemented. More specifically, it is important to understand which is the focus of the experimentation, what CE can be experimented with, whether the same experiment can be applied to different systems, and how the experimentation can be applied safely. Gathering the right information will lead to an efficient implementation of CE, which will be explained further in this work.

1.3.2 Systematic Research Approach

Upon extensive research across various academic databases, including but not restricted to ACM Digital Library, Google Scholar, and Repositório Científico do Instituto Politécnico do Porto, and technical books, using the O'Reilly online platform, a selected number of academic papers was chosen to be used as the scientific foundation of this project.

Keywords such as "Chaos", "Chaos Engineering" and "Resilience" were the main keywords of the project. However, others like "Chaos Monkey" and "Chaos Maturity Model" were searched due to their significant value within the topic. Using these keywords and Booleans operators, a complex search was performed. Only results dating back to the last 5 years were taken into account to reduce the number of research papers.

Example: *[[Title: "chaos"] OR [Title: "resilience"]] AND [Abstract: chaos engineering] OR [Abstract: chaos] OR [Abstract: resilience]] AND [E-Publication Date: Past 5 years]*

Only papers putting more emphasis on the subject, that is those including definitions, tools and key topics, were kept to support this research. This screening was based on titles and abstracts, and then a thorough assessment of the paper's content followed.

All this process was carefully carried out to ensure that the selected ones would indeed contribute substantially.

1.3.3 Snowball Sampling

A snowball sampling technique was incorporated in the more promising papers to increase the research sample and quality. During the thorough content assessment, papers with some intriguing information regarding the main topic were then followed to the referred article/book, allowing for more in-depth research and learning.

1.3.4 Ethical Considerations

Throughout the research process, ethical considerations remained paramount in order to maintain the integrity and transparency of the work. All sources consulted were properly cited in accordance with academic standards, ensuring that the intellectual contributions of the original authors were appropriately recognised. This careful adherence to citation practices reflects a commitment to respecting the work and ideas of others, which is fundamental to responsible academic research.

As the study did not involve the collection or analysis of personal, sensitive or confidential data, there were no direct privacy concerns that needed to be mitigated. However, responsible data handling protocols were followed at all stages of the research to ensure that any external data used was handled with care and in accordance with ethical research guidelines.

In addition, artificial intelligence tools were used to assist in the process of sentence validation and rephrasing. These tools were used with great care to maintain academic integrity, ensuring that their use did not compromise the originality or authenticity of the research.

1.4 Limitations

As stated previously one of the major focuses of CE will not be studied due to the fact that an open-source project will be used.

The project is constrained to on-premises deployment, meaning that the exploration of cloud environments will remain purely theoretical. This constraint imposes specific requirements not only for the project itself but also for the selection of tools. Given the absence of a cloud environment, tools designed to terminate cloud instances or simulate issues with availability zones will not be chosen. Due to this fact, the project size will be also limited to the resources available.

1.5 Structure

This project has the following structure:

- Introduction: an overview of the problem and of the objectives of this project is presented together with the methodology used.
- State of the Art: this chapter describes different topics to give the necessary information for a better understanding of the work done.
- Assessment of Open Source Application Options: the open-source projects found that could be used for the purposes of this work are presented together with the criteria used to select the most complete.
- Hands-On Chaos Experiments: this chapter describes all the processes related to the experiments, definition, implementation and results.
- Conclusion: this paragraph draws the conclusions and outlines the learning outputs achieved thanks to this project

Chapter 2

State of the Art

This chapter contains the knowledge necessary to better understand the work done. It first provides some definitions and the context of use of the word resilience, together with today's testing methods. Furthermore, it explains what Chaos Engineering is, illustrating how it evolved throughout history. Some case studies of other companies that have embraced Chaos Engineering are taken into consideration. Finally, a small sample of chaos tools is presented and investigated.

2.1 Resilience

Every day the technology world is taking a step forward. New ideas, new tools, new softwares and new attacks are created. Nowadays, the world is more techno-logic than analogic, and information, jobs and hobbies are emerging from this drive to the future. But can systems hold the weight of millions of users, together with attackers and faulty equipment? The time when writing code was all that mattered is long gone and softwares need to be functional and available every day at any time.

A system that is always in the dangerous zone and that at any change of environment is pushed beyond its limits is called a Brittleness System. "Brittleness represents the precipitous collapse of performance begotten by stressors, which push the system beyond its ability to manage disturbances and variations gracefully" [3]. Resilience has been inherited and applied to the Information Technology (IT) world as a way to classify systems and softwares. It is defined as the "ability of the system to withstand a major disruption within acceptable degradation parameters and to recover within an acceptable time" [4] OR "the ability to prepare and plan for, absorb, recover from, and more successfully adapt to adverse events" as defined by the National Academy of Science (NAS) [3].

The more a system is resilient, the better the user experience will be, as resilience will prevent system errors from impacting usability. [5]

Across the papers examined, it appears to be some common ground about key factors when referring to resilience. Since there is not an agreement about which are the pillars of resilience, the following will be considered as prerequisites that a system needs to have before implementing chaos experiments: *Thresholds*, *Recovery Time*, *Monitoring* and *Knowledge* [3] [6] [7] [8].

Even if considered as separate metrics, these prerequisites form an interrelated foundation that enhances a system's ability to withstand and recover from disruptions, making them crucial for implementing and testing resilience.

2.1.1 Failure Thresholds

The perfect software has yet to come, but even if it existed, it could be compromised or fail. Knowing the system/software limits is an essential metric to guarantee its normal state, but each system has its own criteria [3]. Physical resources, software limitations, and the number of users are examples of environment criteria that can make a threshold number vary.

There is no standard right number or percentage that has to be applied to a system. Sometimes only time will reveal the value that better defines your system and usually even within the same system threshold numbers vary depending on the resource used.

2.1.2 Recovery Time

Recovery Time is the amount of time considered tolerable to restore a system for it to be fully operational after a disaster or failure. It is usually measured in seconds, minutes or days and it can vary according to criticality. A real-life example could be that of bank systems where even the smallest downtime could have a huge impact on the economy.

2.1.3 Monitoring

Establishing thresholds and recovery time is the first step to creating a resilience system. Nonetheless, knowing the limits of your system is not enough if you do not monitor your system state [8]. If your system makes the Central Processing Unit (CPU) throttling, but it is not monitored, once you identify the problem, you will be already within the count of your recovery time.

Monitoring is the ability to see into the system, allowing it to be analysed.

2.1.4 Knowledge

Incidents are inevitable, the way they are handled is another matter. What are the steps to be taken after an incident? Are you able to find the root problem and replicate it? Is the problem and its solution shared among the project personnel? Do you have a Knowledge Center where you can search for past problems?

These are some of the questions that can be used to evaluate how incidents are treated. Learning from past experiences is also a way to improve the resilience of the system [8].

2.1.5 Testing

Guaranteeing the system functionality is always the first step when developing softwares, either new softwares or upgrades of an existing one. Unit tests and Integration tests are the first phase of testing to be applied when developing softwares.

- Unit Tests - Unit tests aim to test specific components, functions, or use cases of the software. Each test is independent from the others, but they should all produce the same result every time as long as the test does not change.
- Integration Tests – A software is made of different components, even more today with the micro-services adoption. It is necessary to test the different interactions between each component to promote the good functioning of the software as a whole.

Integration testing will test how the different components interact with each other raising eventual interaction problems.

Unlike the tests above, CE aims to discover what you do not know [9]. Unit tests and integration tests have the purpose of confirming the expected behaviour of the software, while CE sets an incognito regarding what is going to happen (Is it going to fail? Yes, where? Why?). CE creates new knowledge through experimentation, contributing to either increase confidence or improve knowledge regarding system properties [10].

Another difference that can be set is the testing scope. Traditional tests are more feature-oriented or more oriented to the software itself. At the same time, CE deals with the system as a whole, thus targeting not only the software but also the infrastructure where it is deployed, possible changes in the environment, as well as interactions between the different services that compose the system.

2.2 Chaos Engineering

Chaos Engineering is "the discipline of experimenting on a system to build confidence in the system's capability to withstand turbulent conditions in production" [11]. CE allows improvements regarding the system resilience identifying its weaknesses before they manifest: "Chaos engineering's sole purpose is to provide evidence of system weaknesses" [12].

These weaknesses in the system are sometimes called *dark debt*. Dark debts are found across complex systems and are not recognizable at the time of creation. Unexpected interactions with the system are generally responsible for the appearance of the so-called dark debt, which leads to unforeseen anomalies [13] [14].

Dependencies among services or even the hardware limits might lead to dark debts that will appear at the most unexpected and improbable time. CE has come to provide a helping hand before something like this might happen [15].

As mentioned above, CE is meant to find new information regarding a system, including how it handles the failure of services, service dependencies or how it behaves due to changes in the environment. CE is not meant to verify an existing problem. Thus, if it is known that the system has a specific problem, CE is not needed for that issue . "[...]'If I know there's a weakness, that the system will fail if a condition occurs, do I need to do chaos engineering?' It's a fair question, and the immediate answer is no[...]" [16].

One of the key features of CE is where the tests are run. Unlike most types of tests and what good practice says, CE is supposed to be run against a production environment [15]. These tests are usually scheduled and announced to clients since they can cause downtime if not planned correctly. During the procedure, teams closely monitor the environment and its behaviour to validate the experiments and to take action if the system fails outside of the scope. This phenomenon is known as *Blast Radius* [17]. The magnitude of the blast radius helps define boundaries to the experiment in order for the team to know when to stop the experiment, if needed. [18]. These experiments are created based on a hypothesis, which helps to define the scope of the test, being one of the main characteristics of CE. It is necessary to idealize a scenario and formulate a hypothesis that will lead to an experiment where the same hypothesis will be proved right or wrong.

As already stated previously, CE can be applied to multiple areas of the system to reach a high level of resilience. [19]

- People, practices, and processes
- Applications
- Platform
- Infrastructure

These four areas will be then specified according to the system to be used. Each hypothesis designed shall fall within the specified areas and depending on the blast radius it can cover more than one of them.

According to [11], to design a valid experiment it is necessary to elaborate a scientific method.

- Define the system standard behaviour - "Steady State"
- Create a Hypothesis involving the steady state.
- Implement the testing case by simulating real-world events.
- Minimize the blast radius

The described procedure can be as specific as necessary. For example, it is possible to create a hypothesis regarding a system feature, a set of actions or even the behaviour of the infrastructure. If the experiments are well elaborated, it is possible to get to a point where it gets hard to disturb your steady state, "The harder it is to disrupt the steady state, the more confidence we have in the behaviour of the system" [11].

2.2.1 History

Although CE is still a relatively new approach and is not yet entirely part of the software life cycle, it has already moved from “Should we do Chaos Engineering?” to “What’s the best way to get started doing Chaos Engineering?” [20].

Netflix was the pioneer of CE when in 2008 it embarked on a journey to transition from its datacenter to the Amazon Web Services (AWS) cloud. After encountering issues with the datacenter that disrupted their services, they invested in finding a way to test and prevent failures from happening. In the following years, CE began to be implemented as a method to intentionally introduce failures and address resilience issues in their system. This investigation and drive for a more resilient system gave birth to a new tool, Chaos Monkey [21].

Chaos Monkey, being the first tool specifically designed for CE, had a huge impact on the field [22]. Its status as an open-source software further catalysed the widespread adoption of Chaos Engineering.

2.2.2 Evolution

Chaos Monkey is a tool "responsible for randomly terminating instances in production" [23]. This tool was meant for developers to introduce chaos into their environments in a controlled way, so they could observe how their systems behaved in case of unexpected events. When this tool was developed, it could terminate a random AWS EC2 instance, allowing Netflix to experiment if resilience had been prompted into the services.

As time passed and new challenges emerged, the existing capability became insufficient. To keep testing resilience it was necessary to examine not only the infrastructure, but also the interactions among services. This shift prompted a renewed emphasis on addressing resilience, leading to the incorporation of the Fault Injection Technique (FIT) into resilience testing. Its use contributed to create a more in-depth comprehension of service interactions and responses to failures.

Netflix decided to integrate the FIT to enhance the functionality of their chaos engineering tool, improving their system resilience through experimentation [24]. To end this era the only thing that was missing was to automate all experiments, so that they could be run randomly against their environments. Chaos Automation Platform (ChAP) was created joining what Netflix had created and automating it [25].

2.2.3 Fault Injection

FIT involves deliberately creating unconventional events in the application and in the infrastructure to test the effectiveness of the implemented recovery mechanisms. This process ensures that the application's resilience is consistently assessed [26] by observing its behavior under abnormal conditions.

Previously, examples of what can be put under stress in a system have been presented, such as CPU consumption. FIT can help introduce issues in a safe and controlled way. [27]. The use of FIT has become a common practice when applying CE into resilience testing. [28] The synergy between FIT, able to deliberately stress the environments, and CE experiments increases the knowledge regarding the system and its resilience level.

Integrating FIT into the tools developed makes it easy to automate testing during the whole application life cycle.

Another improvement that can be accomplished by using FIT is system fault tolerance. Fault tolerance is the ability of the system "to continue operating despite failures or malfunctions" [5]. In certain situations either it is not possible to fix the issues or the cause of the problem has yet to be found. In these scenarios it is important to design and implement ways that help the system operate as normal as possible to handle the issue gracefully.

2.2.4 Case Studies

Facebook

In 2014, Facebook added a whole new dimension to the idea of an infrastructure stress test. The company shut down completely one of its data centers to see how the safeguards it had put in place for such incidents performed in action. This level of testing emerged after the impact of the floods of Hurricane Sandy in 2012 in New Jersey [26].

Jay Parikh, global head of engineering at Facebook, discussed the exercise in his keynote presentation at the company's conference in San Francisco. Described as a major event, the experiment involved turning off "tens of megawatts of power" for a whole day. The goal was to study how the system responded and assess its performance during this period [29].

It was not specified which of Facebook's data centers was shut down. Facebook has facilities in Oregon, Iowa, North Carolina, and Sweden, and leases wholesale data center space in California and Virginia.

The company did run some “fire drills” before the test to get ready, and even though there were doubts that the team would pull the plug, it needed to happen. “We turned the entire region off,” Parikh said. And the prep work paid off. “It was actually pretty boring for us,” he said [29].

Not everything worked as expected, and some problems had to be addressed. But the overall system resisted, and the applications stayed up.

An exercise like this falls into one of the key mindsets of engineering at Facebook, which is embracing failure, Parikh said. Facebook encourages its engineers to take big risks – without being reckless – and does not punish those who take them and fail.

Microsoft Azure

Inspired by Netflix and its pioneering CE tool, Microsoft developed the Search Chaos Monkey. Upon its release into the test environment, the tool introduced a dynamic search service, characterized by a continuously changing topology and state. Regular operational calls were made to this service to ensure its reliability and performance.

This "simple" test had already proved useful since it managed to identify issues with the provision and scaling of workflows: "We’ve caught several bugs this way before they had a chance to escape into production" [30].

For better control, levels of destruction have been incorporated into Search Chaos Monkey, among which **Low Chaos**, **Medium Chaos**, **High Chaos** (Table 2.1).

Table 2.1: Search Chaos Monkey - Levels of Chaos

Low Chaos	Failures that the system can recover from gracefully with minimal or no interruption in availability
Medium Chaos	Failures that the system can recover gracefully but it might degrade the service performance
High Chaos	Major failures that interrupt the service availability

Additionally, a fourth level called **Extreme Chaos** was specifically designed to highlight issues that could lead to data loss or system failure without triggering alerts. Microsoft aims to enhance system resilience by minimizing chaos levels, even if it requires rendering the service unavailable temporarily until the underlying issues are addressed [30].

To increase the number of bugs and issues found by this tool automation is necessary, so failures can be injected as a trigger.

One of the scenarios where this tool showed good results was when a service was found emitting a low-priority alert. Later the team discovered that the alert involved a required background service that was not executing. This meant that the cluster was executing with an unknown state and that the alert was not providing the correct problem, classifying the issue as an Extreme Chaos level.

Due to the lack of ability to replicate the failure, a workaround was implemented to decrease the severity of the issue found. This was done by changing the alert level and notifying the on-call engineer, so that he could apply the manual fix. The root problem was found after guaranteeing that the customer services were functioning properly. By means of fault

injection, used to artificially induce latency into the requests, it was confirmed that the problem was related to a "stall" happening when making external calls.

Having identified the issue the team was able to fix it quickly by decoupling the component and adding some redundancy. Lastly, the automated chaos operation proved that the original issue was resolved, allowing the team to reduce even more the alert level, making it unnecessary to contact the on-call engineer [30].

Google

Google has started havocking chaos within its facilities. In an interview in 2016 Kripa Krishnan, aka Google's queen of chaos, explained how they performed chaos experiments to guarantee that no matter what happened Google would be able to keep running. In this interview, she presented sample tests and showed how the team members gather while the tests are in progress: "We have intense situations in the war room. There's like 20-30 people sitting there" [31].

Since the major goal of CE is to break things, things often go wrong. One of the use cases provided is a Google app with millions of users that slowed down during an orchestrated test on Google's network. Those 15 minutes were "yipping" since everyone was trying to determine if the issue was related to the ongoing experience or not.

As previously stated in chapter 1.2, one of the objectives when applying CE experiments is to observe how the people involved are going to behave. The no internet use-case is one of these situations. If the internet was down, today's traditional communication tools, such as chat rooms and Google Hangouts, would be unavailable, obliging the ones involved to go back to the good old telephone. Several situations emerged in this scenario: people could not find any "bridge phone number or even how to dial a phone to get an outside line" [31].

Another use case shared by the Disaster Recovery Testing (DiRT) team was related with credit card. A data center was put to test regarding Google's emergency funds procedure. Scenarios such as floods and fires were thrown at the emergency funds team to get them to activate the funds, but each time the engineers in charge came up with orthodox solutions to solve the problem without activating the emergency funds.

Not even in one of the scenarios the personnel in charge ever sent any emergency-fund money. Nonetheless, they never let Google site go down: "They take it seriously, this role-playing stuff" [31].

2.3 Project

To implement CE practices an open-source project will be used. It will be deployed on local resources in an on-premises environment. To align with key considerations like observability and thresholds, it is crucial to address any deficiencies in the project beforehand. Therefore, if needed, an initial exercise will be conducted to identify potential issues or incorporate observability features, enabling us to observe and validate results.

Given hardware limitations, the project's size will be a factor to be taken into account. Opting for a microservice architecture is recommended, as CE is specifically tailored for complex systems. This approach facilitates the expansion of services and the exploration of dependencies between them, aligning with the principles of Chaos Engineering to stress-test and uncover vulnerabilities in distributed systems.

2.4 Tools

As explained in detail in section 2.2, there are several areas where chaos experiments can be conducted. The choice of tools varies depending on the system requirements, which make the selection from the several tools available a complex decision.

This section will cover some available tools that can be of use based on the chosen project and the designed system. All tools are open-source, meaning that more popular tools might not be included in this list, such as AWS Fault Injection Service (FIS) [32] or Gremlin [33], both offering CE as a service. The sample of tools presented was gathered from another project [18] and other tools found.

2.4.1 Chaos Monkey

Figure 2.1: ChaosMonkey [34]

The first tool designed as a CE tool was simple. Since Netflix had its services on a cloud provider, this tool could randomly terminate instances in the system environment.

To successfully deploy Chaos Monkey, Spinnaker and a MySQL database are needed. Spinnaker is an open-source, multi-cloud continuous delivery platform used to deploy and manage applications. It provides a flexible and extensible platform that supports the entire application delivery life cycle. Spinnaker was originally developed by Netflix and later open-sourced; it is now maintained by the Spinnaker community [35].

Chaos Monkey makes use of cron jobs to create a schedule of terminations. The configuration file is a .toml file used to contain not only the database and Spinnaker's configurations but also Chaos Monkey's application settings, such as starting and stopping time, script paths, cron file, etc [36].

To determine the instance to be terminated, Spinnaker Application Progamming Interface (API) is used. It exposes an *isDisabled* flag, indicating if a group is disabled or not. The application is then capable of filtering, ensuring that only terminations from active groups occur. A random instance is then picked from that group and terminated at a random time within the timeframe set in the configuration file. This randomness is accomplished using a set of mathematical equations that can be analysed on the official site [37].

2.4.2 Simian Army

Since Chaos Monkey was the first tool developed, with time it became incapable to respond to the developers' needs. The Simian army has integrated the capabilities of Chaos Monkey and added other chaos strategies to improve even more the reliability of the system.

Table 2.2: Simian Army Strategies [38]

Simius Mortus	Shuts down the instance using the EC2 API. This is the classic chaos monkey strategy.
Simius Quies	Removes all security groups from the instance and moves it into a security group that does not allow any access. Thus the instance is running, but cannot be reached via the network. This can only work on VPC instances.
Simius Amputa	Force-detaches all EBS volumes from the instance, simulating an EBS failure. Thus the instance is running, but EBS disk I/O will fail.
Simius Cogitarius	Runs CPU-intensive processes, simulating a noisy neighbour or a faulty CPU. The instance will effectively have a much slower CPU.
Simius Occupatus	Runs disk-intensive processes, simulating a noisy neighbour or a faulty disk. The instance will effectively have a much slower disk.
Simius Plenus	Writes a huge file to the root device, filling up the (typically relatively small) EC2 root disk.
Simius Delirius	Kills any java or python programs it finds every second, simulating a faulty application, corrupted installation or faulty instance. The instance is fine, but the java/python application running on it will fail continuously.
Simius Desertus	Null-routes the 10.0.0.0/8 network, which is used by the EC2 internal network. All EC2 <-> EC2 network traffic will fail.
Simius Nonomenius	Uses iptables to block port 53 for Transmission Control Protocol (TCP) & UDP; these are the DNS traffic ports. This simulates a failure of your DNS servers.
Simius Noneccius	Puts dummy host entries into /etc/hosts so that all EC2 API communication will fail. This simulates a failure of the EC2 control plane. Of course, if your application does not use the EC2 API from the instance, then it will be completely unaffected.
Simius Amnesius	Puts dummy host entries into /etc/hosts so that all S3 communication will fail. This simulates a failure of S3. Of course, if your application does not use S3, then it will be completely unaffected.
Simius Nodynamus	Puts dummy host entries into /etc/hosts so that all DynamoDB communication will fail. This simulates a failure of DynamoDB. As with its monkey relatives, this will only affect instances that use DynamoDB.
Simius Politicus	Uses the traffic shaping API to corrupt a large fraction of network packets. This simulates the degradation of the EC2 network.
Simius Perditus	Uses the traffic shaping API to introduce latency (1 second +- 50%) to all network packets. This simulates the degradation of the EC2 network.

Each strategy outlined in Table 2.2 can be enabled or disabled through the configuration file. Given that this tool, much like Chaos Monkey, is designed for the cloud and tailored to AWS, the project must be deployed in such an environment. While it may be unlikely to be utilized for this particular project, cloud systems remain a prevailing trend, making this tool a good option for AWS systems.

2.4.3 Pumba

Figure 2.2: Pumba [39]

Pumba is a chaos testing command line tool for Docker containers. Being oriented to the platform layer of the system, Pumba is capable not only of interfering with the container readiness but also of causing other disturbances like network failures and resource stress [39].

Since Pumba uses Linux tc tool for the network emulation, the package needs to be either installed within the container or an image that already contains the tc tool has to be used. For the stressing techniques, the *stress-ng* docker image is used.

2.4.4 Chaosblade

Figure 2.3: Chaos Blade [40]

ChaosBlade, an Alibaba open-source experimental injection tool, aligns with the principles of chaos engineering and experimental models to enhance the fault tolerance of distributed systems. Its primary goal is to ensure business continuity during enterprise transitions to the cloud or migration to cloud-native systems. This tool is part of the MonkeyKing project and it was designed taking into consideration almost a decade of experiments of failure testing and drill practices at Alibaba. It has also been pointed out as easy to use and with different capabilities from the other tools presented above [40].

This tool can be used in the following scenarios

- Basic resources: such as CPU, memory, network, disk, process and other experimental scenarios
- Java applications: such as databases, caches, messages, JVM itself, microservices, etc. You can also specify any class method to inject various complex experimental scenarios
- C ++ applications: such as specifying arbitrary methods or experimental lines of code injection delay, tampering with variables and return values
- Container: such as killing the container, the CPU in the container, memory, network, disk, process and other experimental scenarios

- Cloud-native platforms: such as CPU, memory, network, disk, and process experimental scenarios on Kubernetes platform nodes, Pod network and Pod itself experimental scenarios such as killing Pods, and container experimental scenarios such as the aforementioned Docker container experimental scenario;

While the previous tools mainly affect the infrastructure state, like the instance readiness, this tool operates on the application and the platform level.

2.4.5 Chaos Monkey for Spring Boot

Figure 2.4: Chaos Monkey for Spring Boot [41]

Adopting the Chaos Monkey approach but focusing on applying it to Spring Boot allows for a more effective testing of applications, particularly during operation [41].

There are two mechanisms for enabling Chaos Monkey within a Spring Boot environment. The initial approach involves incorporating it into standard application dependencies, typically specified in files such as pom.xml or build.gradle(.kts). Alternatively, one may activate Chaos Monkey as an external dependency during the initiation of the Spring Boot application. Below is presented the "how to" respectively for each method.

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>chaos-monkey-spring-boot</artifactId>
    <version>3.0.2</version>
</dependency>
```

Listing 2.1: Method 1 - Maven

```
java -jar your-app.jar --spring.profiles.active=chaos-monkey --chaos.
    monkey.enabled=true --chaos.monkey.watcher.service=true --chaos.
    monkey.assaults.latencyActive=true
```

Listing 2.2: Method 2 - Spring Profile

It has available HTTP endpoints that can be used to activate/deactivate as well as get information regarding configurations [42].

2.4.6 ChaosToolkit

Figure 2.5: Chaos ToolKit [43]

The Chaos Toolkit provides a framework and tools for orchestrating and automating chaos experiments. It supports a variety of platforms, including cloud environments and on-premises infrastructure.

Adopting the Chaos as Code paradigm, this tool facilitates the declaration and storage of experiments through JSON or YAML files. As an open-source solution, it boasts extensibility, allowing the use of existing plugins or the creation of new ones [44]. The number of available plugins facilitates the integration with known applications [45], including Slack for notifications and Grafana and Prometheus for observability. Chaos Monkey for Spring Boot, chapter 2.4.5, offers a dedicated plugin for integration with Chaos Toolkit [46].

In order to use this tool Python, and specifically the official version Python 3.7+ [47], is needed. Post-installation, it can be conveniently accessed from the command line.

One of the plugins available is Toxiproxy [48]. As the name suggests, it provides an intermediary between the client and the server sides while creating adversities to the upstream or downstream, being ideal for simulating network conditions [49].

Even though ChaosToolkit can be integrated with Python, this tool also exists fully independent with its own server and Comand Line Interface (CLI) [50]. Toxiproxy uses toxics [51], name given to the adversities caused by the tool, to affect the communication flow. The default installation brings useful toxics as part of the tool which affects latency, bandwidth, reset_peer (simulating the TCP RESET), etc. If this is insufficient, Toxiproxy allows you to customize it and add your own toxics [52].

Table 2.3: Tools Surveyed by [18] - Adapted

Tool ID	**Tool Name**	**Description of Functionality**	**Suitable Applications to Test**
T1	Chaos Monkey	Used for instance termination. Terminates virtual machines and containers.	Applications managed with Spinnaker (a continuous delivery platform)
...	...	...	...
T3	Chaos Toolkit	Used for instance termination, simulating network problems and stressing machines. Exact functionality depends on installed drivers.	Applications running on AWS, Azure, Cloud Foundry (a cloud computing platform) or GCP. Applications managed with Kubernetes
...	...	...	...
T11	Pumba	Used for instance termination and simulating network problems. Terminates Docker containers and injects network delays.	Applications containerized with Docker
...	...	...	...
T14	Chaos Monkey for Spring Boot	Used for instance termination, simulating network problems and finding implementation faults in an application's source code. Terminates instances of, adds delays to and throws exceptions in Spring Boot Controllers, RestControllers, Services, Repositories and Components.	Spring Boot (Java) applications
...	...	...	...
T18	Chaosblade	Used for instance termination, simulating network problems, finding implementation faults in an application's source code and stressing machines. Exact functionality depends on installed executors.	Applications containerized with Docker. Applications managed with Kubernetes. Java and Go applications

Chapter 3

Assessment of Open Source Application Options

In this chapter, we identify and assess various Open-Source Projects (OSPs) to find candidates that align with our requirements. A comparative analysis of these projects is conducted in order to find the projects which are more suitable for chaos experimentation. This analysis leads to the selection of the most promising project for our purposes.

3.1 Project Choice

This section will present the projects that met the requirements set for this work and the respective pros and cons. The OSP should have:

- Microservice Architecture
- Monitoring/Health Check
- Easy Deployment
- Docker Based

These elements were chosen based on set criteria, such as the importance of monitoring, and the limitations regarding available hardwares, described in section 1.4

3.1.1 PiggyMetrics

Project Description Architecture

A simple financial advisor app built using Spring Boot and Spring Cloud [53] has a microservice architecture, Fig 3.1 allowing for the services to be installed separately using docker as its primary platform. It is a small-sized application, whose number of actions is limited. It is composed by eleven microservices, including the database, the monitoring and the gateway.

The basic services responsible for the application functionalities are the statistics, the account and the notification service. All of them are Rest APIs, each one with a database: MongoDB. The gateway service is the entry point for the application and it is responsible for the routing of the requests to the respective services.

Monitoring

The main mechanism is the Eureka Service, a SpringBoot discovery service that enables services to register and locate one another. It also facilitates quick troubleshooting by

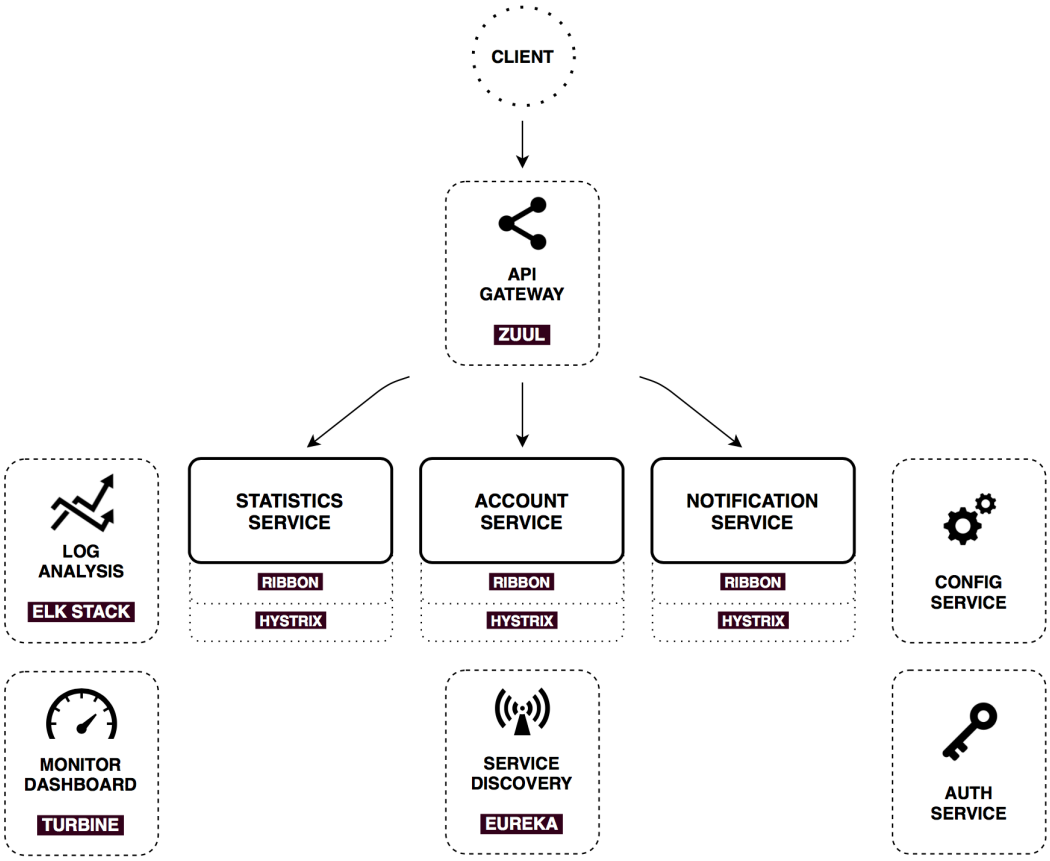

Figure 3.1: Piggy Metrics High-Level Architecture [54]

indicating whether all services are operational. A monitoring dashboard is provided through the use of Hystrix and Turbine. Hystrix uses Circuit Breaker Pattern, which gives control over some statuses and enables the implementation of a fallback method once the defined limits are reached [55] [56]. Each graph helps analyse the health status and traffic volume using a colour coding system and numerical data, such as error percentage, latency, and number of connections. Turbine is used as an aggregator: it allows multiple Hystrix dashboards to be accessed through a single connection point and enables centralized monitoring of all services and configured methods [57].

One minor issue with the current monitoring system is that if it is opened after the experiment has started, previously generated metrics will not be available on the dashboard. This will make it impossible to observe/analyse the application status throughout time. Another negative aspect of this level of monitoring is the non-existent way to create alarms to notify when the system is unhealthy.

Deploy

This OSP falls under the docker requirement, since it can be installed easily using the Docker Compose provided. All docker images are pushed from the docker repository and the containers are initiated within the same docker network to guarantee communication.

3.1.2 TeaStore

Project Description & Architecture

TeaStore is another microservice application [58] featuring 5 distinct services, Fig 3.2. This application was designed to be used as a test case for benchmarks, modelling and resource management research [59]. It uses REST to enable communication among all services and Netflix Ribbon client side as a load balancer. The project offers automation to install the application as well as to test and benchmark it, using LIMBO for HTTP load generator and JMeter.

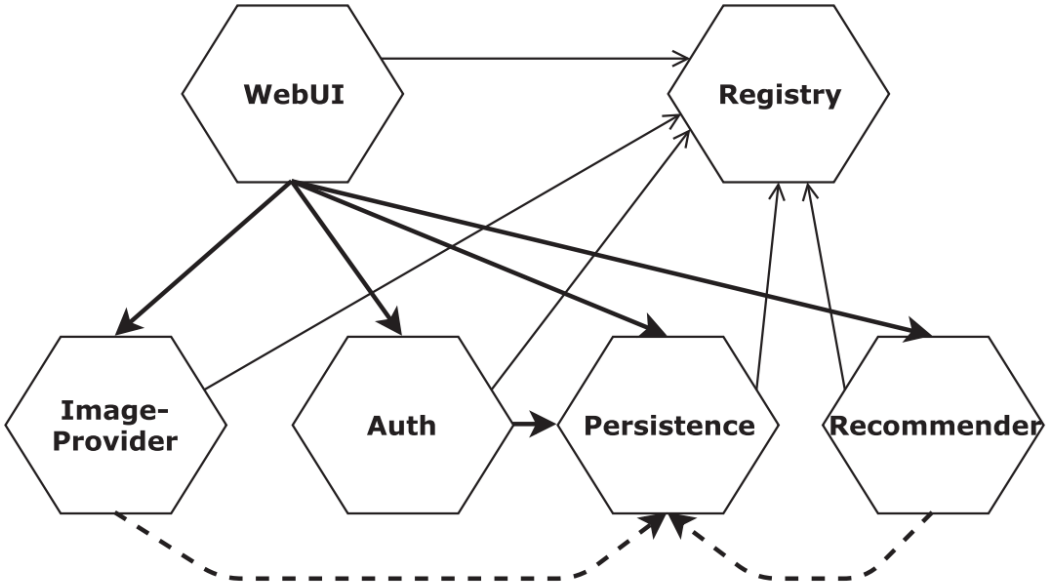

Figure 3.2: TeaStore Architecture [59]

Monitoring

The application offers a health status for the different services, which is helpful when trying to understand if the services are operational. Furthermore, the TeaStore can be deployed using Kieker monitoring, that "provides complementary dynamic analysis capabilities, i.e., monitoring and analyzing a software system's runtime behavior" [60].

Deploy

TeaStore has a good versatility regarding deployment options, as it is available as a single or multiple container. It uses Docker Compose, which meets the deployment requirements, but has more options such as Kubernetes and helm templates.

3.1.3 eShop

Project Description Architecture

eShop is an eCommerce website application with a services-based architecture that uses .NET [61]. It is a sizable application when compared with the ones previously presented. It contains six services: four of them use a SQL database, one uses Redis and all are

registered in a rabbitmq service to be used as an event bus. The application comes with a sample catalog that can be used as a mock/test case as soon as installation is completed.

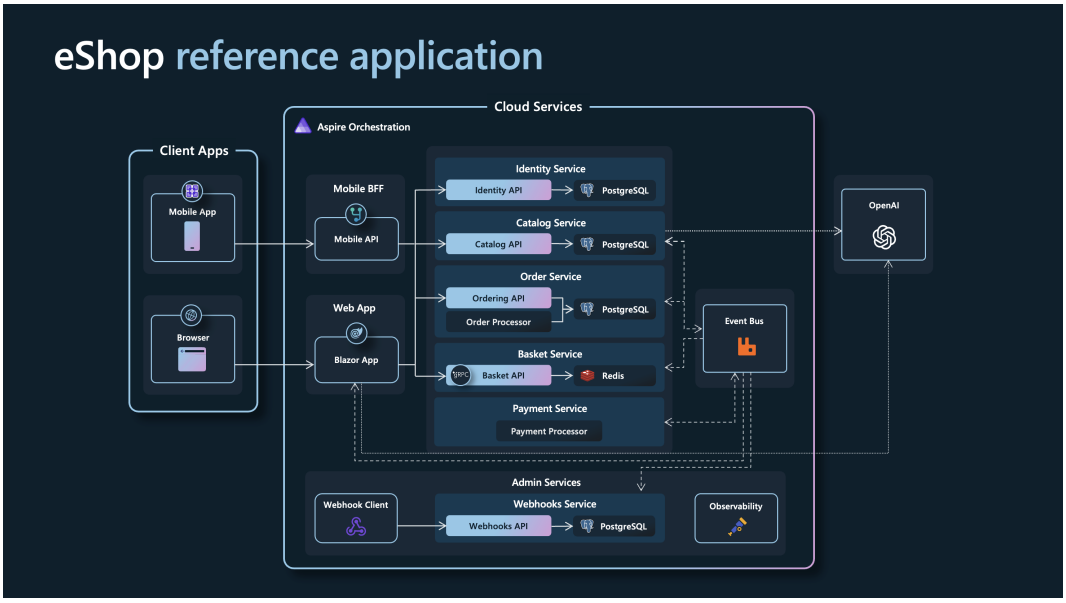

Figure 3.3: eShop High Level Architecture [61]

Monitoring

The monitoring implemented in this application is far more innovative than the one in TeaShop and PiggyMetrics apps. A single access point provides access to the dashboards of every service and to multiple metrics containing also logs, Fig 3.4 and Fig 3.5.

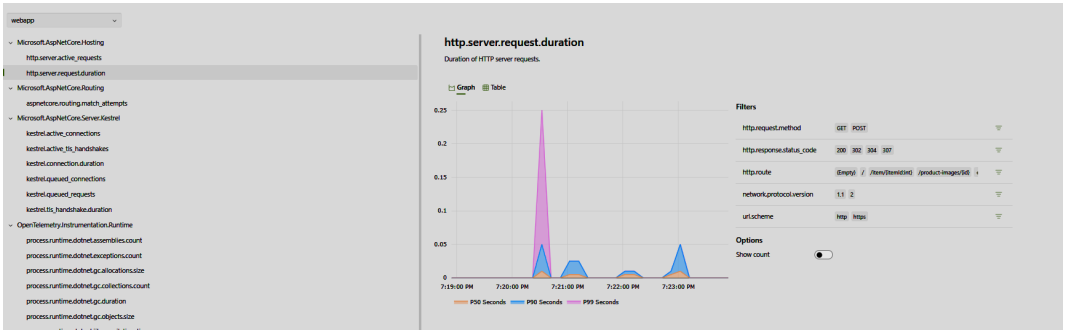

Figure 3.4: eShop Metrics Dashboard

It is also possible to manage the services availability Fig 3.6 and keep track of events traces.

One of the downsides of the existing monitoring is the lack of alarms. Up to now they did not implement a way neither to create alarms based on given metrics or plugins nor to extract the available metrics. Moreover, as a consequence of the absence of alarms, users will not be notified in cases of stress.

Deploy

The application is not prepared for container-style deployment. It is necessary to install all the pre-required softwares. Although there is a single deploy file to initiate the entire application,

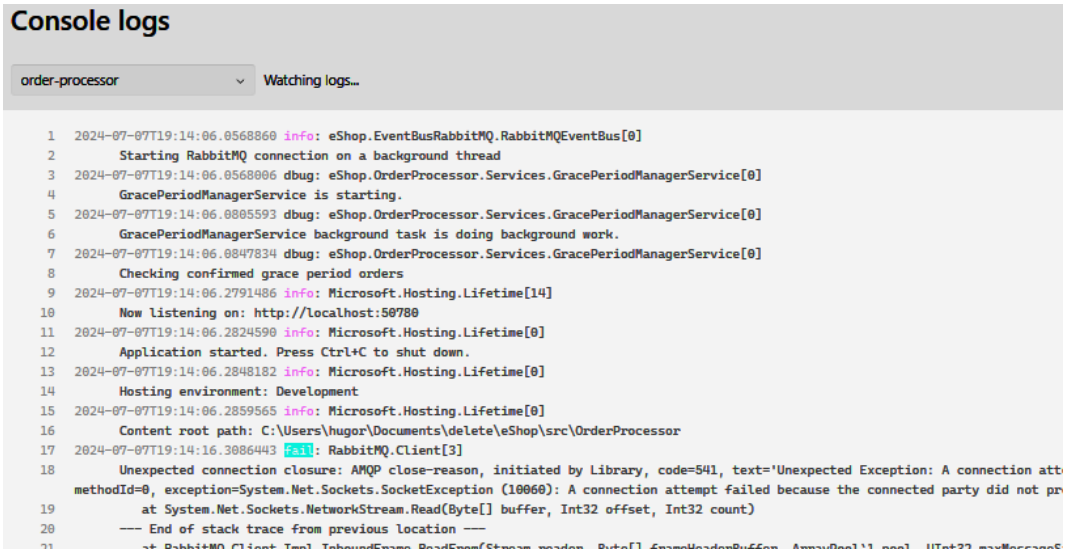

Figure 3.5: eShop Logs Dashboard

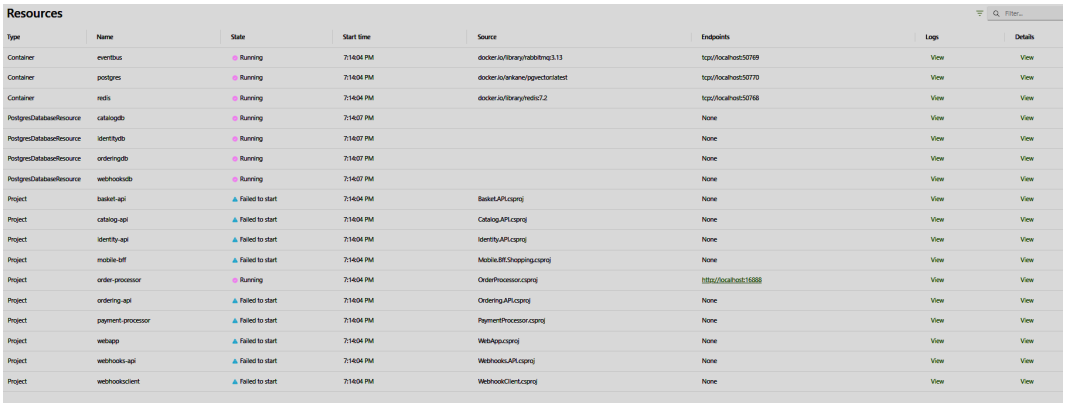

Figure 3.6: eShop Services Availability

it was not possible to initialize the different services individually due to dependencies between them regarding variables. Additionally, the only installation procedure that resulted in a successful deployment was on a Windows machine, which would refrain even more tool availability. Due to all these adversities this application was left behind.

3.1.4 Open-Telemetry Demo

Project Description & Architecture

OpenTelemetry Astronomy Shop is a microservice-based distributed "system intended to illustrate the implementation of OpenTelemetry in a near real-world environment" [62]. OpenTelemetry is a flexible observability framework useful to handle telemetry data such as traces, metrics and logs. Their demo project already includes well-implemented monitoring and logging systems. The application contains twelve main services, listed below [63]:

- Ad Service
- Cart Service
- Checkout Service

- Email Service
- Feature Flag Service
- Frontend
- Load Generator
- Payment Service
- Product Catalog Service
- Quote Service
- Recommendation Service
- Shipping Service

The demo application has a single entry point, the Frontend Proxy, which is the entry point not only for the store but also for the other monitoring tools, such as Grafana and Prometheus.

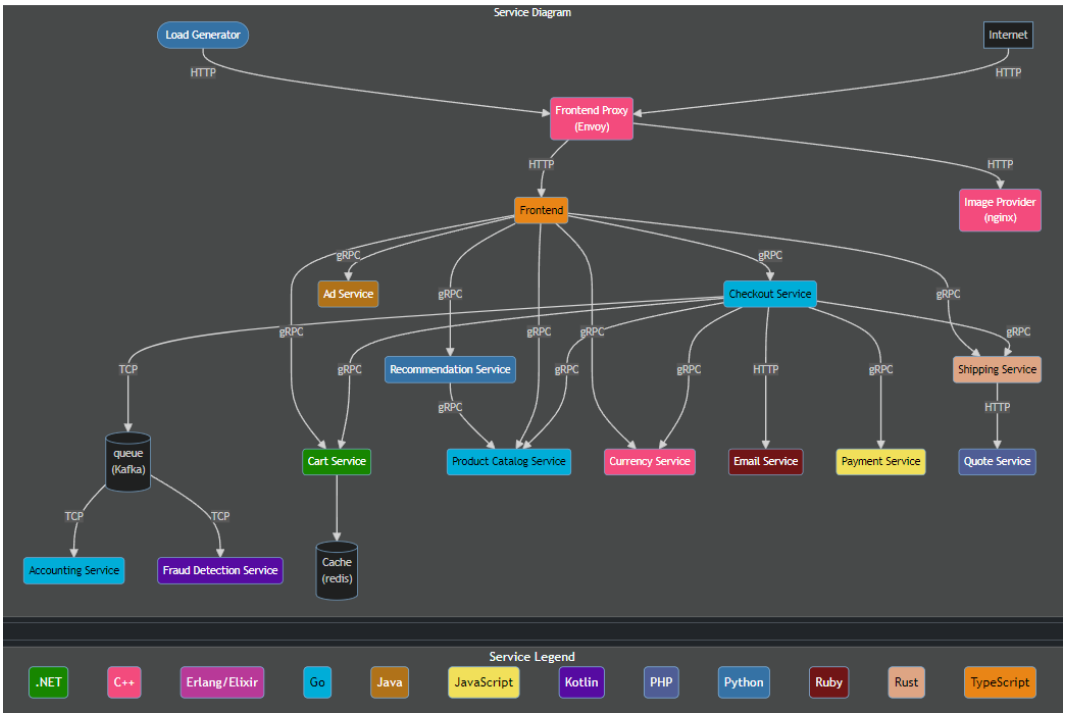

Figure 3.7: Astronomy Shop High-Level Architecture [64]

The application services are written in different languages, such as Java, Go, Python and NodeJS, and include other softwares, such as Postgres, for relational databases, Kafka and Redis. The communication among the services is accomplished by means of gRPC and HTTP protocols. With such a variety of services and technologies, it is possible to test the application in different scenarios.

Monitoring

Being this demo application promoted as "a base for vendors, tooling authors, and others to extend and demonstrate their OpenTelemetry integrations" [65], it is equipped with a

great amount of monitoring tools and pre-configured dashboards. Such tools allow a better understanding of the impact of chaos experiments in the application and better visibility of the behaviour of the application in the different scenarios. The existence of a service to produce load would also facilitate the identification of a possible list of endpoints to send requests to, which is useful to generate more traffic or for testing purposes.

Deploy

The application has been designed for deployment using Docker Compose, and it is possible to deploy the entire application with a single command. Furthermore, if the intention is to use Kubernetes as the project's platform, this can also be accomplished [66].

3.1.5 Selection Criteria

Among the various projects presented, OpenTelemetry Astronomy Shop was chosen for the purposes of this work. The presence of a wide range of services, the diversity of programming languages, the availability of different types of databases and the existing monitoring system were the main factors that contributed to the choice of this project. Although choosing an open-source project can be of help to the community, the time required to adapt and develop the necessary features might affect the possibility of completing the project within the time available.

Therefore, as the focus of this work is to demonstrate how chaos engineering can be incorporated and used in systems, OpenTelemetry project was considered to be a good choice as it already contains key features that allow to continue the development of this work and to carry out diverse experimentation.

3.2 OpenTelemetry Astronomy Shop

This chapter will cover various aspects of the chosen project, OpenTelemetry Astronomy Shop, including details about the monitoring system and the programming language used for each service.

3.2.1 Project Design

The Astronomy Shop uses multiple programming languages, which are presented in Table 3.1 together with the respective services [65].

3.2.2 Monitoring

OpenTelemetry Astronomy Shop's monitoring system consists of three main tools: Grafana, Prometheus and Jaeger. Grafana is a metrics analysis and visualization tool, while Prometheus is an open-source monitoring application that collects metrics from systems and services. Lastly, Jaeger is a distributed tracking tool. Through Grafana it is possible to access the dashboards of each service, where you can visualize the metrics and the history of your system status, such as services latency, CPU and memory usage, etc. Prometheus, responsible for aggregating all metrics, is the datasource of the Grafana tool. It allows to see all metrics generated by the services, including the ones that are not in use by the dashboards. Jaeger, which is also a Grafana datasource, is responsible for collecting the traces of the different services. In addition, it makes it possible to analyse the logs of the services through the use

Table 3.1: Astronomy Shop - Programming Languages

Programming Language	Service
.NET	Cart
C++	Currency
Go	Accounting, Checkout, Product Catalog
Java	Ad
JavaScript	Frontend, Payment
Kotlin	Fraud Detection
PHP	Quote
Python	Recommendation
Ruby	Email
Rust	Shipping

of OpenSearch, which serves as a centralized storage point for the logs. All this information is available in the project documentation [67]. All the dashboards included in this application make use of at least one of the datasources presented.

During the deployment, Grafana will already include the datasources configuration as well as some dashboards.

Grafana Dashboards

There are four predefined dashboards for this project: textitDemo Dashboard, textitOpen-Telemetry Collector, textitOpenTelemetry Collector Dataflow and textitSpanmetrics Demo Dashboad. The Demo Dashboard will be the one used the most for reference and it is composed of three information rows, *Spanmetrics*, *Application Logs* and the *Application Metrics*. The first two rows are filtered by a dashboard variable, which allows to specify a service and shows information about request latency, error rate, requests rate and service logs. The third row shows CPU, rate and memory usage metrics for the recommendation service and a span processor metric for the quote service. The logs visualization, although available, never displayed any of the service logs. This visualization uses OpenSearch as the datasource, and Grafana confirms that the datasource is obtaining data. However, the issue of why the logs were not displayed was not investigated further, as it was not the focus of this project.

OpenTelemetry Collector uses Prometheus as datasource to collect an overall status of the metrics states, such as spans and log records rate, total runtime Sys and Heap memory. This dashboard also allows to visualize the data flow for spans, metrics and log records, Fig 3.8.

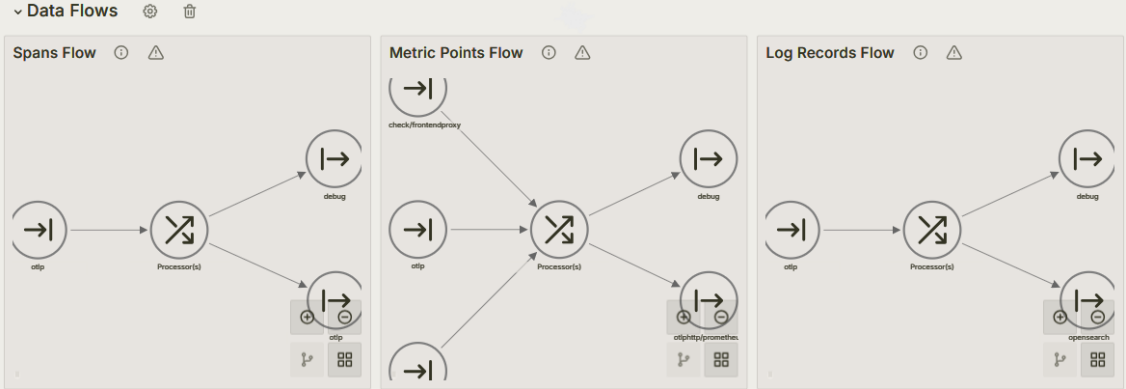

Figure 3.8: Grafana Data Flow Visualization

The third dashboard, OpenTelemetry Collector Data Flow, contains information regarding the system host and metrics exportation. The metrics visualizations are divided in trace, metrics and prometheus. Last but not least, the Spanmetrics Demos Dashboard shows the top seven services per latency, error and mean rate over range. This last could be useful when determining which service has the worst performance or as an alarm trigger.

Chapter 4

Hands-On Chaos Experiments

In this chapter, we delve into the practical aspects of Chaos Engineering by setting up and conducting chaos experiments. Recognising the complexities and potential constraints in applying chaos principles, a specific set of requirements for the target system was defined. These criteria not only streamline the experiment development process but also ensure the relevance and manageability of this approach. With our target system chosen, we articulate the chaos experiment hypotheses. Each hypothesis is systematically broken down into its formulation, implementation, and observed outcomes. This structured approach not only validates our experimental design but also provides actionable insights into the resilience and reliability of the system under test.

4.1 Define a Steady State

The steady state is a big focus for chaos engineers helping the team better define their intervention's scope. This is because of "Chaos Engineering's bias toward verification over validation." [68].

The Grafana dashboards and the Postman collection created based on the loadgen service requests helped us establish the steady-state values of the environment and monitor any changes in those while experimenting. The Postman collection was run using a performance run, which allows to set load type, number of users and the test duration. Due to limitations on the Postman free version the configuration used was,

- Load Profile: Peak
- Virtual Users: 50
- Test Duration: 3m
- Base Load: 11

Both the collection and the postman performance test results are available in a GitHub repository [69] for further analyse being presented here the final results such as in the below Table 4.1.

Table 4.1: Steady State Averages

Average Requests Sent	Throughput	Response Time	Error Rate
23221	122,97 r/s	202,67 ms	0,77

While setting values to define the steady state it was noticed that most of the error rate was related to a single service. Upon stumbling with this event and noticing that in most

of the runs this even was present, the pointed service, checkout-service restarted in all the runs.

Figure 4.1: Checkout Service Restarting

Due to the limited time, and not being the main focus, the root cause for the problem will not be addressed in this paper, although it serves as an example for an early statement, this issue would/should have been identified when running stress tests and would not require chaos engineering experiments before fixing, Chapter 2.2.

Regarding the Grafana dashboards, as previously said, it will be used mainly the *Demo Dashboard* to compare metrics, such as latency and memory allowing us to even compare with some of the Postman results.

4.2 Hypotheses

4.2.1 Hypothesis 1 - CPU Overloading

Hypothesis - The system will react when the CPU usage of a service increases to levels that risk system availability.

This hypothesis aims to ensure that the system can effectively manage CPU overloads by automatically scaling service instances and providing real-time visibility and notifications to maintain performance and availability.

- **Detection**: The system will identify when CPU usage reaches critical limits.
- **Behaviour**: The system will automatically initiate a new instance of the affected service to alleviate CPU load.
- **Monitoring**: Information regarding the overloaded or unhealthy container will be displayed on monitoring dashboards.
- **Error Margin**: The error percentage should remain within a 10% margin of the steady-state error percentage.

This hypothesis allows for the testing of the system's scalability and recovery at the platform level (e.g., Docker), specifically evaluating CPU and Random-Access Memory (RAM) resource management. The application level will also be assessed, focusing on the service itself. In this scenario, minimizing the blast radius is not a priority, since this is not an official environment. Due to this reason it is possible to be more disruptive which is why experiment

number one will be focus on the frontend service. If the frontend is down, no user will be able to access the application or perform any actions.

4.2.2 Hypothesis 2 - Network Issues

Hypothesis - The system can handle network-related issues effectively by presenting alarms on the monitoring dashboard and notifying the on-call team about the problem.

The system can detect network latency or disconnections affecting service communication. It ensures that such issues are promptly reported to the operations team and displayed in real-time on monitoring dashboards.

- **Detection**: The system will detect network latency or disconnections that affect service communication.
- **Notification**: Alerts will be sent to the operations team when significant network issues are detected.
- **Monitoring**: Monitoring dashboards will show real-time network performance and any issues.

Some systems are designed to operate within defined network parameters. This hypothesis will facilitate the acquisition of knowledge regarding the system's behaviour when a service is subjected to network adversities, and will also identify potential deficiencies in the monitoring capabilities that have been implemented.

4.2.3 Hypothesis 3 - Availability

Hypothesis - The system can identify and address missing or unhealthy services.

If no active instance or container of a particular service is available, the system will automatically initiate a new instance to ensure service availability. This proactive approach enhances resilience and availability by reducing downtime through prompt and automated deployment of healthy instances.

- **Detection Monitoring**: The system will monitor the status of each service to identify if no active instance exists. There will be history regarding services availability.
- **Recovery**: Upon detecting the absence of an instance for a specific service, the system will automatically initiate a new instance to restore the service.

This hypothesis focuses on evaluating the system's configuration capabilities, particularly in the context of container management platforms like Docker. Such configurations not only allow for the automatic initiation of new instances when none are available but also enable the setting of default thresholds for required instances. For example, a configuration may specify that Service A must always have at least two healthy instances running.

4.3 Experiments Implementation

This chapter presents the methods used to implement the experiments designed to test the hypotheses defined in the previous chapter. The tools and their respective capabilities have been introduced earlier; therefore, this chapter will focus on how they were utilised

and configured. The environment was fully restored between each experiment to keep all experiments under the default conditions.

4.3.1 Hypothesis 1 - CPU Overloading

To simulate CPU overloading, the ChaosBlade tool was employed. As previously mentioned, this tool allows the simulation of various scenarios, including the one described in the hypothesis.

In addition to the steady-state values set at the beginning, the container stats will also be used to differentiate the system status upon the experiment start, Fig 4.2. A single request demonstrates the expected response time when the system is not under any pressure, Fig 4.3.

NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
frontend	14.22%	119.8MiB / 200MiB	59.91%	3.49GB / 25.8GB	127MB / 58.6MB	25

Figure 4.2: Service Container - Standard Stats

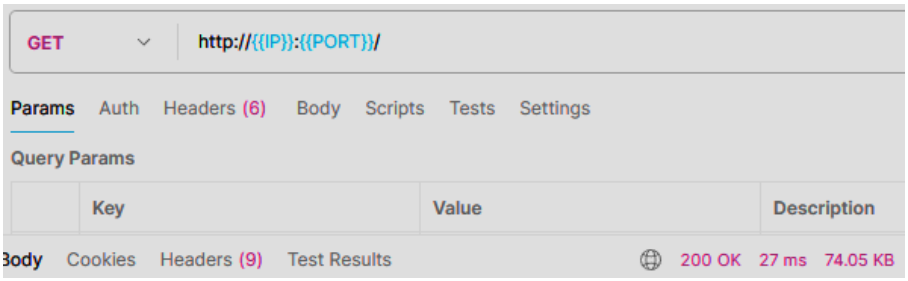

Figure 4.3: Request Response Time - Frontend Service

The definition of other metrics as part of the steady state is essential and will vary depending on the experiment.

ChaosBlade version v1.7.2 was downloaded from the official repository [70]. After downloading and extracting the files to a folder, the contents were copied to the container of the respective service using the docker copy command, better represented in Figure 4.4.

```
1 docker cp chaosblade-1.7.2/ SERVICE:$(docker exec SERVICE pwd)
```

Listing 4.1: Docker Copy

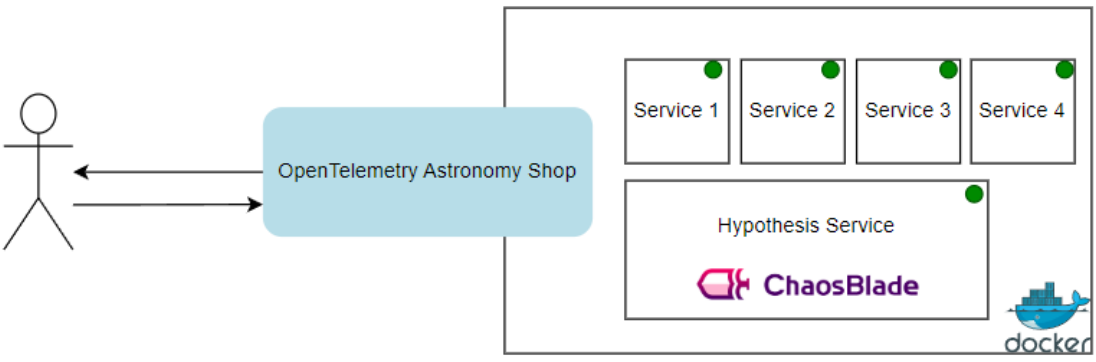

Figure 4.4: Tool Usage Architecture

This approach avoids the need to create a new Docker image based on the original service image, allowing the original image and service to run without modifications. The ChaosBlade CLI was tested by verifying the version in use as in the below example.

```
docker exec -u root -it SERVICE sh
cd chaosblade-1.7.2/
./blade version
```

Listing 4.2: ChaosBlade Exec

Once guaranteed the execution of the chaos blade tool the experience was accomplish using the cpu fullload to induce the CPU overloading.

```
./blade create cpu fullload --timeout 180
```

Listing 4.3: ChaosBlade Experience Execution

This command sets the CPU usage to its maximum for 120 seconds, the duration used for the Postman test. During the simulation, Postman was run under the same conditions as in the steady-state phase to evaluate the service's availability and record the results of the requests. These results were compared with the steady-state values and will be presented in the Results chapter of the respective hypothesis, Chapter 4.4.1.

To observe changes in the container stats, Fig 4.2 and Fig 4.8, it was used the *docker stats SERVICE* command. This allowed for real-time monitoring of the container CPU usage.

4.3.2 Hypothesis 2 - Network Issues

To accomplish this scenario, the tool *toxiproxy* was used.

This hypothesis was validated by using toxics latency and bandwidth first separately and then in the same configuration.

For that it is necessary to install the tool and initiate the toxiproxy server before configuring the toxics. By following the installation process provided on the GitHub page [71] it was pretty straightforward to install the tool, as presented below.

```
git clone https://github.com/Shopify/toxiproxy.git
cd toxiproxy
make build
./toxiproxy-server
```

Listing 4.4: Installation and Toxiproxy Server Deploy

Upon initiating the server it is necessary to create the proxy where the toxics will be inserted, since the command-line is now running the toxiproxy server the below commands were executed in a second terminal.

```
toxiproxy-cli create <proxyName> --listen <addr> --upstream <addr>
```

The upstream when creating the proxy represents the target for our network traffic. Since our system has a single entry point our upstream target will be localhost on port 8080.

The toxics are introduced through toxiproxy CLI that also display all the other available toxics.

```
toxiproxy-cli toxic add <proxyName> --type <toxicType> --toxicName <
    toxicName> --attribute <key=value> --upstream --downstream
```

Listing 4.5: Add Toxic - Model

For this hypothesis all toxics were introduced as upstream toxics, this means that the communication will be tampered when going from the client to server by the proxy, Fig 4.5.

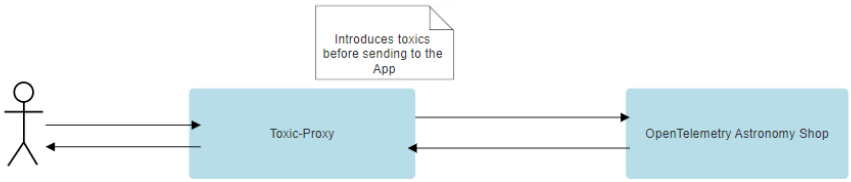

Figure 4.5: Toxic Proxy Communication Diagram

To better present the results the Postman collection used for the steady-state definition will be run to pass over the proxy instead of the application directly, Chapter 4.4.2.

The latency was added under the configuration below.

```
toxiproxy-cli toxic add telemetryProxy --type latency --toxicName
    latToxi --attribute <key=value> --upstream
```

Listing 4.6: Add Toxic - Latency

Upon setting up the adversities the postman collection was change to used the proxy port instead of the OpenTelemetry service port and ran, described results 4.4.

For the second part, the previous toxic was disabled and the bandwidth toxic was enabled.

```
toxiproxy-cli toxic remove telemetryProxy --toxicName latToxi
toxiproxy-cli toxic add telemetryProxy --type bandwidth --toxicName
    bandToxi --attribute <key=value> --upstream
```

Listing 4.7: Add Toxic - Bandwidth

A simple request using a web browser allows us to size the request made when accessing the web page, Fig 4.6.

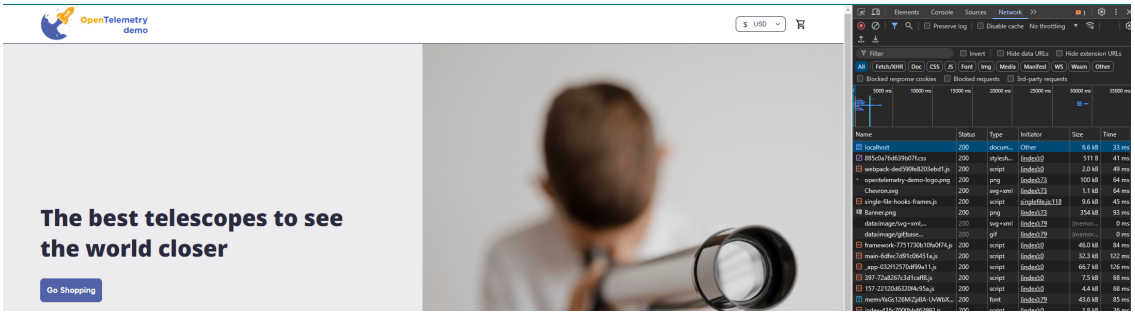

Figure 4.6: Simple Request - Web Page

Due to the small size of the requests the bandwidth was limited to 1 KB/s to present significant adversities. After changing the toxics the collection was run once again to demonstrate differences between an increase of latency and a reduction of available bandwidth, results at 4.5.

In the end, the same collection was run but with both toxics enabled, results 4.6.

4.3.3 Hypothesis 3 - Availability

To test the system's resilience in terms of service's availability, the tool Pumba was used. As previously discussed, Pumba is capable of testing resilience by simulating various scenarios such as killing, stopping, restarting containers, as well as removing and stress testing them.

The tool was downloaded from the official git repository [39], to the systems host. To evaluate the system's ability to handle the complete removal of services, the *kill* capability of Pumba will be used. This action allows for an assessment of how the system behaves when critical services are unexpectedly terminated. Pumba will be executed from the host machine where the containers are running, Fig 4.7, eliminating the need to import or modify the containers themselves.

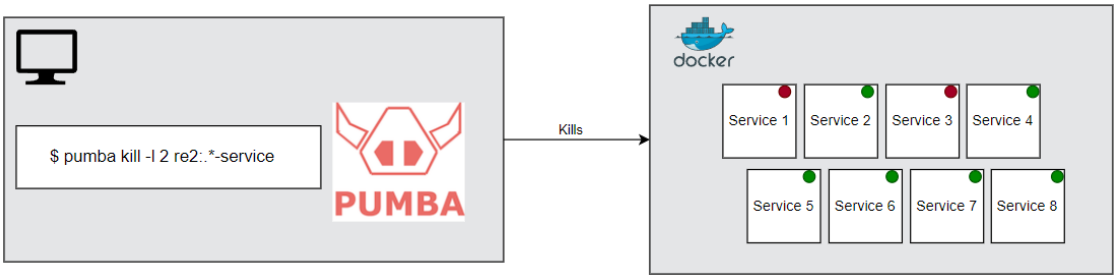

Figure 4.7: Pumba Usage Example

To configure the experiment, it is necessary to specify the names or patterns of the containers to be targeted for removal. The *pumba kill* command provides details on the expected input, as shown in the help command output:

```
$pumba kill --help
    NAME:
        pumba_linux_amd64 kill - kill specified containers

    USAGE:
        pumba_linux_amd64 kill [command options] containers (name, list
    of names, or RE2 regex if prefixed with "re2:")

    DESCRIPTION:
        send termination signal to the main process inside target
    container(s)

    OPTIONS:
        --signal value, -s value  termination signal, that will be sent
    by Pumba to the main process inside target container(s) (default: "
    SIGKILL")
        --limit value, -l value   limit number of container to kill (0:
    kill all matching) (default: 0)
```

Listing 4.8: pumba kill CLI

For this experiment, a list of core services, as defined in section 3.1.4, will be passed to Pumba. The aim is to limit the number of services taken down to two. The following command uses a regular expression to match container names and the *-l* flag to limit the number of containers to be killed:

```
$pumba kill -l 2 re2:.*-service
```

Listing 4.9: Container Kill Command

This command configuration allows for targeting core functionalities while limiting the impact to two services, providing a controlled environment to evaluate the hypothesis under study.

4.4 Results

4.4.1 Hypothesis 1 - CPU Overloading

Experiment Analyses

For this hypothesis, the results will be presented in two phases. The first one will be a direct comparison between the images 4.2 and 4.3, which represent respectively the status of the container upon the usage of the tool and the request response time change but without the heavy work of the Postman performance testing, which will be part of the second phase. This second phase will not only take into consideration the performance test but it will also demonstrate whether the system can withstand/adapt to the conditions.

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
e850b1c38b46	frontend	403.82%	124.3MiB / 200MiB	62.15%	3.46GB / 25.6GB	126MB / 58.6MB	36

Figure 4.8: Service Container Under CPU Overloading - Standard Stats

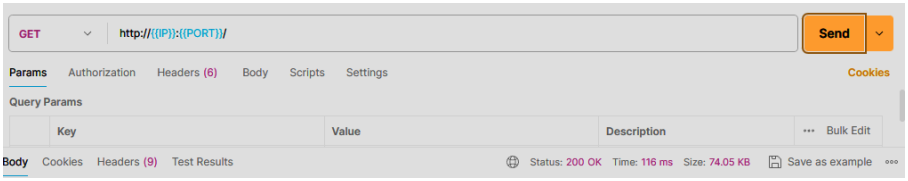

Figure 4.9: Request Response Time Under CPU Overloading - Frontend Service

As predicted when creating the experiment using the Chaosblade tool, the CPU usage of the frontend service container spiked and caused noticeable delays in the response times, Fig 4.9, skyrocketing from 27ms to 116ms, an increase of 329.63%. This change in the response time is even more significant since it is only a single request, meaning that in a production environment with multiple access, this delay could be significantly higher and even more disruptive. On the Grafana dashboards, there is no CPU status panel for javascript services, which means that it is not possible to set explicit alarms regarding the problem presented. However, the team could be notified anyway, since the high CPU usage contributes to an increase in the request-response time, which can trigger a different alarm.

Moreover, Grafana offers a visualization for CPU load of Python services, yet these values pertain solely to the service itself and not to the container, Fig 4.10. Consequently, if the container is compromised due to external factors, alerting the standby team based on the existing metrics will be challenging.

Figure 4.10: CPU Dashboard Issue

The same performance test was run to make a comparison with the steady-state values. To ensure observability over the container and the viability of the experiment, docker stats was used during the test case.

Table 4.2: CPU Overloading Averages

Average Requests Sent	Throughput	Response Time	Error Rate
16063	85,34 r/s	330 ms	1,1

Table 4.2 shows a decrease in performance when compared to the steady state values, Table 4.1. This is particularly concerning given that only one of the services was triggered by an anomaly and raises questions as to how the system would perform if multiple services were under CPU load. This, combined with the lack of information about the health of the containers, could have a significant impact on the overall availability of the solution.

To demonstrate even more the difference in response between the steady state and this experience we can use the request used in Fig 4.3 and Fig 4.9.

Table 4.3: CPU Overloading - GET /

Metric / Env	Steady State	CPU Overloading
Requests Sent	700	494
Throughput	3,71	2,62
Response Time	184	399,67
Error Rate	0,00	0,07

When a single request was presented, the increase in response time could be considered as insignificant; however, when comparing the results of this hypothesis performance test for this specific endpoint with the Steady State, Table 4.3, even though the number of requests was drastically reduced, the response time more than doubled the values of the steady state test and already presented an error rate that gives more emphasis to what was previously stated regarding hidden problems.

Experiment Results

Detection & Monitoring

As demonstrated before, almost any service has a CPU visualization in the existing dashboards and even for those services that do have it, the CPU visualization only regards the

service process and not the container CPU. As a consequence, it becomes more difficult to detect and monitor the CPU resource of the container where the service is executing. The lack of CPU monitoring decreases the team's ability to react upon adversities by depending on other metrics to trigger alarms and notify the team about the degraded systems state. Even by setting alarms for different metrics, such as latency, there is no guarantee that the container CPU state will influence the latency average above the defined threshold.

Behavior
Ideally the system will be able to initiate/deploy new containers when detecting unhealthy ones. This will increase system availability, thanks to the fact that healthy services will always be at disposal and to a possible reduction of downtime.

Error Margin
Based on this experience, we can conclude that the error margin criterion was the only one successfully met. There was an overall performance test improvement of 0.4%, and a 0.07% improvement specifically for the *GET /* request. Although the criterion was met, it is important to clarify that the margin was set as an example. For real-world experiments, this margin should be determined based on additional testing and/or company-specific metrics.

Hypothesis Result
In conclusion, although the margin of error criterion was met, the primary objective of effectively managing CPU congestion through automated scaling was only partially met. The hypothesis was not fully proven due to shortcomings in the detection and monitoring capabilities. For future implementations, enhancing the monitoring system to accurately capture container-level CPU metrics and redefining the criteria for automatic scaling are essential steps towards validating this hypothesis.

4.4.2 Hypothesis 2 - Network Issues

Experiment Analyses

Latency
The Latency experiment was concluded as planned. After introducing the latency toxic, set to 500ms, all requests increased their average response time to over 500ms. The average values when inputting a 500ms latency are,

Table 4.4: Latency Averages

Requests Sent	Throughput	Response Time	Error Rate
10273	54,22 r/s	549 ms	0,37

When comparing it with the steady state results, we can conclude that when in a situation where latency increases to around 500ms, the average number of requests halves. In a real-world scenario, the decrease in answered requests can be damaging for the use case since users will/might start having problems with usability. The increase in the average response time confirms the tool's usability.

Bandwidth
Regarding the bandwidth adversity, it is also noticeable the impact when reduced.

A direct comparison with Fig 4.6 and Fig 4.11 demonstrates how the bandwidth reduction increases the response time and influences the user experience.

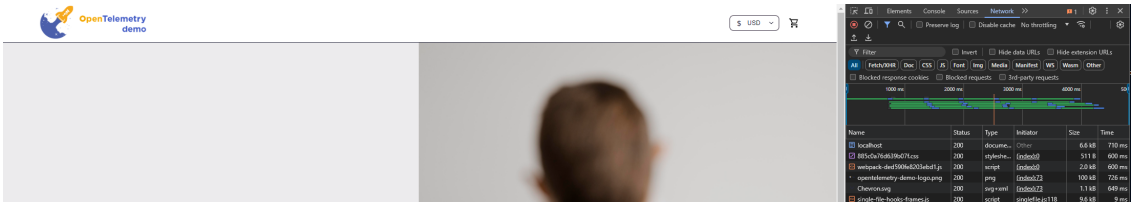

Figure 4.11: Simple Request - Web Page with Reduced Bandwidth

In terms of performance, the same tests were executed, resulting in the following observations, Table 4.5. The average requests, throughput and response time showed a decline in performance compared to the steady-state values, even though not to the same extent as the latency. The error rate demonstrated a notable reduction. However, given the limitations of the available information, it is not possible to draw any definitive conclusions regarding the underlying cause.

Table 4.5: Bandwidth Averages

Requests Sent	Throughput	Response Time	Error Rate
14576	77,12 r/s	360 ms	0,02

Full Toxic

Performing the full network experiment and joining both latency and bandwidth restrictions, we get the scenario in Fig 4.12.

```
hribeiro@hribeiro-server:~$ toxiproxy-cli i telemetryProxy
Name: telemetryProxy    Listen: 127.0.0.1:9090  Upstream: localhost:8080
======================================================================
Upstream toxics:
bandToxi:       type=bandwidth  stream=upstream toxicity=1.00   attributes=[    rate=1  ]
latToxi:        type=latency    stream=upstream toxicity=1.00   attributes=[    jitter=0        latency=500     ]
```

Figure 4.12: Proxy Configuration - Full Toxic

Using the data previously presented, Table 4.4 and Table 4.5, and comparing with the steady-state values, Table 4.1, this experiment led to the results, Table 4.6

Table 4.6: Network Toxics

Metric / Env	Steady State	Latency (r/s)	Bandwidth (ms)	Full Toxics
Requests Sent	23221	10273	14576	10102
Throughput	122,97	54,22	77,12	53,49
Response Time	202,67	549	360	559,33
Error Rate	0,77	0,37	0.02	0,00

The chaos experiment demonstrates that the introduction of network toxins, such as latency, bandwidth, and/or a combination of both, has a significant impact on the performance of the OpenTelemetry Demo Astronomy Shop. The size of the request can have a significant impact on performance metrics, particularly in the presence of high latency and other network toxins. The comparable performance outcomes between the Latency and Full Toxics environments indicate that latency is a principal element of the Full Toxics scenario. This emphasises the necessity of incorporating request size into performance testing and

optimisation, particularly in systems that are anticipated to operate in challenging network environments.

Experiment Results

Detection & Monitoring
As the toxins were artificially added, that is, both latency and bandwidth were introduced into the proxy side, and did not directly affect the service network, no traces were left in the monitoring tools. However, during the steady state phase, it was possible to conclude that the existing dashboards can indeed correctly display relevant information, such as latency and percentiles.

Notification
As mentioned earlier, Grafana can send alarms based on certain events. Although there are no alerts configured for this use case, it would be easy to do so since Grafana has a wide range of channels integrations such as Teams, Discord and PagerDuty. The possibility of defining values when your system is under too much stress will depend on the configuration of the system and on the resources available. Further testing should be done to properly identify your system's stress values.

Hypothesis Result
In summary, while not all criteria elements were fully implemented during the trial, the necessary components are either already in place or can be obtained with straightforward configurations, particularly in terms of notifications. The system's detection and monitoring capabilities are also available, providing visibility into network latency. With these tools, the system has the potential to effectively manage network-related issues once properly configured.

4.4.3 Hypothesis 3 - Availability

Experiment Analyses

During the steady-state phase, an issue was identified with a service that would restart upon an error. This was deemed insufficient, given that not all services would react in the same way to different types of errors. Therefore, concrete proof of how the system manages existing and non-existing services is paramount.

Since not a single Grafana dashboard displays the service availability, it was necessary to use the docker stats command, presented below, as previously used, as a proof of the state of each service container. Having found this deficiency of Grafana, the monitoring goal set for this experience has been proven faulty since no alerts can be configured to notify the team in case one of the services goes down.

```
1 $watch docker ps --format "{{.ID}}//{{.Names}}//{{.Status}}"
```

Listing 4.10: Container Observability

When executing the command, the services ad-service and currency-service, respectively container id *98aa6971f6c* and *3fb45b8b97bb*, were removed from the list of running containers shown in Fig 4.13, where the left side represents the system state after the command and the right side the previous system state.

Figure 4.13: Services Availability

In this scenario, after executing the Postman test run, we can prove all the statements presented at the beginning of this section.

- The system cannot initiate new containers for missing services
- Some services when faced with an internal error might restart, maintaining the existing container
- The system becomes unusable if some of the services are missing, presenting an error rate of over 30%, Table 4.7

Table 4.7: Availability Averages

Requests Sent	Throughput	Response Time	Error Rate
29329	154.91 r/s	138 ms	30,99

The results of the performance test indicate that the system is not adequately prepared to handle the loss of these two services. Ideally, each microservice should be able to function normally in the absence of another to ensure data integrity. Additionally, robust error handling is essential for a resilient software design. The Postman performance test also revealed a big discrepancy between the different endpoints: eight endpoints showed a 0% error rate, while nine endpoints presented a 100% error rate. These statistics indicate that the system should be considered faulty, as it will have a substantial impact on the user experience and data integrity is not guarantee.

Experiment Results

Detection & Monitoring

As pointed out before, Grafana's dashboards do not contain any information regarding the container's healthy state. Using docker stats is helpful when it comes to verify all containers state, but without a way to retrieve that information and analyse it, it would not be possible to efficiently use docker stats as a reliable piece of the system monitoring.

Recovery

As per docker, since the service's deployment uses standard Docker Compose, it is not possible to automatically recreate a down/killed instance without a third party, which can be either a script or a different tool. Using Docker an option would be to switch to Docker Swarm.

Docker Swarm is a native clustering and orchestration tool integrated into docker, allowing you to manage a cluster of docker nodes as a single virtual system. It enables container orchestration, where services are defined in a declarative manner, ensuring a desired state across the cluster [72].

Docker Swarm offers several key features. First, its cluster management capability turns multiple Docker engines into a unified, virtual Docker Engine. It also has a declarative service model, allowing you to specify desired states for services. Furthermore, Docker Swarm includes built-in service discovery and load balancing, ensuring efficient resource distribution. The multi-host networking feature facilitates container communication across different hosts, and the rolling updates feature allows services to carry out smooth updates without downtime.

One of its most particular features is the scaling, which is missing when using simple Docker Compose. It allows you to easily scale services by increasing or decreasing the number of service replicas with simple commands. The platform ensures automatic task distribution across available worker nodes to balance the load effectively. Moreover, Docker Swarm supports self-healing, automatically reassigning tasks from failed nodes to healthy ones, which ensures high availability.

Other platforms, like Kubernetes [73], also have scaling capabilities incorporated allowing the initiation of new pods when no healthy ones are available. They are also able to scale horizontally, increasing the replication of the same services.

Hypothesis Result
In conclusion, the hypothesis is not supported due to the absence of both monitoring capabilities to detect missing services and the necessary scaling mechanisms within the current Docker Compose setup.

Chapter 5

Conclusion

Chaos Engineering is a relatively new concept that has yet to be widely adopted in the field, in particular by smaller and mid-size companies. This project aimed to explore and spread knowledge on several key aspects of Chaos Engineering (CE). The fundamental concept of CE (**RQ1**) is addressed, its current adoption rate and challenges (**RQ2**) are examined, and guidance on how to effectively implement CE (**RQ3**) is provided.

In section 2.2 we have established that Chaos Engineering is a disciplined approach to identifying vulnerabilities within complex systems through hypothesis-driven experimentation. This practice allows teams to observe and analyse the system's behaviour under unexpected conditions, in order to prepare it for real-world scenarios. While concerns about potential downtime of production environments may contribute to low adoption rates, as addressed in **RQ2**, section 1.3.1, the lack of awareness and understanding of CE is also a factor that plays a role in the low levels of adoption. Being able to identify the system's maturity level is important as well, since missing capabilities like monitoring will turn all CE practices useless due to missing data needed to analyse the findings.

Setting some guidelines on what can be considered part of the system's maturity, 2.1, was one of the key findings of this project, since without it CE is merely an aesthetically pleasing addition in to your system lifecycle. CE heavily relies on monitoring and traceability tools to allow concrete analyses. For a successful CE implementation, **RQ3**, a strategic plan to guarantee the safety of the system is also needed. CMM is a framework presented in section 1.1 that can guide organizations during the adoption process, outlining the different stages. Companies can then slowly create/gain trust in their experiments while getting some of the benefits related to CE. Chapter 4 demonstrates, by means of a practical implementation of CE, the process of generating hypotheses and illustrates how to configure certain tools to simulate stress conditions.

After carrying our various experiments, I am confident that the chosen platforms, like docker or Kubernetes, will have a major impact on the open-source tools available for the experimentations. Finding tools compatible with Docker, despite its popularity for deploying microservices, proved more challenging than initially anticipated.

These key findings also highlighted some limitations of the project. Since the project relied on personal resources and open-source materials, the available monitoring options were limited, particularly given the microservice architecture and the Docker-based deployment. This constraint significantly reduced the range of software tools that could be used, as discussed in section 3.1. In case of a project that does not already include monitoring or Docker-based deployment, considerable additional time is required to implement these features, time that was not available within our project's constraints.

Additionally, hardware and network limitations presented challenges in ensuring consistent conditions during experimentation, which is crucial for maintaining a rigorous scientific methodology.

Even with these constraints, this project was able to demonstrate some of the principles of CE and gave simple examples on how to start it. The project led to a deeper learning on how to experiment and elaborate hypotheses, analyse the results and look for solutions and other break points.

Future research could focus on expanding the scope of CE experiments to include a broader range of system components, such as databases and service communications. This would allow for the exploration of how CE principles can be applied to different areas of a system's architecture, providing a more comprehensive understanding of its potential benefits.

Additionally, a detailed investigation of fault injection techniques enhanced with practical examples would be a great way to further contribute to the knowledge based on CE. Another promising option for future work would be to apply CE within a mature application and system, allowing for real-world case study of the advantages of CE integration.

In order to confirm what companies do know about CE and how they feel about embracing it, it would be beneficial to conduct interviews with professionals from different roles, such as project managers, developers and DevOps engineers. This would provide insights into their perception and understanding of CE.

In conclusion, this project, while challenging, has been deeply enlightening. Hands-on work on a topic as dynamic and promising as Chaos Engineering has provided valuable learning opportunities. While CE has the potential to significantly improve system resilience and reduce long-term operational challenges, its adoption may be limited by factors such as tight timelines, limited funding and lack of awareness.

However, the responsibility for pushing these innovative practices forward lies on us, the younger generation. By promoting new methods and technologies, we can drive the continued progression of systems and applications, ensuring they are better equipped to withstand the complexities of the modern digital landscape.

Bibliography

[1] Paulo Jorge Ricardo Andrade. "Engenharia de Resiliência". eng. Accepted: 2019-06-28T13:42:09Z. MA thesis. 2018. url: https://recipp.ipp.pt/handle/10400.22/14148 (visited on 12/28/2023).

[2] *9. Chaos Maturity Model - Chaos Engineering [Book]*. en. ISBN: 9781491953068. url: https://www.oreilly.com/library/view/chaos-engineering/9781491988459/ch09.html (visited on 12/28/2023).

[3] *1. Resilience in Software and Systems*. en. isbn: 978-1-09-811381-0. url: https://learning.oreilly.com/library/view/security-chaos-engineering/9781098113810/ch01.html#:~:text=the%20resilience%20of%20what%2C%20to%20what%2C%20and%20for%20whom (visited on 08/04/2024).

[4] Metodi Hadji-Janev, Mitko Bogdanoski, and Agnieszka Piekarz, eds. *Handbook of Research on Civil Society and National Security in the Era of Cyber Warfare:* Advances in Digital Crime, Forensics, and Cyber Terrorism. IGI Global, 2016. isbn: 978-1-4666-8793-6 978-1-4666-8794-3. doi: 10.4018/978-1-4666-8793-6. url: http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/978-1-4666-8793-6 (visited on 12/26/2023).

[5] *What Is Fault Tolerance? | Creating a Fault-tolerant System*. en. url: https://www.fortinet.com/resources/cyberglossary/fault-tolerance (visited on 01/04/2024).

[6] *The Five Pillars of Resilience Engineering | InformationWeek*. en. url: https://www.informationweek.com/cyber-resilience/the-five-pillars-of-resilience-engineering (visited on 12/26/2023).

[7] Raimund Laqua. *The Four Cornerstones of Resilience*. en. Mar. 2020. url: https://www.leancompliance.ca/post/the-four-cornerstones-of-resilience (visited on 12/26/2023).

[8] Riana Steen and Terje Aven. "A risk perspective suitable for resilience engineering". en. In: *Safety Science* 49.2 (Feb. 2011), pp. 292–297. issn: 09257535. doi: 10.1016/j.ssci.2010.09.003. url: https://linkinghub.elsevier.com/retrieve/pii/S0925753510002237 (visited on 11/11/2023).

[9] "Chaos Engineering - Unknown". en. In: *3. Overview of Principles*. isbn: 978-1-4920-4385-0. url: https://learning.oreilly.com/library/view/chaos-engineering/9781492043850/ch03.html#:~:text=It%20is%20an%20exploration%20of%20the%20unknown (visited on 12/03/2023).

[10] "Experimentation". en. In: *3. Overview of Principles*. isbn: 978-1-4920-4385-0. url: https://learning.oreilly.com/library/view/chaos-engineering/9781492043850/ch03.html#:~:text=Experimentation%20either%20builds,our%20own%20system (visited on 12/03/2023).

[11] *PRINCIPLES OF CHAOS ENGINEERING - Principles of chaos engineering*. url: https://principlesofchaos.org/ (visited on 12/03/2023).

[12] "CE Production". en. In: *Chaos Engineering Distilled*. isbn: 978-1-4920-5099-5. url: https://learning.oreilly.com/library/view/learning-chaos-engineering/

9781492050995/ch01.html#:-:text=It%E2%80%99s%20only%20in,avoid%20an%20outage (visited on 12/01/2023).

[13] "Dark Debt". en. In: *Chaos Engineering Distilled*. isbn: 978-1-4920-5099-5. url: https://learning.oreilly.com/library/view/learning-chaos-engineering/9781492050995/ch01.html#:-:text=The%20STELLA%20Report,reveals%20its%20presence.%E2%80%9D (visited on 12/01/2023).

[14] Chrissy Kidd. *What is 'Dark Debt'? How to Eliminate the Dark Debt in your IT Organization*. en-US. url: https://www.bmc.com/blogs/dark-debt/ (visited on 08/07/2024).

[15] "CE helping hand". en. In: *Chaos Engineering Distilled*. isbn: 978-1-4920-5099-5. url: https://learning.oreilly.com/library/view/learning-chaos-engineering/9781492050995/ch01.html#:-:text=Chaos%20engineering%20provides,the%20real%20world. (visited on 12/01/2023).

[16] "Known Weakness". en. In: *Chaos Engineering Distilled*. isbn: 978-1-4920-5099-5. url: https://learning.oreilly.com/library/view/learning-chaos-engineering/9781492050995/ch01.html#:-:text=%E2%80%9CIf%20I%20know,answer%20is%20no (visited on 12/01/2023).

[17] "Blast Radius". en. In: *Chaos Engineering Distilled*. isbn: 978-1-4920-5099-5. url: https://learning.oreilly.com/library/view/learning-chaos-engineering/9781492050995/ch01.html#:-:text=When%20considering%20whether,its%20Blast%20Radius (visited on 12/01/2023).

[18] Hugo Jernberg, Per Runeson, and Emelie Engström. "Getting Started with Chaos Engineering - design of an implementation framework in practice". en. In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. Bari Italy: ACM, Oct. 2020, pp. 1–10. isbn: 978-1-4503-7580-1. doi: 10.1145/3382494.3421464. url: https://dl.acm.org/doi/10.1145/3382494.3421464 (visited on 12/01/2023).

[19] "CE Areas". en. In: *Chaos Engineering from Beginning to End*. isbn: 978-1-4920-5099-5. url: https://learning.oreilly.com/library/view/learning-chaos-engineering/9781492050995/ch06.html#:-:text=People%2C%20practices%2C%20and,Infrastructure (visited on 12/02/2023).

[20] *Introduction: Birth of Chaos - Chaos Engineering [Book]*. en. ISBN: 9781492043867. url: https://learning.oreilly.com/library/view/chaos-engineering/9781492043850/preface02.html#:-:text=%E2%8%9CShould%20we%20do,doing%20Chaos%20Engineering%3F%E2%80%9D (visited on 12/28/2023).

[21] *Chaos Engineering*. en. url: https://www.gremlin.com/chaos-engineering/ (visited on 12/26/2023).

[22] *Chaos Maturity Model*. en. isbn: 978-1-4919-8845-9. url: https://learning.oreilly.com/library/view/chaos-engineering/9781491988459/ch09.html (visited on 01/03/2024).

[23] *Termination behavior - Chaos Monkey*. url: https://netflix.github.io/chaosmonkey/Termination-behavior/ (visited on 12/25/2023).

[24] *The Evolution of Chaos Engineering | Netflix Video at AWS re:Invent 2022*. en-US. url: https://aws.amazon.com/solutions/case-studies/netflix-reinvent-2022-evolution-of-chaos-engineering/ (visited on 12/28/2023).

[25] Netflix Technology Blog. *ChAP: Chaos Automation Platform*. en. Feb. 2018. url: https://netflixtechblog.com/chap-chaos-automation-platform-53e6d528371f (visited on 12/28/2023).

[26] Tiago Boldt Sousa et al. "Engineering Software for the Cloud: External Monitoring and Failure Injection". In: *Proceedings of the 23rd European Conference on Pattern Languages of Programs*. EuroPLoP '18. New York, NY, USA: Association for Computing Machinery, July 2018, pp. 1–8. isbn: 978-1-4503-6387-7. doi: 10.1145/3282308.3282316. url: https://doi.org/10.1145/3282308.3282316 (visited on 01/04/2024).

[27] *What is fault injection?* en. Feb. 2021. url: https://www.gremlin.com/blog/what-is-fault-injection/ (visited on 01/04/2024).

[28] *A fault injection platform for learning AIOps models | Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. url: https://dl.acm.org/doi/abs/10.1145/3551349.3559503 (visited on 01/04/2024).

[29] *Facebook Turned Off Entire Data Center to Test Resiliency | Data Center Knowledge | News and analysis for the data center industry*. url: https://www.datacenterknowledge.com/archives/2014/09/15/facebook-turned-off-entire-data-center-to-test-resiliency#close-modal (visited on 12/26/2023).

[30] Microsoft Azure. *Inside Azure Search: Chaos Engineering*. en-US. July 2015. url: https://azure.microsoft.com/en-us/blog/inside-azure-search-chaos-engineering/ (visited on 12/12/2023).

[31] Julie Bort. *Meet Kripa Krishnan, Google's queen of chaos*. en-US. url: https://www.businessinsider.com/profile-of-google-disaster-recovery-testing-boss-kripa-krishnan-2016-8 (visited on 12/26/2023).

[32] *Resilience Testing Tools - AWS Fault Injection Service - AWS*. en-US. url: https://aws.amazon.com/fis/ (visited on 07/07/2024).

[33] *Reliability Testing & Chaos Engineering | Gremlin*. en. url: https://www.gremlin.com/ (visited on 07/07/2024).

[34] *Home - Chaos Monkey*. url: https://netflix.github.io/chaosmonkey/ (visited on 12/25/2023).

[35] *Spinnaker*. en. url: https://spinnaker.io/ (visited on 12/24/2023).

[36] *How to deploy - Chaos Monkey*. url: https://netflix.github.io/chaosmonkey/How-to-deploy/#prerequisites (visited on 12/25/2023).

[37] *Termination behavior - Chaos Monkey*. url: https://netflix.github.io/chaosmonkey/Termination-behavior/ (visited on 12/25/2023).

[38] *The Chaos Monkey Army*. en. url: https://github.com/Netflix/SimianArmy/wiki/The-Chaos-Monkey-Army (visited on 12/25/2023).

[39] Alexei Ledenev. *Pumba: chaos testing tool for Docker*. original-date: 2016-03-22T14:20:27Z. Dec. 2023. url: https://github.com/alexei-led/pumba (visited on 12/27/2023).

[40] *Chaosblade: An Easy to Use and Powerful Chaos Engineering Toolkit*. original-date: 2019-03-12T11:02:05Z. Dec. 2023. url: https://github.com/chaosblade-io/chaosblade (visited on 12/28/2023).

[41] Codecentric. *Chaos Monkey for Spring Boot*. en-US. url: https://codecentric.github.io/chaos-monkey-spring-boot/ (visited on 12/26/2023).

[42] *Chaos Monkey for Spring Boot Reference Guide*. url: https://codecentric.github.io/chaos-monkey-spring-boot/latest/ (visited on 12/26/2023).

[43] *Chaos Toolkit - Chaos Engineering for Developers*. original-date: 2017-09-24T15:49:05Z. Dec. 2023. url: https://github.com/chaostoolkit/chaostoolkit (visited on 12/27/2023).

[44] *Chaos Toolkit - The chaos engineering toolkit for developers*. url: https://chaostoolkit.org/ (visited on 12/27/2023).

[45] *Overview - Chaos Toolkit - The chaos engineering toolkit for developers*. url: https://chaostoolkit.org/drivers/overview/ (visited on 12/27/2023).
[46] *Spring - Chaos Toolkit - The chaos engineering toolkit for developers*. url: https://chaostoolkit.org/drivers/spring/ (visited on 12/27/2023).
[47] *Install - Chaos Toolkit - The chaos engineering toolkit for developers*. url: https://chaostoolkit.org/reference/usage/install/ (visited on 12/28/2023).
[48] *ToxiProxy - Chaos Toolkit - The chaos engineering toolkit for developers*. url: https://chaostoolkit.org/drivers/toxiproxy/ (visited on 06/22/2024).
[49] *Shopify/toxiproxy-readme*. original-date: 2014-09-04T13:56:38Z. June 2024. url: https://github.com/Shopify/toxiproxy?tab=readme-ov-file#toxiproxy (visited on 06/22/2024).
[50] *Shopify/toxiproxy*. original-date: 2014-09-04T13:56:38Z. June 2024. url: https://github.com/Shopify/toxiproxy (visited on 06/22/2024).
[51] *Shopify/toxiproxy-toxics*. original-date: 2014-09-04T13:56:38Z. June 2024. url: https://github.com/Shopify/toxiproxy?tab=readme-ov-file#toxics (visited on 06/22/2024).
[52] *toxiproxy/CREATING_TOXICS.md at main · Shopify/toxiproxy*. en. url: https://github.com/Shopify/toxiproxy/blob/main/CREATING_TOXICS.md (visited on 06/22/2024).
[53] Alexander Lukyanchikov. *sqshq/piggymetrics*. original-date: 2015-03-29T17:56:31Z. July 2024. url: https://github.com/sqshq/piggymetrics (visited on 07/07/2024).
[54] Alexander Lukyanchikov. *sqshq/piggymetrics-infrastructure*. original-date: 2015-03-29T17:56:31Z. July 2024. url: https://github.com/sqshq/piggymetrics?tab=readme-ov-file#infrastructure (visited on 07/07/2024).
[55] *Configuring Hystrix Dashboard in your Spring Boot application - Masterspringboot*. url: http://www.masterspringboot.com/cloud/netflix/configuring-hystrix-dashboard-in-your-spring-boot-application/ (visited on 07/07/2024).
[56] Netflix Technology Blog. *Hystrix Dashboard + Turbine Stream Aggregator*. en. Apr. 2017. url: https://netflixtechblog.com/hystrix-dashboard-turbine-stream-aggregator-60985a2e51df (visited on 07/07/2024).
[57] *Home*. en. url: https://github.com/Netflix/Hystrix/wiki/Home (visited on 06/15/2024).
[58] *DescartesResearch/TeaStore*. original-date: 2017-08-18T06:22:29Z. June 2024. url: https://github.com/DescartesResearch/TeaStore (visited on 06/15/2024).
[59] Jóakim von Kistowski et al. "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research". In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. ISSN: 2375-0227. Sept. 2018, pp. 223–236. doi: 10.1109/MASCOTS.2018.00030. url: https://ieeexplore.ieee.org/document/8526888 (visited on 07/07/2024).
[60] *Kieker | Application Performance Monitoring and Dynamic Software Analysis*. url: https://kieker-monitoring.net/ (visited on 08/14/2024).
[61] *dotnet/eShop*. original-date: 2023-10-18T23:00:45Z. June 2024. url: https://github.com/dotnet/eShop (visited on 06/15/2024).
[62] *open-telemetry/opentelemetry-demo*. original-date: 2022-04-26T19:47:29Z. July 2024. url: https://github.com/open-telemetry/opentelemetry-demo (visited on 07/07/2024).
[63] *Services*. en. url: https://opentelemetry.io/docs/demo/services/ (visited on 07/07/2024).

[64] *Demo Architecture*. en. Section: docs. url: https://opentelemetry.io/docs/demo/architecture/ (visited on 07/07/2024).
[65] *OpenTelemetry Demos Services Languages*. en. url: https://opentelemetry.io/docs/demo/#language-feature-reference (visited on 07/07/2024).
[66] *open-telemetry/opentelemetry-demo-deployment*. original-date: 2022-04-26T19:47:29Z. July 2024. url: https://github.com/open-telemetry/opentelemetry-demo?tab=readme-ov-file#quick-start (visited on 07/07/2024).
[67] *Collector Data Flow Dashboard*. en. Section: docs. url: https://opentelemetry.io/docs/demo/collector-data-flow-dashboard/ (visited on 06/15/2024).
[68] *3. Overview of Principles*. en. isbn: 978-1-4920-4385-0. url: https://learning.oreilly.com/library/view/chaos-engineering/9781492043850/ch03.html (visited on 08/04/2024).
[69] Hug0Ribeir0. *Hug0Ribeir0/ChaosEngineering-Bring-Order-to-Chaos*. original-date: 2024-07-18T13:18:41Z. July 2024. url: https://github.com/Hug0Ribeir0/ChaosEngineering-Bring-Order-to-Chaos (visited on 08/05/2024).
[70] *chaosblade-io/chaosblade*. original-date: 2019-03-12T11:02:05Z. Aug. 2024. url: https://github.com/chaosblade-io/chaosblade/tags (visited on 08/04/2024).
[71] *Shopify/toxiproxy-installing*. original-date: 2014-09-04T13:56:38Z. June 2024. url: https://github.com/Shopify/toxiproxy?tab=readme-ov-file#1-installing-toxiproxy (visited on 06/22/2024).
[72] *Swarm mode overview*. en. 100. url: https://docs.docker.com/engine/swarm/ (visited on 07/05/2024).
[73] *Production-Grade Container Orchestration*. en. url: https://kubernetes.io/ (visited on 07/05/2024).