Gonçalo Claro Baptista

FAILURE INJECTION IN MICROSERVICE APPLICATIONS

1 2 9 0 UNIVERSIDADE D COIMBRA

1 2 9 0

UNIVERSIDADE D
COIMBRA

Gonçalo Claro Baptista

FAILURE INJECTION IN MICROSERVICE APPLICATIONS

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

October 2021

Faculty of Sciences and Technology

Department of Informatics Engineering

Failure Injection in Microservice Applications

Gonçalo Claro Baptista

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

October 2021

UNIVERSIDADE D
COIMBRA

This page is intentionally left blank.

This work was carried out under the project P2020 - 31/SI/2017: AESOP — Autonomic Service Operation, supported by Portugal 2020 and UE-FEDER. This work was also partially supported by national funds through the FCT - Foundation for Science and Technology, I.P., within the scope of the project CISUC - UID/CEC/00326/2020 and by the European Social Fund, through the Regional Operational Program Centro 2020

This page is intentionally left blank.

Abstract

To achieve dependability, system designers often resort to fault-tolerance mechanisms. The evaluation of these mechanisms requires the observation of failures, which typically are relatively uncommon. To increase the failure rate, practitioners employ fault injection techniques, resulting in an increased occurrence of failures and allowing the evaluation of the systems dependability properties. While various fault injection tools exist for this end, they are usually limited in scope, applicability and in their configuration abilities for microservice applications.

We propose *Defektor*, a generalist and extensible tool capable of controlling a fault injection campaign on multiple types of applications, particularly microservice-based applications, and compatible with various container orchestration technologies and fault injection tools. The *Defektor* configuration follows an high-level approach, based on an injection campaign plan specifying the instructions for the *Defektor* operation and the parameters of the fault injection campaign. *Defektor* automates the entire workflow, consisting of defining the campaign plan, generating a workload, specifying and injecting the faults, and collecting data, aiding the experiment repeatability, improving the consistency of results, and saving time.

Keywords

Microservices, Fault injection, Cloud-native

This page is intentionally left blank.

Resumo

Para alcançar confiabilidade, os *designers* de sistemas costumam recorrer a mecanismos de tolerância a falhas. A avaliação desses mecanismos requer a observação de avarias, que normalmente são relativamente incomuns. Para aumentar a ocorrência de avarias, são empregues técnicas de injeção de falhas, resultando num maior número de eventos de avarias e permitindo a avaliação das propriedades de confiabilidade do sistema. Embora existam várias ferramentas de injeção de falha para este fim, estas são geralmente limitadas em extensão, aplicabilidade e na capacidade de configuração para aplicações baseadas em micro-serviços.

Propomos o *Defektor*, uma ferramenta generalista e extensível capaz de controlar uma campanha de injeção de falhas em vários tipos de aplicações, particularmente aplicações baseadas em micro-serviços, e compatível com várias tecnologias de orquestração de contentores e ferramentas de injeção de falhas. A configuração do *Defektor* segue uma abordagem de alto nível, com base num plano de campanha de injeção especificando as instruções de como o *Defektor* deve operar bem como os parâmetros da campanha de injeção de falha. O *Defektor* automatiza todo o fluxo de trabalho, consistindo em definir o plano de campanha, gerar uma carga de trabalho, especificar e injetar as falhas e recolher os dados, auxiliando na repetibilidade das experiências, melhorando a consistência dos resultados e economizando tempo.

Palavras-Chave

Micro-serviços, Injeção de falhas, *Cloud-native*

This page is intentionally left blank.

Acknowledgements

This work would not be possible without the effort, assistance, and support of my family, friends, and colleagues. Thus, I would like to express my heartfelt gratitude to all of them.

Starting by thanking my mother and father, as well as my whole family, for their support during this entire and long journey, and who always gave and will always give me some of the most important and beautiful things in life, love and friendship.

In second place, I would like to thank all professors who were directly involved in this project. To my supervisor, Professor Filipe Araújo, who contributed his vast wisdom and experience, to my co-supervisor, Professor Raul Barbosa, who contributed his vision and guidance about the main road we should take, and to Professor João Durães, for the time and help provided.

In third place, I would like to thank my colleagues and PhD students, André Bento and Jaime Correia for all of the time you have spent with me, supporting me through my doubts and problems, passing on your knowledge and wisdom, and, last but not least, for the informal talks.

In fourth place, I would like to thank Department of Informatics Engineering and the Centre for Informatics and Systems, both from the University of Coimbra, for allowing and providing the resources and facilities to complete this project.

In fifth place, I would like to thank to Portugal National Distributed Computing Infrastructure (INCD) for providing hardware to develop our solution and run experiments.

And finally, my sincere thanks to everyone that I have not mentioned and contributed to everything that I am today.

This page is intentionally left blank.

Contents

This page is intentionally left blank.

Acronyms

API Application Programming Interface. 5, 17, 20, 21, 23, 24, 28, 29, 33–35, 40, 48

AWS Amazon Web Services. 7, 13

CPU Central Processing Unit. 1, 13

DevOps Development and Operations. 2

GCE Google Compute Engine. 13

HTTP HyperText Transfer Protocol. xvi, 13, 15, 16, 45, 46, 49–55, 61

IDE Integrated Development Environment. 34, 59

JSON JavaScript Object Notation. 33, 34

JVM Java Virtual Machine. 34

OLTP Online Transaction Processing. 12

OS Operating System. 30, 31, 38

PID Process IDentification. 47

REST REpresentational State Transfer. 17, 20, 21, 23, 24, 28, 33–35, 40, 42

SaaS Software as a Service. 5

SOA Service-Oriented Architecture. 5

SRE Site Reliability Engineering. 2

SSH Secure SHell. 31, 32, 38, 45, 47, 61

UI User Interface. 2, 14–16

URI Uniform Resource Identifier. 32

URL Uniform Resource Locator. 43

VM Virtual Machine. 31, 32

YAML YAML Ain't Markup Language. 33, 34

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

This document presents the final dissertation in the context of the Master in Informatics Engineering, Specialization in Software Engineering advised by Prof. Filipe Araújo and Prof. Raul Barbosa and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

1.1 Context

With the exponential growth of software systems, new challenges are imposed to continue scaling services efficiently. For this purpose, new solutions and development patterns were explored, studied, and developed. One procedure that has emerged in the last decade has been to decouple the various components of an application (which until then were all encapsulated in a single block) into interconnected small blocks, each one responsible for performing specific functions. These components are known as microservices and have become the dominant trend in large companies selling products/services based on computing platforms. Even though this solution improves the system's resilience and availability, this architecture makes the observability and comprehension of failure propagation a more challenging task.

Since the system's various components are decoupled, where a larger cluster can have hundreds or thousands of instances of microservices, traditional telemetry techniques are unable to provide a complete picture of the system due to microservices' complex relations and dependency trees. To address this issue, new observability techniques must be employed to mitigate this problem. Thus, dedicated engineering teams use monitoring, logging and tracing to observe and maintain records of work performed in a microservice system. Monitoring consists of measuring aspects related to the infrastructure where the system is deployed, such as Central Processing Unit (CPU), memory, disk, latency, and others. The logging provides an overview of a discrete triggered log per event. Tracing oversees the flow of program execution, as requests go through the system's various components.

One of the tracing forms, used mostly in distributed systems, is called distributed tracing, and is characterized by monitoring applications even if their state is partitioned into multiple services, machines, or geographical locations. Despite being a valuable technique for improving system dependability, the frequency of failures, which is normally low, must be increased in order for engineering teams to precisely pinpoint the source of the failures.

In order to find the system's failures, a new discipline arose to act proactively against

possible hidden threats in the system. This discipline is called Chaos Engineering, and it has one single mission: make the system as resilient as possible. The pioneers of this practice soon realized that the higher the failure rate, the more effective its detection and mitigation was. Thus, effort should be allocated to proactively inject faults in the system to collect relevant telemetry data. This data is then important to understand how the system behaves when some of its components do not work properly.

The combination of the two presented disciplines gives to Site Reliability Engineering (SRE) and Development and Operations (DevOps) teams enough tools and data on how resilient the system is and on how much margin of improvement there is.

1.2 Goals

One of the limitations we discovered in our current work is a lack of systems capable of controlling a fault injection campaign and providing useful data for the development of analytic tools. To mitigate this problem, our team proposes a generalist and extensible tool called *Defektor* that automates the entire injection campaign workflow and data collection, facilitating experiment reproducibility, enhancing consistency of results, and saving significant time. For example, the tool may purposefully delay one service's response to ingress requests in order to observe how it propagates to other services. Similarly, it may switch off or drain computing resources from architectural parts to check if the overall application can cope with the situation.

It is also our goal to give a simplified way of integration of our solution in the mainstream container orchestrators, so the practitioner does not have to install and configure the essential integrated components of *Defektor* manually.

One client User Interface (UI) will also be developed to agile the process of creating a descriptive plan responsible to give detailed instructions on how *Defektor*'s should perform an injection campaign.

In short, it is expected that skills in distributed systems will be developed as well as a good knowledge of the state of the art regarding telemetry and chaos within a cluster of microservices. This thesis should culminate with a software artifact capable of utilizing principles and methods from the project's two primary disciplines: distributed tracing and chaos engineering.

1.3 Research Contributions

From the work presented in this thesis, the following research contributions were made:

(communication) Gonçalo Baptista, Jaime Correia, André Bento, João Soares, António Ferreira, João Durães, Raul Barbosa, and Filipe Araújo. Defektor: An Extensible Tool for Fault Injection Campaign Management in Microservice Systems. INForum 2021.

(paper) Gonçalo Baptista, Jaime Correia, André Bento, João Soares, António Ferreira, João Durães, Raul Barbosa, and Filipe Araújo. Defektor: An Extensible Tool for Fault Injection Campaign Management in Microservice Systems. Internation Parallel and Distributed Processing Symposium (IEEE IPDPS 2022).

The latter contribution is still subject to review. The conference reviews will be sent out to the authors on November 30, 2021.

1.4 Document Structure

This document is structured as follows:

- Chapter 2 - aims to give a good understanding of the core concepts necessary to understand what will be the proposed solution.
- Chapter 3 - the existing technologies and tools on the two core disciplines of this thesis are presented and discussed.
- Chapter 4 - provided a brief description of the capabilities we envisioned for our solution. To clarify it, some functional requirements are identified and well defined using use cases.
- Chapter 5 - the architecture of our proposed solution is presented following the C4 Model.
- Chapter 6 - the tools and technologies used, and a detailed and rigorous overview of the main components of the developed tool.
- Chapter 7 - provided a proof of concept by creating an experiment setup and analyse the results.
- Chapter 8 - an analysis will be made of the planning of this project.
- Chapter 9 - aims to give some reflections on what was developed during the thesis and what is planned to be implemented in future work.

This page is intentionally left blank.

Chapter 2

Background

The current chapter aims to give a good understanding of the core concepts necessary to understand what will be the proposed solution. To do so, the core concepts—Microservices (section 2.1), Chaos Engineering (section 2.2) and Distributed Tracing (section 2.3)—are presented and explained with evidence-based literature review.

2.1 Microservices

The term “Micro-Web-Services” was initially introduced in 2005 by Dr. Peter Rogers at a cloud computing conference, the Web Services Edge conference. Two years later, Juval Löwy expanded the idea of promising case studies about a granular use of services in his book [42]. In 2011, during an event in Venice for software architects, the term "microservices" was used to describe a prototype architectural style based on the granularity of services. This term would be eventually be formally used to describe this Service-Oriented Architecture (SOA) in 2012 [44, 46].

Microservices is an architectural style that structures an application as a set of small independent services, interconnected through API’s, organized in order to implement rules and business logic [59, 15].

The emergence of this architectural style arises from the empirical and technological advances in computer science, more specifically in software engineering and cloud distributed computing. In this regard, improvements in Application Programming Interface (API) and several contributions in technology stack (virtualization and containerization), service management and architecture setting both in proposed solutions and literature reviews [55]. Notwithstanding the scientific progress, the exponential growth of the codebases made too difficult to make functional changes and maintain big applications. Thus, Software as a Service (SaaS) approach is almost mandatory to continue scaling [71].

Microservices stand out for its minimal and small form with a well-defined single responsibility. They provide increased resilience, scalability, maintainability and testability when comparing to other architectural patterns. These attributes enable businesses to optimize resources. On the one hand, teams tend to develop and deploy faster than other approaches, leading to a faster time to market. On the other hand, decoupled small services reduce infrastructure costs and minimize downtime [52].

Prior to the popularity of the microservice architecture pattern, the monolithic architecture one was more in vogue. This pattern packages and deploys all components in

one single unit application. For instance, a server-side application is commonly built as a monolith because all features are bundled in the same package. Thus, any functional change or bugfix demands compiling and deploying a new version.

Figure 2.1 provides a very clear representation of the main differences of both architecture patterns.

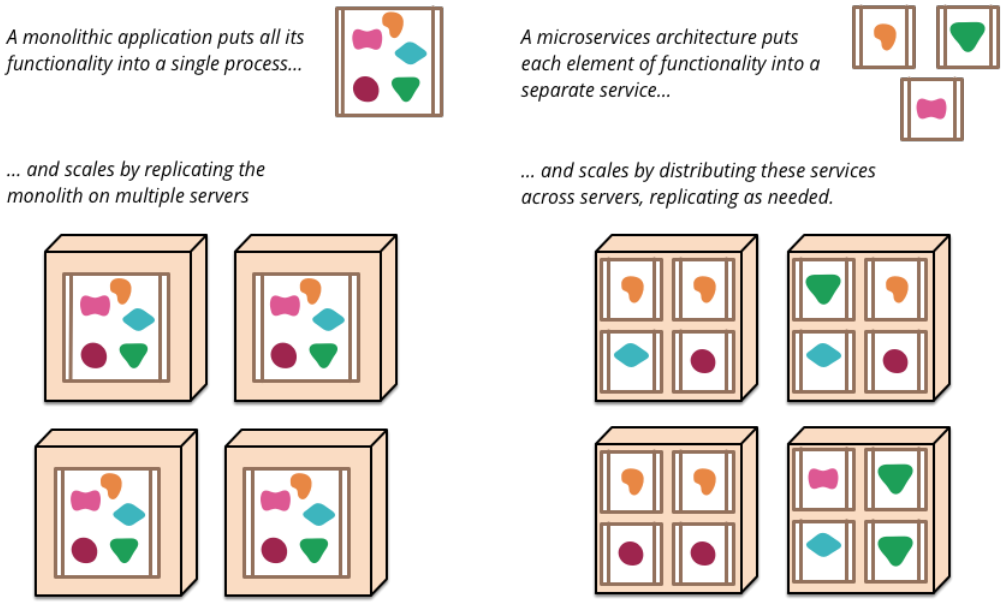


Figure 2.1: Monolithic and Microservices architectural patterns [19].

These two approaches are both valid in the right context. For instance, a product following the monolithic architectural pattern starts to show its weaknesses when the client demands features either in large quantities or in complexity. Thus, this solution fits better in simple applications that do not demand much scalability or business logic. Withal, as this pattern's components are usually very coupled and dependable to work correctly, the number of developers allocated to such a project should be small.

Considering that a company owns a reasonable number of human resources and its client demanded a very complex and scalable product. Even in this case, this monolithic solution might still not be the most appropriate if among the developers there is no expertise and know-how in microservices. Microservices demand well-qualified professionals, such as DevOps, to leverage their business value properly [21].

Two areas where microservices tend to be problematic are monitoring and predicting behaviour facing turbulent conditions, especially when the system takes large dimensions. For this matter, a multitude of disciplines and techniques have been created to help engineers mitigate these problems. Two of them will be covered in this document due to their importance for fully understanding the proposed solution: chaos engineering (Section 2.2) and distributed tracing (Section 2.3).

2.2 Chaos Engineering

As previously discussed, despite their many advantages, microservice-based platforms are sometimes too complex to foresee the repercussions of unanticipated events such as a

hardware failure, invalid parameters in runtime configuration, or abnormal usage of the clients' service. Although each microservice individually may be very focused and well implemented, the increased interaction between many discrete components increases the possibility of failures. Also, because the system is now distributed among many loosely coupled components, fault localisation and causality analysis become very hard, making the evaluation of the system's dependability properties complex.

Fault tolerance mechanisms are an important aspect of modern systems, to handle fault activation occurring in the operational phase and avoid or mitigate failures. Thus, during development, it is very important to understand the system behaviour when some components fail, in order to better design mitigation and recovery mechanisms. Because the activation of hidden faults and consequent failures are relatively rare (otherwise those faults would have been found and corrected), practitioners use fault injection techniques to increase the rate of fault activation in order to observe failures and be able to characterise the system's behaviour.

One discipline that employs this principle (deliberately increase fault rate) is Chaos Engineering. Formally, Chaos Engineering is "the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production" [58]. That is, arbitrary and intentional failures are introduces into production environment to assess whether a malfunction on a service/instance triggers a chain of events that can impact negatively the availability and the performance of the system.

This discipline was first introduced by Netflix when, in 2010, they intended to migrate their services from their local infrastructure to Amazon Web Services (AWS) cloud infrastructure. To validate if the cloud provider could guarantee their availability and reliability quality attributes, Netflix engineers designed a resiliency tool called Chaos Monkey that would abruptly terminate production instances. This practice may appear illogical, especially because it is attempting to weaken the system in the production environment. However, essential data was generated and gathered either about any critical weaknesses present in the system or to validate implemented mitigation plans [73]. In the following year, and inspired by the success of Chaos Monkey, new services (denominated with other simian names) with new capabilities of inducing abnormal conditions in the system started to be developed and put to use, increasing the resilience of their infrastructure [7, 51]. In the subsequent years, other companies such as Microsoft [49], Google [10], Amazon [10] and Facebook [66] started to realize the importance of this technique to test the resilience of their own systems.

The formalization of this discipline consists in five fundamental principles [58]:

1. Build a Hypothesis around Steady State behaviour
2. Vary Real-world Events
3. Run Experiments in Production
4. Automate Experiments to Run Continuously
5. Minimize Blast Radius

In order to clarify these principles, the first step is to accurately and completely measure relevant metrics such as throughput, latency and failure rate of packets with the system in a steady state. Then, occurrences and events, generated in the production environment, must be observed and prioritized by possible impact on the system and/or frequency of

occurrence. The experiments carried out, such as abruptly ending arbitrary computational instances, should be fully automated and executed in a production environment. This way, it is possible to obtain real data through real interactions with the system. Finally, carrying out experiments in this environment may cause potential unnecessary impacts on customers. Although there must be a well-defined plan for possible short-term negative impacts, it is the responsibility and obligation of the teams responsible for this process to ensure that the consequences of these systemic tests are minimized, controlled and contained. It is worth noting that during this process, the important thing is not to monitor and understand how the system works, but if it works.

Based on the Chaos Engineering principles, a proposed solution [7] to run a experiment may follow the next steps:

1. Start by shaping system's 'steady state' with some measurable output that indicates its nominal behaviour.
2. Hypothesize that this steady state will be verified either in control and experimental group.
3. Introduce any kind of realistic malfunction in the system.
4. Try to refute the hypothesis comparing the behaviours between the experimental and the control group.

The more difficult it is to disturb the system's normal behaviour, the more confident we are in our system's ability to withstand turbulent situations. Any failures detected, on the other hand, must be addressed as soon as possible before they reach the system and severely damage the client experience.

These failures are only possible to be observed when analyzing the data generated by employed observability and telemetry tools. One of the most reliable observability technique used in microservices environments, as it was previously mentioned, is Distribute Tracing. This subject will be addressed in the Section 2.3 to give a good understanding of its purposes.

2.3 Distributed Tracing

Once the various components of the system are decoupled, the traditional logging methods are not effective for fully comprehending the big picture of the system's health because they lack contextual metadata to troubleshoot a request as it travels through an enormous dependability tree. Thus, to address this issue, teams often resort to Distributed Tracing to enhance the system's telemetry.

Distributed tracing "is a method used to profile and monitor applications, especially those built using a microservices architecture" [54], which helps DevOps teams to pinpoint system anomalies, diagnose steady state problems and have a good notion of resources attribution [61].

Conceptually, distributed tracing services proposes a solution that assigns each external request a unique ID, propagates the ID of that same request to all services involved in its treatment. All the log messages must also include that ID and finally, record the information (for instance, start and end timestamps, duration, HTTP status code, etc.) in a centralized service [57].

Multiple standards and tools have emerged since the early days of large-scale microservices implementations, when each organization was developing their own solution. As a result, there is no standard specification in place today as the various available solutions have distinct formats for their components as well as varied annotation conventions. To address this issue, multiple organizations (Google, Microsoft, IBM, and others) and people have been working to build a standard for interoperability between tracing tools.

For instance, the OpenTracing specification [53] is done at the level of what is called a trace. A trace represents an end-to-end request inside the system and it is composed by building blocks that contain the same *trace identifier*, called spans. Spans represent work done by a single service and its relevant metadata (e.g., duration, timestamps, etc.). As the requests goes through the chain of services, spans are created with a *span identifier* associated to its trace. This *span identifier* as well as a *parent identifier* are necessary to represent parent/child relationships It may also occur that a request only targets one service that is dependability free. Therefore, a trace only contains a span and we can say that this trace is a span.

Figure 2.2 represents a trace and gives an idea, with a simplified sample, on how spans correlate within themselves and progress through time.

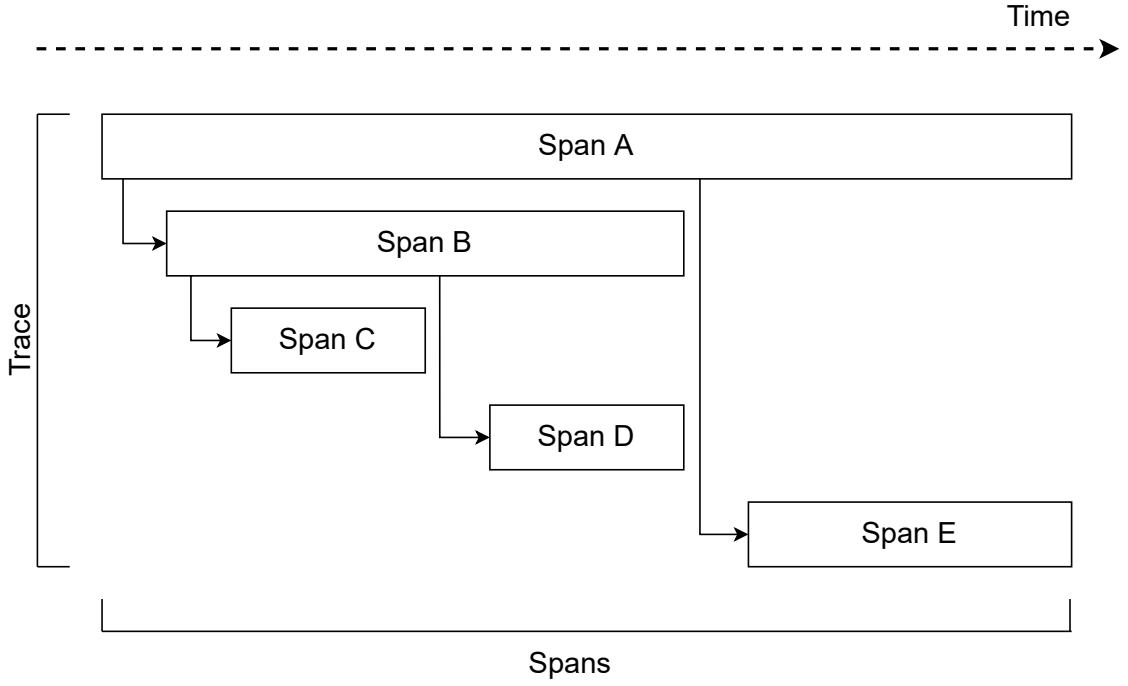


Figure 2.2: Sample trace over time.

In Figure 2.2 we can observe a trace composed by five spans, each one of them having a parent span except the "Span A". This is the root span or also called an "orphan span". In the other hand, only "Span A" and "Span B" possess child spans. We can also observe that a span can be triggered by any span, as long as they are dependable of each other.

Looking at the factor time, we can observe that all the spans have a start time, an end time and, therefore, a duration. It is expected that a parent span will start before any child span and its duration will cover all its child spans' durations. Thus, it is expected that a trace only ends if no spans in its chain are still doing work.

It should also be noted, looking at Figure 2.2, that "Span A" has two direct dependable spans—"Span B" and "Span E"—where the second one only starts after the first one ends.

This is most likely scenario, yet it may occur that a parent span triggers two or more spans at the same time. This is only possible in a scenario that the child spans do not depend on each other and, therefore, can execute as parallel tasks.

These distributed tracing tools most often provide what is called a span tree. A span tree represents, as a graph, all spans' dependencies inside one trace. This gives a clearer view of causality relationships among spans/services. This method abstracts the time factor as it is not relevant for dependability purposes.

To elucidate this technique, we are using the Figure 2.2 sample trace to build its span dependability tree, shown in Figure 2.3.

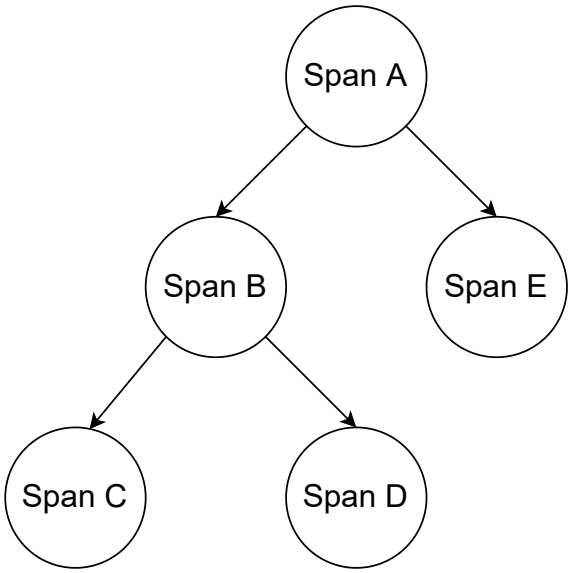


Figure 2.3: Span tree.

This technique can be extended from one trace to all traces gathered by a trace collector. As one trace does not represent the wide range of shapes a trace can have in a system, nor is it trivial to infer all service dependabilities based on the generated ones, some distributed tracing tools provide what can be called the system architecture graph. Recurring to the sample of traces gathered by the system, it is possible to infer what the causality relationships between services are in the whole system, in multiple contexts.

Once all nuclear concepts are exposed and reviewed in the literature, the reader should be equipped with an ability to understand the technologies that will be addressed in the following chapter (Chapter 3) as well as the proposed solution of this thesis (Chapter 4 & Chapter 5).

Chapter 3

State of the Art

In this chapter, we present and discuss a literature overview about fault injection subject and the existing technologies and tools on the two core disciplines of this thesis—chaos engineering and distributed tracing.

It is important to note that all tools are open-source and none of them serve the same purpose as the solution presented later in the report. Many of them may even be integrated into it. However, since our solution share principles and functionalities with these two distributed computing disciplines, an enlightened view of the state of the art is essential.

The comparisons made between the various tools regarding their pros and cons will only give an aggregate view of the characteristics that will be addressed in each tool subchapter. Thus, this study do not have the purpose to rank them in order to choose one.

3.1 Fault injection

Testing a system's resilience and fault tolerance is an essential element of the validation process for distributed systems. This requires the observation of the system under evaluation in the presence of faults. Fault activation, on the other hand, is typically an uncommon occurrence. To address this, fault injection has been used to accelerate fault activation by inserting artificial faults into a given component of the system to observe how other components or the overall system behave. The object of the observation is not the target itself but another part (or the entire system). Fault injection has been used for decades to evaluate system dependability properties [30], [9], [72], [16] and risk assessment [47].

Early work of [4] and [5] proposed the initial frameworks defining the conceptual components of a fault injection experiment: the set of faults (faultload), the set of operations to activate the system (the workload), the set of raw measurements (system observation) and the model to convert the raw measurements into meaningful properties concerning the system behaviour. Fault injection experiments are controlled by a set of typical tools which include the fault injector (actually inserts the intended faults into the target), the workload generator (submits the work to the system), a monitor (observes the system), and a controller (orchestrates the experiment), as described in the early work of [26].

Usually, the system is exercised first without any faults injected—the "golden run"—and then again, one or more times, with faults injected in a given component according to the faultload specification. Realistic faultloads representing real faults occurring in the operational phase are particularly hard to define, and even harder for complex software

systems, where it is very difficult not only to understand which faults are realistic, but also how to inject them.

The fault injector is, by far, the most difficult component to implement, due to the complex nature of the faults (software faults), and also due to to reachability and control issues, which are technically very challenging, often having to bypass the normal semantics and behaviour of the platform and operating system. Besides the technical challenges involved in building fault injectors, another issue surrounding fault injection experiments is the ratio of fault activation. Once inserted into the target system, the fault must be activated to eventually cause an error (an internal wrong state in the system) that may or may not cause a failure (the unwanted behaviour that is observed). In particular, for software faults, to activate the fault, the workload must ensure that the execution path covers the inserted fault. To overcome this difficulty, many fault injectors insert not a fault but an error. The premise here is that the errors are a representative consequential state of the intended fault.

Fault injection has been used for several decades both in academia and industry and many early fault injectors were developed for more or less specific target types and scenarios. Examples of early tools are: specific for hardware systems ([5], [32], [43], among many others), specific for simulating hardware memory-related faults, such as bit-flips and stuck-at faults (FIAT [60], FERRARI [29], FINE [31]), and specific for given target systems, such as Online Transaction Processing (OLTP) systems [72] or web servers [16]. The specificity for a given target system or experimental scenario ties the tools to specific platform mechanisms and capabilities, limiting the fault models the tool is able to inject. Several initiatives addressed this problem by proposing modular fault injection tools. Examples include Xception [9], NFTAPE [65], Goofi [1]. The success of such modular tools is moderate given that there remains some dependency from the underlying system or target and new scenarios have become relevant for which these tools were simply not prepared to address, such as microservice architectures.

3.2 Chaos Engineering

In this section, the most relevant chaos engineering tools explored—*Litmus* (subsection 3.2.1), *Chaos Monkey* (subsection 3.2.2) , and *Istio* (subsection 3.2.3)—will be presented, introducing each one of them and comparing their pros and cons.

These tools all share one aspect in common: they can enforce failures in a service belonging to an application based on a microservices architecture. However, they do not automate a typical chaos engineering experiment, like the one presented in Section 2.2, as it is a very specific task for every microservices cluster.

3.2.1 Litmus

Litmus is an open-source toolset for cloud-native applications created by MayaData in 2018 capable of doing Chaos Engineering. It provide tools to orchestrate chaos inside a *Kubernetes* [11] microservices cluster at multiple levels: container [14] level, pod [38] level or node [36] level.

It adopts *Kubernetes*' approach to define the desired chaos experiments in a declarative manner, using manifests with custom resources. It also manages metrics for each chaos experiment to custom durations or severities of each fault injector. All the fault injectors

can be found in Litmus' chaos experiments documentation [41]

Litmus is composed by four components:

- Chaos Operator
- Chaos Experiment
- Chaos Engine
- Target Application

The Chaos Operator is the core of the tool, and it is in charge of running the experiments and report its results after the experiment is finished. These experiments, called "Chaos Experiments," are the actual fault/chaos introduced in the system, e.g., killing a container inside a pod, delete or drain Central Processing Unit (CPU) and memory resources in a pod, or stress or restart a node.

The Chaos Engine is the linker between the Chaos Operator and the Target Application. It is the component responsible for receiving any experiment parameters, such as duration and the target. Finally, the Target Application is the actual target of the chaos experiment. Some configurations are needed to be performed to enable the Chaos Operator to find the target.

3.2.2 Chaos Monkey

Among all the Chaos Engineering tools, *Chaos Monkey* is one of the pioneers and arguably the one that kickstarted Chaos Engineering's usage outside large companies. Netflix released it in 2012, and its major purpose was to randomly disable production instances to give confidence that the system was resilient enough to this type of failure, so the customers would not experience punctual malfunctions.

The project is open-source and it was initially prepared to only deal with AWS. However, as the years went by, new supported cloud environments were added such as Google Compute Engine (GCE) and *Kubernetes*.

Experiments with this tool were made in the middle of a business day, in a strictly monitored environment with engineers standing by. Thus, any failures in the system could either be contained and do not impact customers streaming experience but also be a learning lesson to build automatic recovery mechanisms.

3.2.3 Istio

Istio is an open-source service mesh that controls how microservices share data with each other. It uses sidecar proxies next to microservices to forward requests from/to other services. It is also crucial to distributed tracing tools as they use this layer of abstraction to get the data about the ingress and egress requests.

Even though this tool may be tagged as Chaos Engineering tool, it is in fact a service mesh capable of doing chaos. It provides to those who setup their service mesh in a microservices cluster two plugins to enforce failures inside the system: Hypertext Transfer ProtocolHyperText Transfer Protocol (HTTP) delay fault and HTTP abort fault.

The way *Istio* enforces chaos is presented with an example in Figure 3.1.

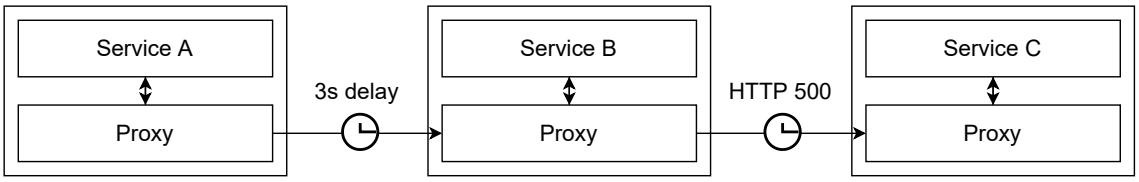


Figure 3.1: Sample of *Istio* injecting chaos.

As already discussed, this tool can control requests within the cluster. As we can see in Figure 3.1, "Service A" sends a request to "Service B" with a 3-second delay. If "Service B" does not timeout, it will send a bad request to "Service C", causing the request to fail. Therefore, the requested operation will not be performed completely.

This parameters are provided using a manifest where it is possible to include how many seconds an arbitrary service should delay a request to another arbitrary service. It can be also included the probabilistic value of the fault to be injected, either in delay and in abort.

3.2.4 Comparison

In this subsection, a comparison of advantages and disadvantages between the various covered tools will be presented. As already mentioned, it is not intended to make any sort of ranking among them, but to have a clear idea where each one stands out from the others and where it could be improved. This comparison is shown in Table 3.1.

Table 3.1: Chaos Engineering tools: pros and cons comparison.

	Litmus [40, 41]	**Chaos Monkey** [50]	**Istio** [24]
Pros	Open-source; Disposes around fifty chaos experiments (type of fault injectors); Web User Interface (UI) that provides a dashboard to monitor successful workflows; Good documentation;	Open-source; It has solid historical working proof; Built into Spinnaker [63]; Encourages to prepare for random instance failures;	Open-source; Easy setup; Can flag that a span was impacted by the failure; Platform (container orchestrator) independent;
Cons	Only works in *Kubernetes*; Steep learning curve; Hard troubleshooting; Hard to manage permissions based on what experimented is desired;	Hard to control the experience as it is randomized; Requires Spinnaker and MySQL [48]; No UI;	Only provides two fault injectors; No UI;

3.3 Distributed Tracing

In this section, the most relevant distributed tracing tools explored during a research phase will be presented, giving a introduction to each one of them and comparing their pros and cons.

These tools aim to collect information about traces generated within a cluster of microservices and present them to the user, DevOps, through an user interface . Yet, they

are not equipped to reveal to DevOps any symptoms that the system may not be operating in its expected behaviour. This task have to be performed by a specialized team based on the information generated by the distributed tracing tools.

The tools will be presented following the order of presentation and, at the end, a comparison is made between the pros and cons of each one.

3.3.1 Zipkin

Zipkin is an open-source software based on Google Dapper design. It is Java-based application and provides a number of distributed tracing functionalities. Its main goal is to gather timing-based information about traces to troubleshoot latency problems in a microservices environment.

This tool was adopted by large companies such as AirBnB and Uber. It supports OpenTracing, OpenCensus and OpenTelemetry frameworks and provides an UI that no only presents graphically traces' data and spans' metadata but also presents dependency diagrams. The workflow/architecture is shown in Figure 3.2.

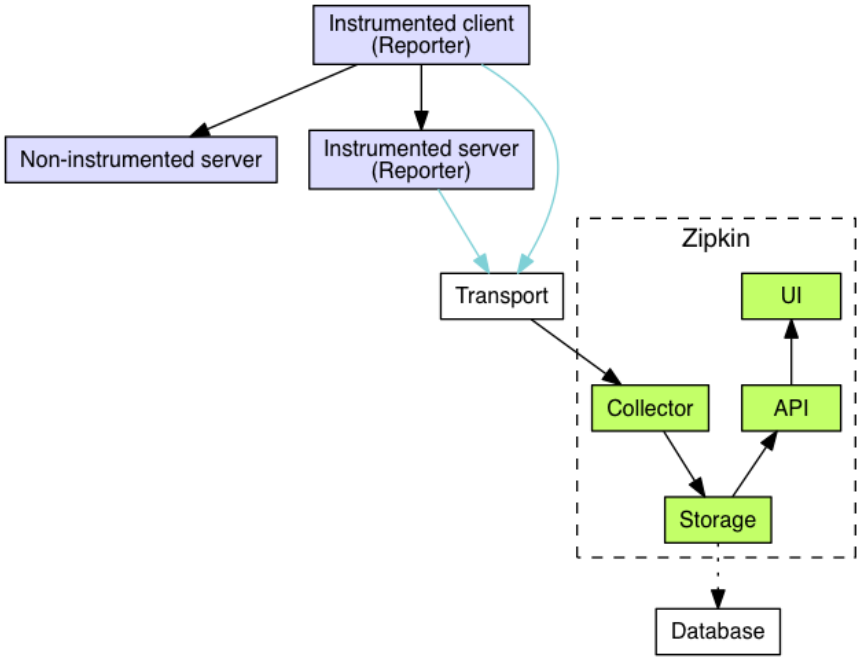


Figure 3.2: Zipkin architecture [75].

This tool can persist traces either in MySQL, Cassandra [2], Elasticsearch [17] and an in-memory database. In addition, Zipkin provides transport mechanisms like, e.g. RabbitMQ, HTTP, and Kafka, and a it provides libraries for most popular high-level languages such as Java, and JavaScript.

3.3.2 Jaeger

Jaeger was created by Uber and was written in Google's programming language, Go. It is similar to Zipkin (in fact, it was designed based on Zipkin implementation), but there are some differences: Jaeger provides dynamic sampling (Zipkin only provides fixed sampling),

a REST API, a renewed ReactJS-based UI and it only as full support for OpenTracing standard.

This tool's architecture, shown in Figure 3.3.2, is more distributed and complex that Zipkin's. Yet, it leverages performance, readability and scalability.

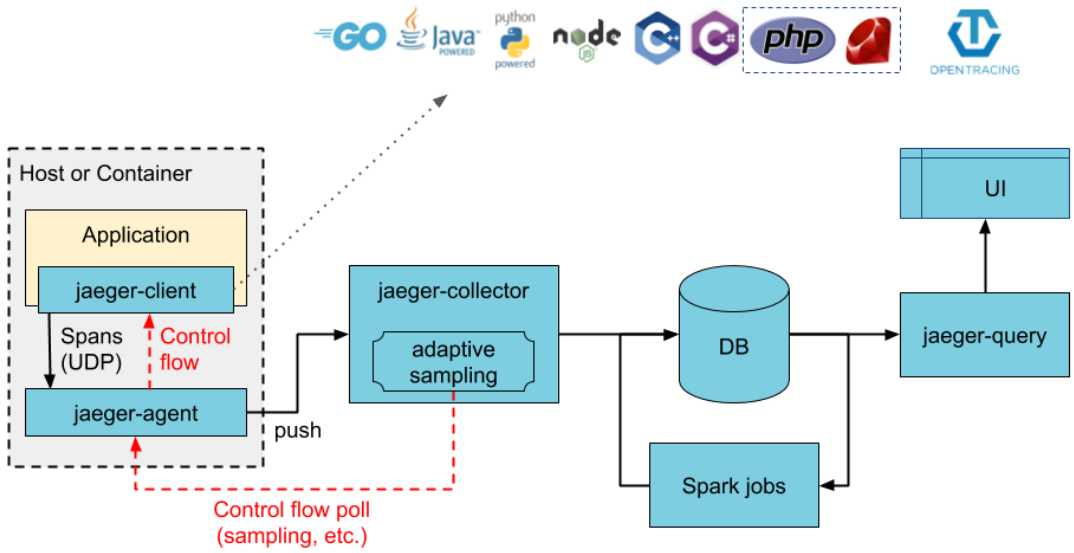


Figure 3.3: Jaeger architecture [27].

3.3.3 Comparison

In this subsection, a comparison of advantages and disadvantages between the various covered tools will be presented. As already mentioned, it is not intended to make any sort of ranking among them, but to have a clear idea where each one stands out from the others and where it could be improved. This comparison is shown in Table 3.2.

Table 3.2: Distributed Tracing tools: pros and cons comparison.

	Jaeger	**Zipkin**
Pros	Open-source; Dynamic sampling rate; Renewed browser UI.	Open-source; Provides multiple transport protocols; Supports main distributed tracing specifications; Browser UI;
Cons	Only supports two transport protocols (Thrift and HTTP). OpenTelemetry is not supported yet;	Fixed sample rate;

Now that the start of the art of tooling regarding distributed tracing and chaos engineering is covered and the core concepts were also presented, we consider that the reader is properly equipped to fully understand what will be exposed in the next chapters regarding the proposed solution.

Chapter 4

Requirements

In this section, we present the main features of *Defektor* and explain the interaction of the tool with the underlying systems to control the fault injection experiments and manage the target application execution. To clarify it, we give a general description about our proposed solution and discuss about *Defektor*'s actors and architectural driver. Nevertheless, some functional requirements are identified and well defined using use cases.

4.1 General Description

The product to be developed, called *Defektor*, a server application that exposes a REpresentational State Transfer (REST) API used by a command line client to manage injection campaigns.

Operation of *Defektor* requires a number of components: the application, an injection plan, a workload generator, a fault injector and a data collector mechanism, to extract the resulting data from a specified data storing systems. *Defektor* interacts with all the involved tools to exercise and inject faults on the target application, while running on a separate infrastructure, avoiding resource contention.

The workload generator can be instantiated on multiple slave computing instances and the injection plan the details specifying this aspect. The integration of *Defektor* with the other tools involved in the experiment, in particular the fault injection tools, is done through adaptor plugins.

The user interacts with the command line client to provide an injection plan containing high-level instructions to perform a fault injection campaign. Asynchronously, *Defektor* server will inject a fault in the target application affecting its dependability of one or more services and collects the data, leaving it available for the user to download later through the command line client.

Regarding the injection plan, it describes all aspects of the campaign, including the following information: system type, targets, fault injectors, workload generators, and data collectors. Based on the plan it receives, the *Defektor* connects to the supporting platform or infrastructure and allows the fault injector to manipulate the target application.

The user controls all aspects of the plan, however, typically, the plan will first define a golden run, and then the run with faults. Thus, the behaviour of the system without any fault or interference is observed beforehand, as is typical in the well established fault

injection, robustness testing and, later, Chaos Engineering experiments [58]. For example, a plan making two runs in addition to the golden run would be as follows:

1. Run workload generator for a predetermined amount of time;
2. Collect data (traces/metrics);
3. Apply fault 1 to a given target component A;
4. Run workload generator for a predetermined amount of time;
5. Collect data (traces/metrics);
6. Remove fault 1 injected on component A
7. Apply fault 2 to a second target component B;
8. Run workload generator for a predetermined amount of time;
9. Collect data (traces/metrics);
10. Remove fault 2 injected on component B.

4.2 Actors

Three actors were identified: System, User and Plugin Developer. The User is responsible for giving detailed instructions and triggering all the functionalities system provides, whereas the System is prepared to ingest user instructions and perform these tasks synchronously or asynchronously. These System's capabilities can be extended by the Plugin Developer either by adding more fault injectors or to extend the injection campaign to other microservice platforms.

4.3 Architectural Drivers

Enabling the use of a high-level plan and make *Defektor* extensible for different types of microservice-based systems were the two main design drivers.

The tool is made generic by allowing extension by plugin, enabling the addition of new fault injectors, data collectors and the ability to interact with new systems and platforms. All these plugins are referenced in a generic fashion in the plan, keeping it focused on the campaign definition, freeing both *Defektor* and the user from the implementations details. For example, shutting down a machine may appear in the plan, but the details about how that is done are encapsulated in the fault injector, which talks to the target system, by means of a system connector plugin (refer to Section 5).

4.4 Functional Requirements

Functional requirements are the defined functionalities that are intended to be present in a system and its components. To present it, we assigned an id and a priority to each one. The concept of priority represents the importance of each one to meet the product's

desired goal described in the previous section. Three priority levels were then used: High, Medium, and Low. High priority requirements are mandatory for the system to work properly. Without them, the system is not finished; medium priority requirements improve the system significantly and are of utmost importance; low priority requirements will only add value to functionalities already implemented.

The identified functional requirements are presented in 4.1 sorted by priority levels.

Table 4.1: Functional requirements specification.

ID	**Name**	**Priority**
FR-1	Add an Injection Plan	High
FR-2	Validate an Injection Plan	High
FR-3	List All Fault Injectors	High
FR-4	List Available Target Types	High
FR-5	List Target Instances	High
FR-6	Apply Workload	High
FR-7	Fault Injection	High
FR-8	Data Collection	High
FR-9	Manage Fault Injection Campaign	High
FR-10	Fault Injection Extensibility	High
FR-11	Target System Extensibility	Medium
FR-12	Data Collector Extensibility	Medium
FR-13	Delete an Injection Plan	Medium
FR-14	List All Injection Plans	Medium
FR-15	Get a Specified Injection Plan	Medium
FR-16	Add Slave Machine	Medium
FR-17	Delete Slave Machine	Medium
FR-18	List All Slave Machines	Medium
FR-19	Get a Specified Slave Machine	Medium
FR-20	Test Parallelization	Low
FR-21	Concurrent Injection Of Multiple Faults	Low
FR-22	Syntactic Sugar	Low

4.5 Use Cases

Use cases are a way of describing the system's behaviour, capturing the intention between the different actors and their interaction with the system, useful for properly describe functional requirements. They can be defined both in simpler ways, with an explanatory narrative, and in more complex and formal ways, detailing all its aspects and, depending on the system and its purpose, the ideal format for presenting them can vary.

Considering the complexity and the dimension of this project, we concluded that using the simple use case template matches the necessities for a proper description of each functional requirement. This template consists of an ID, the corresponding Actor, and a Basic Flow that explains the flow of the use case and all errors that may occur with a brief narrative.

The nineteen functional requirements that were identified in Table 4.1 are presented and detailed in the Tables 4.2–4.23 as use cases.

Table 4.2: Add an injection plan use case.

Use Case FR-1	Add an Injection Plan
Actor	User
Basic Flow	User submits a previously validated injection plan to give detailed instructions on how the system should operate the fault injection campaign. To do so, the user should describe it in a .json file and submit it via REST API endpoint. The manifest is then scrutinized to find any invalid input or repeated persisted plan. In the case of approbation, the plan is then persisted and executed. In case it is not approved either due to invalid input or repetitiveness, the system must be explicit about the reason.

Table 4.3: Validate an injection plan use case.

Use Case FR-2	Validate an Injection Plan
Actor	User
Basic Flow	User submits an injection plan in order to check any syntax error, inconsistencies, or invalid input. To do so, the user describes the injection plan in a .json file then submits it in the respective REST API endpoint. It is returned either valid or not valid and the reasons why.

Table 4.4: List all fault injectors use case.

Use Case FR-3	List All Fault Injectors
Actor	User
Basic Flow	User wants to get the information about all available fault injectors system can use to force malfunction in the target cluster. To do so, the user requests this information in the respective REST API endpoint.

Table 4.5: List available target types use case.

Use Case FR-4	List Available Target Types
Actor	User
Basic Flow	User wants to get the information about all available target types (pods, nodes, containers, services, etc.) that can be subject to a fault injection. To do so, the user requests this information in the respective REST API endpoint.

Table 4.6: List target instances use case.

Use Case FR-5	List Target Instances
Actor	User
Basic Flow	User wants to get the information about all available target instances that can be subject to an fault injection. To do so, the user provides a valid target type in the respective REST API endpoint. It is returned all available instances with that specific target type present in the system.

Table 4.7: Apply workload use case.

Use Case FR-6	Apply Workload
Actor	System
Basic Flow	System uses an artificial workload to exercise the target application. To do so, *Defektor* gathers the necessary properties from the submitted plan to properly orchestrate an arbitrary number of worker machines. These worker machines resort to configurable Docker containers [45] to exercise the target application for a requested duration and a degree of severity.

Table 4.8: Fault injection use case.

Use Case FR-7	Fault Injection
Actor	System
Basic Flow	System injects faults with different characteristics, severities, and target types, resorting to third-party fault injectors if they are available. To accomplish this, the injection plan describes the necessary properties, *i.e.*, injector plugin, target instance, and some relevant parameters, *e.g.*, the identifier of a process to kill.

Table 4.9: Data collection use case.

Use Case FR-8	Data Collection
Actor	System
Basic Flow	System has a data collection mechanism allowing the user to extract the resulting data from a specified data store of the target system.

Table 4.10: Manage fault injection campaign use case.

Use Case FR-9	Manage Fault Injection Campaign
Actor	System
Basic Flow	System must be capable of orchestrate all of the different components of the campaign. This components are plan management, the state store, the workload generation, the fault injection, and the data collection.

Table 4.11: Fault injection extensibility use case.

Use Case FR-10	Fault Injection Extensibility
Actor	System
Basic Flow	System's fault injection capabilities must be extended via plugins. To do so, the system must load any package file that can apply some malfunction in a cluster. Thus, the system is not only more extensible and has a well better-defined purpose for each component but also new injectors can be integrated without recompiling the core of the application.

Table 4.12: Target system extensibility use case.

Use Case FR-11	Target System Extensibility
Actor	User
Basic Flow	As there are multiple container orchestrators available in the market, our application intends to cover the most variety of them. Thus, the injection plan must include the proper instructions to identify and detail the supported target system type where injection campaign will be performed.

Table 4.13: Data collector extensibility use case.

Use Case FR-12	Data Collector Extensibility
Actor	User
Basic Flow	As there are telemetry tools available for different target systems available in the market, our application intends to cover the most variety of them. Thus, the injection plan must include the proper instructions to identify and detail the data collection system where injection campaign will be performed.

Table 4.14: Delete an injection plan use case.

Use Case FR-13	Delete an Injection Plan
Actor	User
Basic Flow	User specifies a persisted injection plan to be deleted. To do so, the user identifies what injection plan he wants to delete and then submits his intention in the respective REST API endpoint. In case the injection plan identifier provided does not meet any persisted plan, it is returned an error to the user. Otherwise, it is successfully deleted.

Table 4.15: List all injection plans use case.

Use Case FR-14	List All Injection Plans
Actor	User
Basic Flow	User wants to get the information about all persisted injection plans. To do so, the user requests this information in the respective REST API endpoint.

Table 4.16: Get a specified injection plan use case.

Use Case FR-15	Get a Specified Injection Plan
Actor	User
Basic Flow	User wants to get the information about a specific persisted injection plan. To do so, the user has to preemptively know the plan's identifier he wants to get the information and submit it in the respective REST API endpoint. If this identifier meets any persisted one, the information is returned. If not, an error is returned.

Table 4.17: Add slave machine use case.

Use Case FR-16	Add Slave Machine
Actor	User
Basic Flow	User wants to add a salve machine in order to perform the fault injection campaign from external sources. Thus, the data is more reliable because it deals with the network infrastructure where the cluster is deployed. To do so, the user describes all detailed information in a .json and submits it to the respective REST API endpoint. The manifest is then validated in order to find any valid input or any existing equal slave machine.

Table 4.18: Delete slave machine use case.

Use Case FR-17	Delete Slave Machine
Actor	User
Basic Flow	User specifies a slave machine to be deleted from the available slave machine list. To do so, the user identifies what injection plan he wants to delete and submits it in the respective REST API endpoint. If the identifier is valid, the target slave machine will be deleted. If it is not, it is returned an error.

Table 4.19: List all slave machines use case.

Use Case FR-18	List All Slave Machines
Actor	User
Basic Flow	User wants to get the information about all persisted slave machines. To do so, the user requests this information in the respectiveREST API endpoint.

Table 4.20: Get a specified slave machine use case.

Use Case FR-19	Get a Specified Slave Machine
Actor	User
Basic Flow	User wants to get the information about a specific persisted slave machine. To do so, the user has to preemptively know the slave machine's identifier he wants to get the information and submit it in the respective REST API endpoint. If this identifier meets any persisted one, the information is returned. If not, an error is returned.

Table 4.21: Test parallelization use case.

Use Case FR-20	Test Parallelization
Actor	System
Basic Flow	System must be able to perform parallel testing with multiple similar clusters in order to leverage more resources and scale horizontally. To do so, system must receive that instruction and the access information for all the cluster to perform injection campaigns in parallel.

Table 4.22: Concurrent injection of multiple faults use case.

Use Case FR-21	Concurrent Injection Of Multiple Faults
Actor	System
Basic Flow	System must be able to perform multiple fault injections in the same injection campaign. Although it does add complexity to results analysis, it may be convenient to push even harder system's reliability testing.

Table 4.23: Syntactic Sugar use case.

Use Case FR-22	Syntactic Sugar
Actor	System
Basic Flow	System must be able to make it easy to succinctly describe multiple runs with different parameterisations *i.e.*, improve the syntax for accepting vector parameters (*e.g.*, `slaves: [1, 10, 100]`).

Quality attributes and legal or technical restrictions were not identified. Thus, with that said, the architecture of the application has the necessary basis to be developed. This subject will be addressed in the next chapter.

This page is intentionally left blank.

Chapter 5

Architecture

In this Chapter, the architecture is presented following the C4 Model defined by Simon Brown [6]. This model consists in four diagrams: 1 - Context Diagram, 2 - Container Diagram, 3 - Component Diagram and 4 - Code Diagram. Here, we present the first three diagrams as the last one closely resembles the source code structure. To complete this chapter, we clarify and introduce some concepts related to *Defektor* and approach what we envision for our Plugin system.

Defektor is meant to be a generic, high-level, extensible fault injection campaign management tool that can be adapted, through plugin addition, to interact with and inject faults on any system and application. It is designed according to a client-server paradigm so that the process of managing a campaign is asynchronous to the practitioners and does not depend on the state of their local machine.

5.1 Concepts

To make the architecture easier to understand, this subsection lists and explains the concepts and abstractions used by *Defektor*.

- ***Defektor*:** a generic, high-level, extensible fault injection campaign management tool, designed according to a client-server paradigm.
- **dfk:** the command line client used to control *Defektor*.
- **Plan:** a high level description of the injection campaign, containing a list of injection steps, and respective parameters.
- **Injection:** each individual injection step, where a fault is injected with an *Injektor* and the application exercised with a *Workload generator*.
- **Injektor:** a plugin that implements the fault injection logic, or connects to an external fault injector.
- **Workload Generator:** a docker container encapsulated, application-specific, workload generator used to exercise the target application.
- **Slave Machine:** a generic, docker-enabled machine, used to run the *Workload Generator* and exercise the target application.

5.2 Context Diagram

In this section the Context Diagram is presented. This diagram has the least scope of detail, nevertheless it is extremely important because it represents our system as a big box with all the relationships with users and external systems. Therefore it gives a clear view of all the external dependencies that the system must manage. Figure 5.1 shows the interaction between *Defektor* and two classes of users and two external systems.

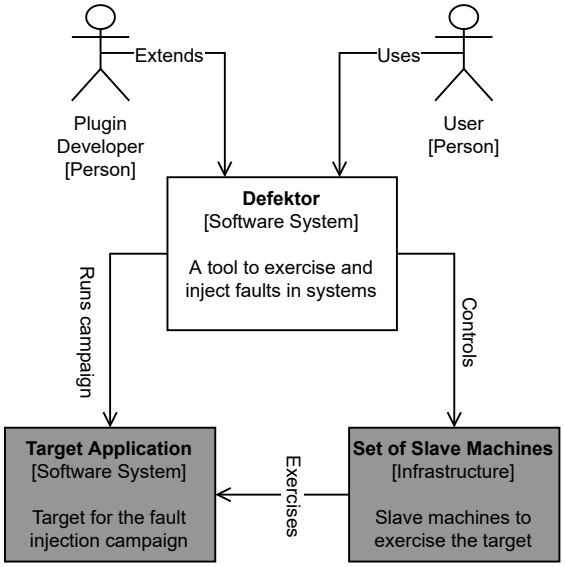


Figure 5.1: Context diagram.

Figure 5.1, it is possible to notice that our tool, named *Defektor*, interacts with two persons and with two external systems. The *User* is the practitioner deploying and starting fault injection campaigns, while the *Plugin Developer* is the person responsible for developing and maintaining plugins enabling fault injections, connection to new system types and data collection from different data stores. Once every configuration is ready, *Defektor* may run an injection campaign targeting the *Target Application*. To make the experiment more reliable, our application can use a set of *Worker Machines*, to generate load and exercise the *Target Application*.

5.3 Container Diagram

In this section the Container Diagram is presented. This diagram decomposes the big box that represented our system in the previous diagram in interrelated containers. Each container represents an executable and deployable sub-system. This provides a better understating of the shape of our software, at a high-level, and the distribution of responsibilities and dependencies across the main modules of the system. The Figure 5.2 presents this diagram to our proposed solution.

Figure 5.2 shows the components within our application. Starting from top to bottom, we have a console application, *dfk* that, with a minimalist interface, provides the *User* some abstraction and helpful hints regarding the interaction with *Defektor*'s REST API. This API is encapsulated in a container that we called *Defektor* Daemon. This component contains the core of the program and all attributes that were described in the formal requirements.

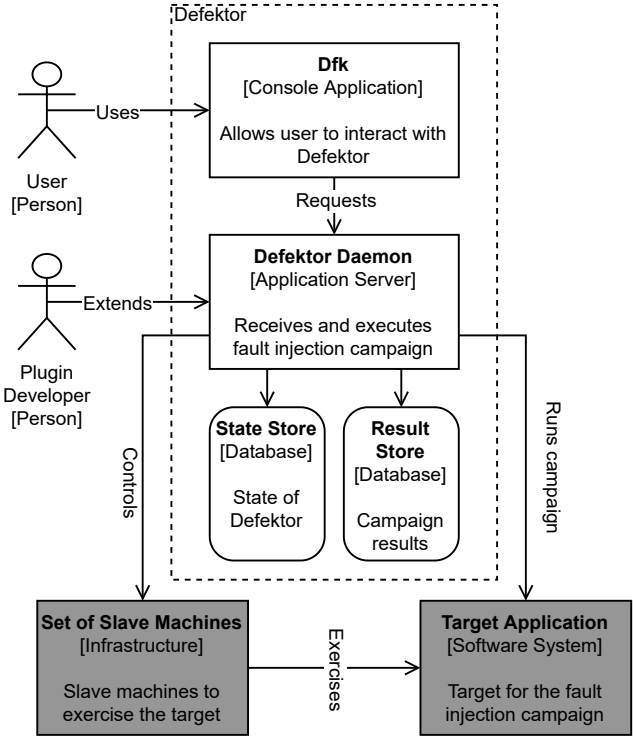


Figure 5.2: Container diagram.

To persist the crucial data, *Defektor* uses two different databases: one to persists its state (injection plans, slave machines information, status of an experiment, etc.), and the other to store traces to make these tracing data more disposable and organized to the *User*.

5.4 Component Diagram

This diagram gives a more zoomed-in view of the system, decomposing every decomposable container in a group of related functionalities encapsulated behind a well-defined interface. The Figure 5.3 portrays a more detailed vision of *Defektor* Daemon.

The plugins are the components that deserve the most emphasis are the plugin ones. We designed *Defektor* with two main principles in mind: make the tool agnostic to the cloud system and avoid recompiling the core, whenever a new type of target system is added. To achieve these goals, we designed *Defektor* following a plugin architecture. This system can be split into two different groups: core and plugins. The core parts are statically loaded and are responsible for generic functions, like serving the API, handling plans and managing and orchestrating the plugin modules. The plugins are run-time loadable, stand-alone components that provide specialized functionalities, such as fault injectors and connectors, to interface with supporting systems and platforms. This plugin architecture will be further detailed in the next subsection.

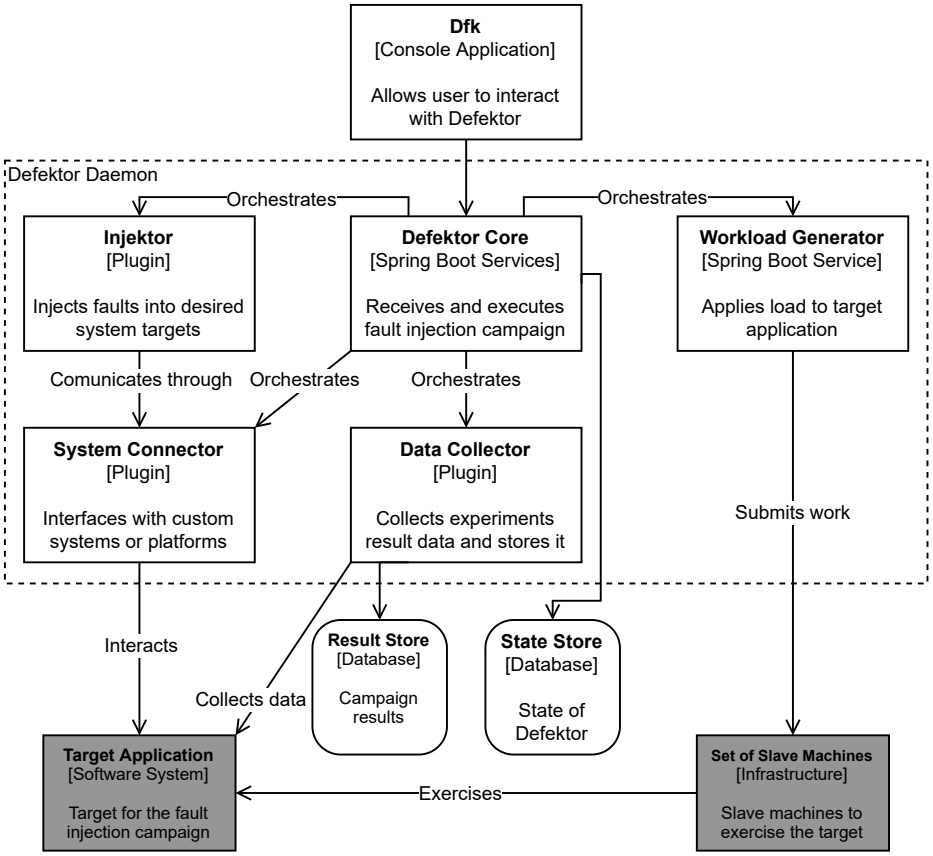


Figure 5.3: Component diagram.

5.5 Plugin System

We use the following types of plugins:

- *Injektors*, which are responsible for implementing the fault injection logic.
- *System Connectors*, which are responsible for interfacing with the platforms supporting the *Target Application*; this may for example be the Operating System (OS), an hypervisor, or a container orchestration system, such as *Kubernetes*.
- *Data Collector* plugins, which provide the specialized logic to extract the resulting data from the *Target Application*; common examples would be Prometheus, OpenZipkin or other data stores typically used for monitoring data.

Even though they are designed to be independent modules, *Injektor* plugins require System Connector plugins to interface with the system where the target application is running. The choice of splitting injection and system connection in two different types of plugins, comes from the expectation of having multiple injectors per system type, thus preventing the duplication of logic. In this fashion, when the interface for a system type changes, only the respective System Connector will require update.

Once a plugin is imported, its functionalities become available to be used by *Defektor*. The communication between both components must follow an abstract interface that must be implemented by the plugin.

A final, but not less important, consideration is usability. Since plans will have to differ based on the plugins used, each plugin should implement the necessary functions for inspection. As examples, from the *dkf* command line, the user should be able to determine which are the valid targets for a specific type of injection, or which targets are provided by a given system connector, as well as which configurations are needed by each injection type.

5.5.1 System Connector Plugins

The *System Connector* type plugin is a bridge between *Defektor* and the target application or its supporting platform, *i.e.*, OS, *hypervisor* or *Kubernetes*. The *Defektor* Core can query the *System Connector*, to get information of the *Target Application*, while the *Injektor* type plugin uses it to inject failures. This plugin must implement the following interface:

- `help`: returns a brief introduction and some details on the plugin interaction with the *Target Application* and how *Injektor* plugins should send their instructions to be performed.
- `configure`: a function that can be called to assign some configuration parameters to the plugin object.
- `getTargetTypes`: returns a list of all target types the connector can interface with, *i.e.*, machine, virtual machine, container, pod or process.

Nevertheless, some functions must be added to the plugin to properly connect and interact with the target application. For instance, to perform a fault injection in a single Virtual Machine (VM), the first question to arise would be: "How can I interact with the *Target Application*?". As an example, one could write an Secure SHell (SSH) *System Connector* that would need to be parameterised with the appropriate SSH credentials, by means of the `configure` function. Once the connection is established, there should be a function (*i.e.*, called *sendSSHCommand*) that could be used by an *Injektor* plugin, to send commands to the *Target Application*, to force its malfunction.

5.5.2 Injektor Plugins

The *Injektor* type plugin is responsible for injecting faults in the *Target Application*. *Injektor* plugins depend on *System Connector* plugins, to mediate the interaction with the *Target Application*. The latter plugins encapsulate the system, by allowing the interface to stay unchanged, while encouraging sharing of the same *System Connector* code.

Injektor plugins must implement the following interface:

- `performInjection`: receives injection parameters and performs the injection.
- `stopInjection`: stops or removes the failure injection.
- `getTargetTypes`: this returns a list of targets where this particular injector can perform a fault injection.
- `getTargetInstancesByType`: it returns all the instances, with an identifier, of a type where this particular injector can perform a fault injection.

- `getInjectionStatus`: returns the status of the injection (running, stopping, or stopped).

Following the same environment that was previously given as an example, the VM, we can assume that a user wants to perform the most basic failure injection: shutdown the instance.

Considering that virtually all cloud-native system instances are running some *Linux* distribution, and having the plugin access to the *System Connector* `sendSSHCommand` function, it becomes trivial to achieve this goal, by sending the string `sudo shutdown` to the *Target Application* via SSH.

5.5.3 Data Collector Plugins

The *Data Collector* plugin is responsible for collecting the data generated during each run. Effectively, this is the portion of the system that returns the data for analysis. As data is system, application and purpose specific, the tool returns it in some generic format, *i.e.*, an array of bytes or file, and it is up to the practitioner to interpret and analyse it. The plugin must implement the following interface:

- `configure`: a function that can be called to assign some configuration parameters to the plugin object.
- `getData`: function that returns data or some Uniform Resource Identifier (URI) to it.

As an example, since *Jaeger* is a widely used data store for traces, a *Data Collector* could be written for it that would return user-selected metrics for each run.

Chapter 6

Implementation

In this section we present what tools and technologies as well as the job that has been done regarding our three main software components: *Defektor* Daemon, *dfk* and *Defektor* plugins.

6.1 Tools and Technologies

In this section we present the tools and frameworks that were utilized to build our server application, our client interface and the plugin system.

6.1.1 Languages

The programming language we chose to develop *Defektor* Daemon and its plugins was Java. This choice was due to a number of reasons. Firstly, Java offers the capability of runtime class loading. This aspect turns out to be convenient for a proper plugin architecture implementation. Thus, whenever new plugins are added there is no need to recompile the *Defektor* core. The second reason is Java reflection. This feature allows us to inspect or modify runtime attributes of classes. This comes handy because the different types of plugins (*injektors*, *system connector*, and *data collectors*) can be implemented having different shapes and behaviors. The last reason has to do with the fact that Java contain multiple frameworks for REST API implementation. The chosen framework will be addressed later in this subsection.

Python was picked to develop *dfk*. Since *Swagger* [67], the software used to generate our client and server stubs, is compatible with a number of programming languages, Python was picked owing to personal preference.

Other languages were employed in *Defektor* development process. We used two markup languages: JavaScript Object Notation (JSON) and YAML Ain't Markup Language (YAML). The former was used to represent the injection plan. The latter was used to elaborate the OpenAPI specification.

6.1.2 Development Environment

Defektor was developed under *Ubuntu* [8], a *Linux* distribution. We also used a minimal *Ubuntu* image for our slave machines instances.

We employed IntelliJ IDEA by Jetbrains as our Integrated Development Environment (IDE) due to personal preference. This IDE provides intelligent coding assistance for Java Virtual Machine (JVM). Nevertheless, it is the most popular IDE for Java development [62].

To build and manage the Java project, we used Maven [3]. Furthermore, we used a *maven* plugin to generate the server stub from the OpenAPI specification.

Finally, we used Git [20] for version-control. Both *Defektor*, OpenAPI specification, and *dfk* projects are available in public repositories, at [68], [69], and [70], respectively.

6.1.3 Frameworks

As previously said, Java provides multiple frameworks for web applications. One that stands out for its popularity is Spring Boot [64]. Spring Boot was the chosen framework to implement our server-side for two reasons: the main reason is familiarity with the framework; the second reason is that *Swagger* supports creating a Spring server stub from the OpenAPI specification.

In what regards state storing, we opted for an unconventional approach. No relational databases were used because we assumed that would be an overkill solution. Instead, we used *MapDB* [28] framework that stores in regular system files any kind of representational data, having the performance compared to in-memory solutions.

6.2 Defektor Daemon

In this section we will address the main components of *Defektor* Daemon and explain with detail how we approached and implemented every one of them.

6.2.1 Plan

The injection plan is one of the components that we have put more work and attention because it is arguably one the most important components of this project. We needed to assure that all the components that act in the injection campaign had enough freedom for customization. Thus, *Defektor* expects different types of configurations or parameterisations depending what plugin it refers to.

We opted to represent it in JSON format since that is the language that our REST API handles objects to be serialized/deserialized. Listing 1 presents the *Plan* structure. To conserve space in the document and adding the fact that JSON and YAML are convertible into each other, we present *Plan* in YAML format.

We can observe that *Plan* contains three properties: *name*, *system*, and *injektions*. Property *name*, which is a *string*, represents an arbitrary descriptive name of the *Plan* *i.e.*, if the injection campaign is set to inject an HTTP Abort in *cart* service with a fault activation probability of 100%, one possible name could be *HttpAbortCart100percent*. Property *system* is an object that specifies what system the *Target Application* is deployed on. This object contains a *name* field that specifies with a string the name of the system *i.e.*, if the practitioner has its *Target Application* deployed in *Kubernetes* this property should be set to "kubernetes". Furthermore, at least one configuration of the desired system must be preemptively available to *Defektor*'s system configuration repository. The property *injektions* is an array of *injektion* type object. Each *injektion* contains the number

of *replicas* that the injection campaign will be run with the same parameterisation. It also contains an *ijk* (injektor) property that represents what fault injector and with which parametrization *Defektor* should apply in the *Target Application*. Other property present in the *injektion* type object is *workload*. This parameter contains an (Docker) *image* object that represents the conventional way to specify a Docker image: *user*/*name*:*tag*. The *cmd* field defines a command to be executed by the slave machine when it starts its job, whereas the *env* field sets environment variables for the slave machine. Moreover, it specifies how many *Slaves* will be employed to create load in the *Target Application* and how much replicas (Docker containers) will be deployed per *Slave*. Lastly, regarding *workload* object, it is parameterisable the *duration* of each run, the golden run and the fault injection run. To finish our describing of all parameters, we have a *dataCollector* property that specifies what *Data Collector* plugin will be used as well as the parameters to configure the data extraction.

```
1  name: <string>
2  system:
3    name: <string>
4  injektions:
5    - replicas: <number>
6      ijk:
7        name: <string>
8        params:
9          ...
10     workload:
11       image:
12         user: <string>
13         name: <string>
14         tag: <string>
15       cmd: <string>
16       env:
17         ...
18       slaves: <number>
19       replicasPerSlave: <number>
20       duration: <number>
21     dataCollector:
22       name: <string>
23       params:
24         ...
```

Listing 1: Injection plan structure.

It is worth nothing nothing that the ellipsis present in Listing 1 refers to objects that are dynamic, depending what plugin it refers to. This nuances will be addressed later in this section for each plugin.

6.2.2 REST API

Defektor exposes a REST [74] API used by a command line client to manage injection campaigns. This API was described in an *OpenAPI* specification document, available in Appendix 9. Furthermore, we employed a tool called *Swagger* to generate a *Spring* server

stub.

Defektor exposes twenty one endpoints grouped in six categories: plan, slave machine, target, plugin, system config, and campaign. To conserve space we omitted complex objects that may be included in POST operations. These can be found in the Appendix 9.

Plan

Defektor supports creation, syntax validation, deletion, and listing all or just one specific *Plan*. It does not support PUT operations since there is no logical reason to alter a valid *Plan* once it has been submitted because the injection campaign is already running or it has already finished.

Table 6.1 presents all the endpoints available to manage the resource *Plan*.

Table 6.1: Resource *Plan* REST endpoints.

Endpoint	**Description**
GET /plan	List all plans.
GET /plan/{planId}	Gets a plan.
POST /plan	Creates a plan.
POST /plan/validate	Validates a plan.
DELETE /plan	Deletes a plan.

Slave Machine

Defektor supports creation, and deletion and listing all or just one specific Slave. Even if is not good practice to expose an endpoint that can delete all entries of a resource, we consider that it may be exhausting to delete one by one a respectable amount of *Slave Machines*.

Table 6.2 presents all the endpoints available to manage the resource *Slave Machine*.

Table 6.2: Resource *Slave Machine* REST endpoints.

Endpoint	**Description**
GET /slave	List all slaves.
GET /slave/{slaveId}	Gets a slave.
POST /slave	Adds a slave.
DELETE /slave	Deletes all slaves.
DELETE /slave{slaveId}	Deletes a slave.

Target

Defektor supports listing all target types, and all target instances given a target type.

Table 6.2 presents all the endpoints available to manage the resource *Target*.

Table 6.3: Resource *target* REST endpoints.

Endpoint	**Description**
GET /target	List all targets.
GET /target/{target}	List target instances.

Plugin

Defektor supports listing all available *Injektor*, *System Connector*, and *Data Collector* plugins. Make a new plugin available to *Defektor* is not done via REST API. This will be addressed later in the report.

Table 6.4 presents all the endpoints available to list the resource *Plugin*.

Table 6.4: Resource *Plugin* REST endpoints.

Endpoint	**Description**
GET /plugin/ijk	List all *injektors (ijk)* .
GET /plugin/sysconnector	List all system connectors.
GET /plugin/datacollector	List all data collectors.

System Config

Defektor supports listing persisted system types configurations as well as adding new ones. The desired system configuration that hosts the *Target Application* must be available to *Defektor* before the plan is submitted. For example, if the practitioner describes that he wants to perform a fault injection in a SSH-enabled linux machine, there must be at least one configuration of that system type when the plan is submitted in order to *Defektor* start the injection campaign.

Table 6.5 presents all the endpoints available to manage the resource *System Config*.

Table 6.5: Resource *System Config* REST endpoints.

Endpoint	**Description**
GET /system/config	Lists all system configurations.
GET /system/config/{systemName}	Lists all systems configurations of one type.
POST /system/config	Adds a system config.

Campaign

Defektor supports deletion and listing all or just one specific *Campaign*. The *Campaign* represents the work done by *Defektor* regarding one submitted plan. It contains relevant

information about the injection campaign *i.e.*, running status (running, finished, abnormally interrupted), start and finish timestamps, as well as the data collection output.

Table 6.6 presents all the endpoints available to manage the resource *Campaign*.

Table 6.6: Resource *Campaign* REST endpoints.

Endpoint	**Description**
GET /campaign	List all campaigns.
GET /campaign/{planId}	Gets a campaign.
DELETE /campaign/{planId}	Deletes a campaign.

6.2.3 Workload Generator

As it was previously presented, a workload generator aims at exercising a target application. Any load generator contains some core parameters. Firstly, any workload generator must have a *host* which is the target for the load *e.g.*, http://host:8080/. Any load generator should also have a parameter that defines its severity *e.g.*, *numberOfClients = 10* or *requestsPerSecond = 25*. Finally, some unnecessary parameters may be included *e.g.*, insert errors in the requests or set a duration for exercising the target application.

To make *Defektor* capable of handling any application-specific workload generator, we decided that the best approach would be to use Docker. Thus, the practitioner should deliver his workload generator application encapsulated in a Docker image [13]. This Docker image is a file containing the application code, libraries, dependencies and other files needed to build a Docker container. By using Docker containers, *Defektor* does not need to manage its deployment and runtime issues since it is handled outside of the application itself. Furthermore, this approach is compatible with any OS. Therefore, a *Slave* machine can run in any machine as long as it can be controlled via SSH and it has Docker installed.

```
workload:
  image:
    user: 'robot-shop'
    name: 'rs-load'
    tag: 'latest'
  cmd: 'sh shesellsshellsbytheseashore.sh'
  env:
      host: 'https://wwww.robot-shop.com'
      numClients: 10
      silent: 1
      error: 1
  replicasPerSlave: 1
  slaves: 1
  duration: 600
```

Listing 2: Workload sample parameters.

In Listing 2, we present a sample parameterisation using a Docker image (robot-shop/rs-load:latest) provided by Instana to specifically exercise one of its microservice application, Stan's Robot Shop.

The explanation about every parameter present in this listing will be skipped since it has already been done in subsection 6.2.1. However, some deeper clarification about this component should be addressed.

Considering that the practitioner desires an n amount of *Slave* machines to exercise, n *Slave* machines configurations must be available to *Defektor*. Listing 3 displays the configuration parameters needed to provide a new *Slave* machine to *Defektor* work with.

```
1 address: 192.168.1.1
2 port: 22
3 credentials:
4   username: slave
5   key: "~/.ssh/id_rsa"
```

Listing 3: Slave machine configuration.

Once the n desired *Slave* machines are available, *Defektor* assesses how many replicas the practitioner desires for each *Slave*. The number of replicas indicates to *Defektor* how many workload generator Docker containers will be exercising the *Target Application* in each single *Slave* machine. This dynamic between *Defektor*, set of *Slave* machines, and the *Target Application* is presented in Figure 6.1.

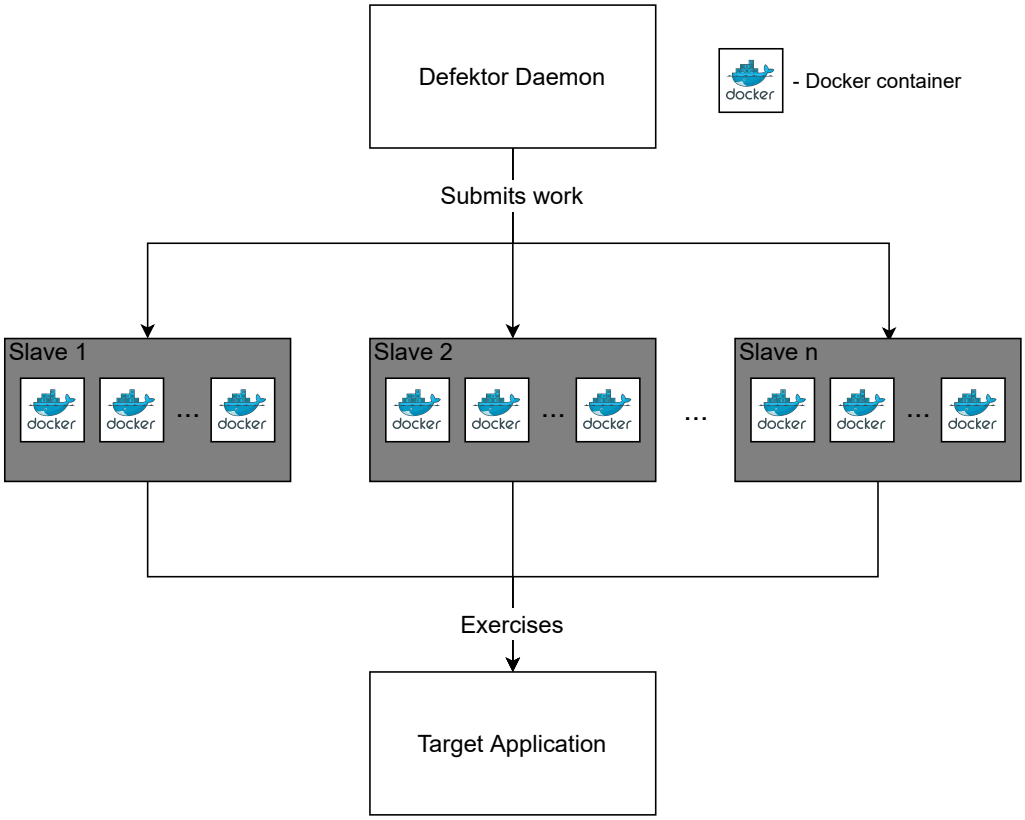


Figure 6.1: Workload workflow.

Here we can observe both slaves and replicas per slave (Docker containers in the figure above) scale horizontally. The number of slaves can be virtually infinite. Nonetheless, the amount of replicas may be constrained due to slave instances available hardware.

Defektor orchestrates all *Slave* machines using two different commands: *docker run*—to build and run the Docker container -, and *docker stop*—to stop a running Docker container. The *docker run* command is built with the given parameters. If we consider the parameters present in Listing 2, the resulting command would be:

docker run --name="some_container_name" -e host=https://wwww.robot-shop.com -e numClients=10 -e silent=1 -e error=1 robotshop/rs-load:latest

To execute *docker stop* command, one simple parameter is needed:

docker stop "some_container_name"

It starts by sending a *docker run* command and, after the set time *duration* has passed, it sends a *docker stop*, finishing the golden run. For the fault injection run, it executes the same commands, in the same order, separated by the same time *duration*.

Being R the number of replicas per *Slave*, S the number of *Slave Machines*, and C the number of artificial clients, we can use Equation 6.1 to calculate the total number of artificial clients, TC, exercising the *Target Application* in each run (golden and fault injection runs) during an injection campaign:

$$TC = S \cdot R \cdot C \tag{6.1}$$

6.2.4 State Store

Defektor needs a mechanism to persist its state. As previously discussed, we referenced the reasons why we leveraged MapDB Java framework to serve as *Defektor* state storing mechanism. This mechanism would have to persist different types of resources: *campaign, plan, slave, sysconfig*. Each one of them contains a list of its specific resource.

```
1 public class DefektorRepository<T> {
...
20     public void save(T t, String dbFilePath) {
21         DB db = DBMaker.fileDB(dbFilePath).make();
22         List<T> tList = (List<T>) db.indexTreeList("list",
           ↪  Serializer.JAVA).createOrOpen();
23
24         tList.add(t);
25         db.commit();
26         db.close();
27     }
...
89 }
```

Listing 4: MapDB add new object operation.

As a side note, all resources except *campaign* must be delivered by the practitioner via *Defektor* REST API. The *campaign* resource is generated by *Defektor* asynchronous

process during one injection campaign and then persisted to expose to the practitioner relevant data such as timestamps, and output files.

The persisted objects are stored in binary *.db* files. Listing 4 displays a small code example on how to persist a new object to a repository. We can notice how simple is to manage this resources with this framework by looking at line 21 and 22. Once line 22 is executed, the resource is managed as if it were a regular *java.List*.

Since there are four different types of resources to persist its state, we employed Java's generic capabilities with type parameter, *T*, so it can handle any of them with the same generic *class*. The only parameter that differs across all of the resources is the string *dbFilePath*, because each one has a unique *.db* file. This parameter points to either one of the four files inside the */state* directory, depending what resource is being managed.

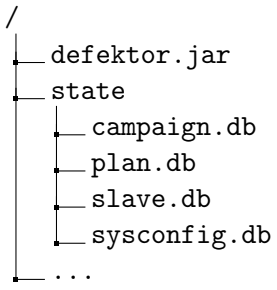


Figure 6.2: Defektor state directory tree.

6.2.5 Plugin Management System

Defektor leverages a plugin system to extend its capabilities, either in the systems it can connect, the fault injectors it can invoke, or the data collectors it can employ. Thus, *Defektor* must have a built-in plugin management system. This management consists in three steps: installation, loading, and instantiation.

Installation

Defektor will be distributed with some available plugins, which will be listed later in this report. Nevertheless, if the practitioner develops himself a plugin, the installation process is very trivial. After he packages his plugin code in a *.jar* file, the practitioner should place it in the corresponding */plugins/lib* subfolders. To give a more accurate picture of *Defektor* directory tree, we display it graphically in Figure 6.3.

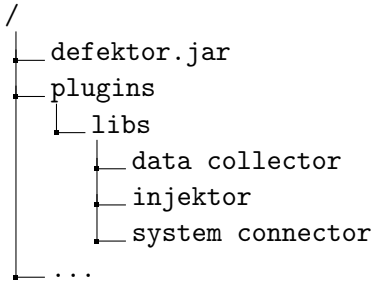


Figure 6.3: Defektor plugin directory tree.

After the new plugins have been correctly installed in the appropriate folders, *Defektor* must be restarted for them to be loaded. The loading process will be addressed in the next subsection.

Loading

The plugin management system's loading phase is responsible to load the installed plugins into the *Defektor* environment. As of right now, *Defektor* only do performs this process when it boots up. We plan for future work that *Defektor* may be capable of loading plugins without rebooting the application.

Defektor starts by scanning the directories where plugins are installed and inspects all the *.jar* files. This *.jar* files must meet two requirements: have the implemented plugin Java class and a *JAR Manifest File* containing the plugin name (*PluginName*) and the canonical name of the plugin Java class (*PluginClass*).

Listing 5 shows a what could be a manifest file for the *Kubernetes System Connector* plugin.

```
1 Manifest-Version: 1.0
2 PluginName: kubernetes
3 PluginClass:
  ↪ pt.uc.sob.defektor.plugins.sysconnector.kubernetes.KubernetesSystemPlug
```

Listing 5: Kubernetes plugin MANIFEST.MF.

The canonical name (the name of the class along with the package) is needed so *Defektor* knows what class to look for inside the *.jar* package. Once that plugin class is properly loaded, *Defektor* is ready to instantiate one object of the added plugin. This instantiation process will be addressed in the next subsection.

Instantiation

The plugin management system must be capable of instantiating objects. Thus, *Defektor* can instantiate its loaded plugins by their *PluginName*. In fact, when a *Plan* is added, the desired *System Connector*, *Injektor* and *Data Collector* names must exactly match the *PluginName* assigned to each loaded plugin.

6.3 dfk

A previously referenced, *dfk* is a Python command line client used to control *Defektor*. We employed *click* which is a Python package that allows one to create a command line interface in composable way [56]. To make *dfk* interact with *Defektor*, we created package named *defektor api* that is generated by Swagger given *Defektor* OpenAPI specification (available in Appendix 9). This REST client package gives *dfk* the means to access *Defektor* endpoints.

To setup *dfk*, the practitioner should have Python and pip installed. To aid in *dfk* installation process, we deployed it PyPI [18]. Thus, to install *dfk*, user must execute the

following command:

$ pip install dfk

After the installation process is completed, the pratictioner can access *dfk* main menu. This main menu is displayed in Figure 6.4.

```
➜  ~ dfk
Usage: dfk [OPTIONS] COMMAND [ARGS]...

  🐳 dfk - Write once, run away!

Options:
  --help  Show this message and exit.

Commands:
  campaign     🎯 Injection campaigns.
  easter-egg   🐰🥚 The easter-egg command.
  ijk          💉 Fault Injektors.
  login        🔑 defektor login.
  plan         📜 Fault injection plans.
  slave        👷 Slave machine management.
  sysconfig    💻 System configuration.
```

Figure 6.4: *dfk* main menu.

We can observe that it contains seven different commands. To start using *dfk*, the practitioner should provide the *Defektor* Uniform Resource Locator (URL), otherwise none of the other commands can be executed. To do so, the user must run the command:

$ dfk login <defektor_ url>.

To give a deeper demonstration of *dfk*, we will show how to add and list *Slave* machines. Figure 6.5 portrays the available commands to manage *Slave* machines.

```
➜  ~ dfk slave
Usage: dfk slave [OPTIONS] COMMAND [ARGS]...

  👷 Slave machine management.

Options:
  --help  Show this message and exit.

Commands:
  add     Add a slave machine to run the work-loads.
  list    Lists slaves.
  remove  Removes a slave machine.
```

Figure 6.5: *dfk* slave menu.

Figure 6.6 displays adding two new *Slave* machines.

```
➜ ~ dfk slave add '192.168.1.1' 'slave_01' '/home/goncalo/.ssh/id_rsa_slave_01'
{'address': '192.168.1.1',
 'credentials': {'key': '/home/goncalo/.ssh/id_rsa_slave_01',
                 'username': 'slave_01'},
 'id': 'ca19a870-8136-48d8-9394-b77077f15e99',
 'port': 22}
➜ ~ dfk slave add '192.168.1.2' 'slave_02' '/home/goncalo/.ssh/id_rsa_slave_02'
{'address': '192.168.1.2',
 'credentials': {'key': '/home/goncalo/.ssh/id_rsa_slave_02',
                 'username': 'slave_02'},
 'id': '9e401b41-8901-404d-8400-ab75a66cd330',
 'port': 22}
```

Figure 6.6: *dfk* slave add.

Figure 6.7 shows listing the available *Slave* to *Defektor*. In this case, the response contains the newly added *Slave* machines.

```
➜ ~ dfk slave list
[{'address': '192.168.1.1',
 'credentials': {'key': '/home/goncalo/.ssh/id_rsa_slave_01',
                 'username': 'slave_01'},
 'id': '82891ac6-7655-447d-8e4d-ac8250be1b78',
 'port': 22}, {'address': '192.168.1.2',
 'credentials': {'key': '/home/goncalo/.ssh/id_rsa_slave_02',
                 'username': 'slave_02'},
 'id': '4e4c9c36-6e3b-43ba-bac8-482525cf7bad',
 'port': 22}]
```

Figure 6.7: *dfk* slave list.

6.4 Plugins

In this section we present all plugins that were developed for *Defektor*.

6.4.1 The *System Connector* Plugins

This section presents the developed *System Connector* plugins. It is important to note that the plugins showed in the next two subsections all share the methods presented in section 5.5.1. However, because these plugins interact with a variety of systems, new system-specific methods must be built to offer *Injektors* with methods that are appropriate for their purposes.

Two plugins were developed to attend two different system types: *Kubernetes* and SSH-enabled linux machines.

Kubernetes

Kubernetes is a widely known container orchestrator in the distributed computing industry, well suited for microservice-based applications [11]. Additionally, *Kubernetes* has complementary tools, like service meshes, which provide control and observability mechanisms. This is the case of *Istio* [24], an open-source service mesh, which has been one

of the most used service meshes both for evaluation and production purposes [12]. Among other functionalities, *Istio* provides two primitive fault injection mechanisms that we may leverage for our own case: HTTP delay and HTTP abort faults [25]. Thus, we considered that would be relevant to develop some *Injektors* and a *Kubernetes System Connector* plugin to demonstrate our solution.

This *Kubernetes System Connector* plugin provides two methods in order to establish a bridge between the *Target Application* and its supported *Injektors*:

- applyManifest: is equivalent to a *kubectl apply -f <filename>* command [33] which accepts an *.yaml* or *.json* file in order to create or update a resource in the cluster where the *Target Application* is deployed.
- removeManifest: is equivalent to a *kubectl delete -f <filename>* command [34] which deletes an existing resource in the cluster.

As the configuration to access a *Kubernetes* cluster can contain an excessive number of parameters, we opted for the traditional *kubectl* mechanism [37]. Thus, *Defektor* externalized this configuration, not overloading the system configuration.

Two plugins were developed for this *System Connector*: HTTP Delay and HTTP Abort. These will be further explained in the subsections 6.4.2 and 6.4.2, respectively.

SSH-enabled linux machine

Linux machines are the most used to host servers and applications, including microservice applications. For that reason, we decided that it would be relevant to develop a *System Connector* capable of controlling any Linux instance via SSH.

This plugin offers one method to enable its supported *Injektors* to send commands to *Target Application*:

- sendSSHCommand: redirects one or more *string* commands from the *Injektor* to the *Target Application*.

In order to establish a SSH connection, some parameters must be passed to this plugin object via its *configuration* method. The Listing 6 displays the configuration parameters needed to be passed to *Defektor*.

```
configs:
  username: "goncalo"
  host: "192.168.1.1"
  port: 22
  privateKey: "~/.ssh/id_rsa"
systemType:
  name: "ssh-enabled-linux-machine"
```

Listing 6: SSH-enabled linux machine configuration parameters.

Two plugins were developed for this *System Connector*: Instance Shutdown and Process Terminator. These will be further explained in the subsections 6.4.2 and 6.4.2, respectively.

6.4.2 The *Injektor* Plugins

This section presents the developed *Injektors* plugins. It is important to note that the plugins showed in the next two subsections all implement the methods presented in section 5.5.2.

Four plugins were developed capable of inject faults in two different system types: HTTP Abort and HTTP Delay for *Kubernetes* and *Instance Shutdown* and *Process Terminator* for SSH-enabled linux machines.

HTTP Delay

This fault injection inserts a delay in HTTP packets for a target service. For example, if this failure is injected in Service B and Service A requests something from Service B, there is a probability $P(D)$ that Service B will delay its response x seconds. The practitioner may parameterise both, the probability and x in the injection plan as it is displayed in Listing 7.

```
ijk:
  name: httpdelay
  params:
    namespace: robot-shop
    service: cart
    fixedDelay: 5s
    faultOccurrence: 100
```

Listing 7: HTTP Delay *injektor (ijk)* parameters.

This sample parameterisation describes that the service [39] *cart* deployed in the *robot-shop* namespace [35] has 100% probability to delay the response to its ingress requests by five seconds.

HTTP Abort

This fault injection enables the possibility to insert an error in HTTP packets destined for a specific service. For instance, if this failure is introduced in Service B and Service A requests something from Service B, there is a customisable probability that Service B will respond with a configurable HTTP code. For example, if we want to cause a failure in the target service, sending `5xx` HTTP status code as a response to a request will be handled as an error by the service that requested the faulty service. The practitioner may configure both the failure probability and the HTTP status in the injection plan as it is portrayed in Listing 8.

```yaml
ijk:
  name: httpabort
  params:
    namespace: robot-shop
    service: ratings
    httpStatus: 555
    faultOccurrence: 75
```

Listing 8: HTTP Abort *injektor (ijk)* parameters.

This sample parameterisation describes that the service *cart* deployed in the *robot-shop* namespace has 100% probability to delay the response to its ingress requests by five seconds.

Instance Shutdown

The goal of this plugin is to terminate any SSH-enabled Linux Machine. Some requirements must be met *i.e.*, the user that was connected via SSH must have permissions to execute some *sudo* commands.

Process Terminator

The main purpose of this plugin is to kill a process. It can take either a Process IDentification (PID) or a process name as the argument. If it receives the former, as it is displayed in Listing 9, the plugin kills the process corresponding the specified PID. If it receives the latter, as it is portrayed in Listing 10, the plugin kills all the processes that match the given process name.

```yaml
ijk:
  name: processterminator
  params:
    pid: 9999
```

Listing 9: Process Terminator PID parameterisation.

```yaml
ijk:
  name: processterminator
  params:
    processName: "find"
```

Listing 10: Process Terminator process name parameterisation.

6.4.3 The *Data Collector* Plugin

This sections presents the developed *Data Collector* plugins. It is important to note that the only plugin that will be introduced implements the methods presented in section 5.5.1.

Jaeger

This *Data Collector* is capable of collecting traces of one or more services from the Jaeger software. This plugin only expects two arguments, as it displayed in Listing 11.

```
dataCollector:
  name: jaeger
  params:
    host:  http://www.example.com:16686
    service: all
```

Listing 11: Jaeger parameterisation.

The first parameter, *host*, specifies where the Jaeger system exposes its API. The second parameter, *service*, specifies which service's traces should be collected. As of right now, *Defektor* can work with a single service—specifying the exact name of the service—or with all services—assigning this parameter with the keyword 'all'. For future work, we plan to also accept a [vector] of services.

Chapter 7

Experimental Results and Analysis

We now present and the experiment setup and will analyse the results of the experiments. This results serve somewhat as a *Defektor* proof concept.

7.1 Target Application

The design of *Defektor* gives the practitioner the ability to perform consistent fault injection campaigns on different microservice-based applications. As we mentioned before, to achieve this goal in a fully generic manner, *Defektor* resorts to plugins, to interface with the supporting infrastructure and platforms, collect data, and to implement fault injection mechanisms.

To run our experiments, we needed an application compatible with the plugins we had already created. We thus required an open-source Kubernetes-based microservice application with the *Istio* service mesh enabled. We found two applications that met these criteria: *Stan's Robot Shop*, by *Instana* [22] and *Bookinfo Aplication*, by *Istio* [23]. The latter was developed with the goal of serving as a sample application for testing out all of *Istio*'s features. It is, however, a very basic application with only four services bearing little resemblance to any real-world product.

Stan's Robot Shop, on the other hand, is an e-commerce sample application with a higher degree of complexity and a closer resemblance to a real-world deployable product. It depends on eleven containerized services running on Kubernetes: *cart*, *catalogue*, *dispatch*, *mongodb*, *mysql*, *rabbitmq*, *ratings*, *redis*, *shipping*, *user*, and *web*. These are in charge of handling user actions, such as logging in, browsing the catalogue, adding items to the cart, paying, and shipping the desired products. Given the size and contents of the application, we selected *Stan's Robot Shop* to perform the experimental evaluation. Furthermore, *Instana* provides a load generator to exercise its own application. We can take advantage of this generator for our own experiments.

7.2 Experiment Setup

For our experiment setup, we selected the two *Kubernetes* injektors plugins: HTTP Delay and HTTP Abort because the chosen *Target Application* is deployed in *Kubernetes*.

For the HTTP Delay fault, we want to analyse what happens to the average response

time of the server, while for the HTTP Abort fault, we want to analyse what happens to the failure rate and if this fault propagates to other services.

In the HTTP Delay and HTTP Abort experiments we target the *cart* service. The time for each run (golden run and fault injection run) was set to 600 seconds and the load generator was configured with one client exercising the system, by performing arbitrary tasks in the e-commerce website.

For each experiment setup, we will display the injektion Plan that produced those results.

7.3 Results

In this section we present the results that demonstrate that *Defektor* can, in fact, manage a fault injection campaign with its available plugins.

7.3.1 HTTP Delay

To start this experiment, we started the injection campaign described in the *Plan* displayed in Listing 12. This plan sets 600 seconds for the golden run and the fault injection. During the fault injection run, a five seconds delay is injected in all *cart* service's ingress requests with a fault activation probability of 100%.

```
name: DelayCart5sec100Percent
system:
  name: kubernetes
injektions:
- replicas: 1
  ijk:
    name: httpdelay
    params:
      namespace: robot-shop
      service: cart
      fixedDelay: 5s
      faultOccurrence: 100
  workload:
    image:
      user: robot-shop
      name: rs-load
      tag: latest
    env:
        host: http://system.example.com/
        numClients: 10
        silent: 1
        error: 0
    slaves: 1
    replicasPerSlave: 1
    duration: 600
  dataCollector:
    name: jaeger
    params:
      host:  http://www.example.com:16686
      service: all
```

Listing 12: Injection plan used in to generate results in Figure 7.1.

Figure 7.1 shows the average response time of the entire system when the above *Plan* was injected.

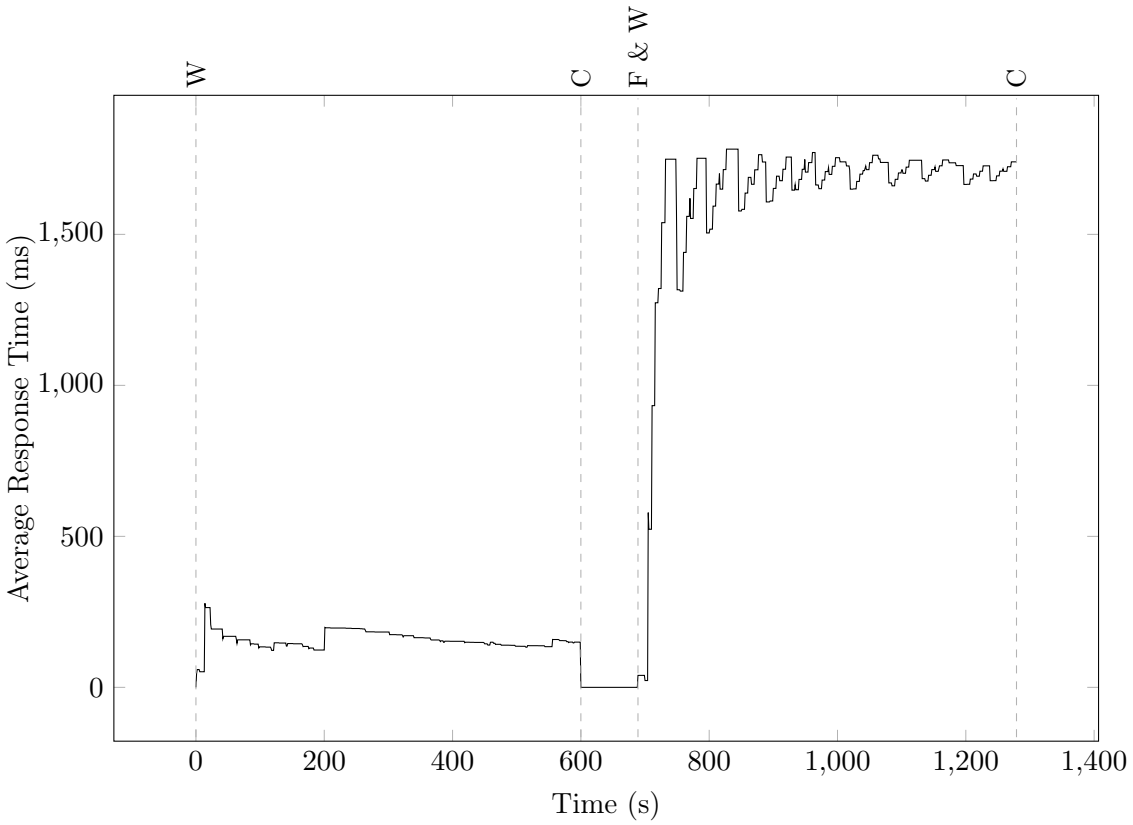


Figure 7.1: Sample HTTP Delay injection campaign, with typical injection stages: Run **(W)**orkload; **(C)**ollect Data; Inject **(F)**ault.

It starts the load generator at $x = 0$, exercising the target application. At $x = 600$, *Defektor* stops the load generator and cools down the system to collect data without interference. The time interval $x \in [0, 600]$ represents the golden run. At $x \in]600, 689]$ the system uses the *Jaeger* data collector plugin to extract the traces and metrics generated by the golden run and store the data in *Jaeger* [27]. Due to the infrastructural limitations on writing and reading this data on storage, this process takes more time than usual.

The time interval $x \in]689, 1289]$ represents the fault injection run. The HTTP Delay injection starts at $x = 690$. From this instant until $x = 1289$ the system is again exercised and the *cart* service will take exactly 5 seconds to respond every single request it receives. During this run, we can observe that the average response time of the entire application dramatically increases, indicating that the failure was successfully deployed.

To determine the impact of a slow service on the overall application or to exercise a load balancer or some fault-tolerance mechanism, one could exercise varying levels of delay in this same *cart* service. This would involve carrying out multiple experiments, with different fault activation probabilities. We set these fault activation probabilities to {0%, 25%, 50%, 75%, 100%}, still in the *cart* service, and display the results in Figure 7.2.

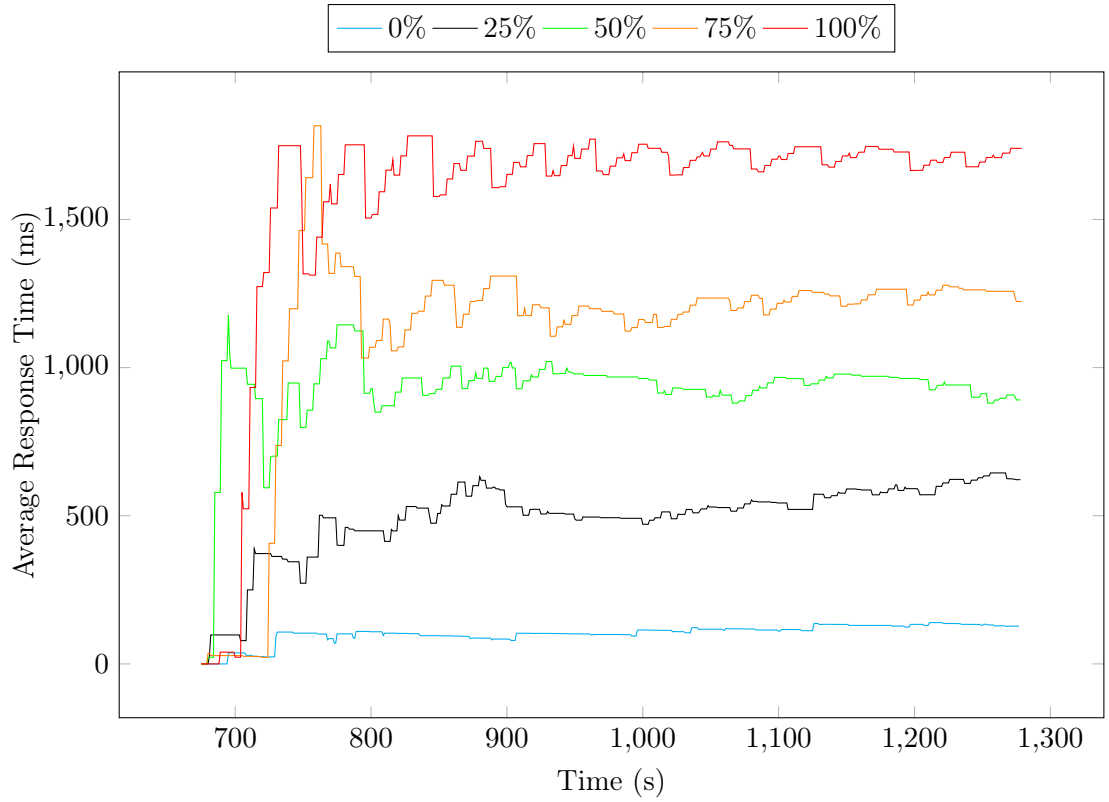


Figure 7.2: HTTP Delay injections campaigns for different failure activation probabilities.

For visualization purposes, we omitted the golden run phase in all experiments and kept only the fault injection run in the interval $x \in [680, 1280]$ (for a total of 10 minutes per run). The plots show that the higher the fault activation probability is, the higher the total average response time is. This demonstrates that *Defektor* is capable of injecting this fault with varying degrees of severity in the application.

7.3.2 HTTP Abort

We now discuss the experiments with HTTP Abort faults. These faults affect 100% of the requests to the *cart* service. We study *Defektor*'s capability to manage an HTTP Abort fault campaign and its impact on the system's overall failure rate, whenever a service is responding with HTTP error status codes to other services.

To start this experiment, we started the injection campaign described in the *Plan* displayed in Listing 13. This plan sets 600 seconds for the golden run and the fault injection. During the fault injection run, an 555 HTTP code is injected in all *cart* service's ingress requests with a fault activation probability of 100%.

```yaml
name: DelayCart5sec100Percent
system:
  name: kubernetes
injektions:
- replicas: 1
  ijk:
    name: httpabort
    params:
      namespace: robot-shop
      service: cart
      httpStatus: 555
      faultOccurrence: 100
  workload:
    image:
      user: robot-shop
      name: rs-load
      tag: latest
    env:
        host: http://system.example.com/
        numClients: 10
        silent: 1
        error: 0
    slaves: 1
    replicasPerSlave: 1
    duration: 600
  dataCollector:
    name: jaeger
    params:
      host:  http://www.example.com:16686
      service: all
```

Listing 13: Injection plan used in to generate results in Figure 7.3.

Figure 7.3 portrays an injection campaign sample (simple moving average n=30). This figure shows the number of requests the application serves per second. We may compare Figure 7.3 to Figure 7.1, as it covers the entire experiment, first in the golden run, then in the faulty run. It starts the load generator at $x = 0$, exercising the target application in the golden run. At $x = 600$ *Defektor* stops the load generator. At $x \in]600, 682]$ the system uses the *Jaeger* data collector plugin, to extract the traces and metrics generated by the golden run. The time interval $x \in]682, 1282]$ represents the fault injection run. The HTTP Abort injection is injected at $x = 683$. From this moment until $x = 1282$ the system is again exercised and the *cart* service returns the 555 HTTP status code for 100% of the requests it receives. During the golden run, we can observe that the entire system does not have failures, but that these start as soon as the abort failure is injected.

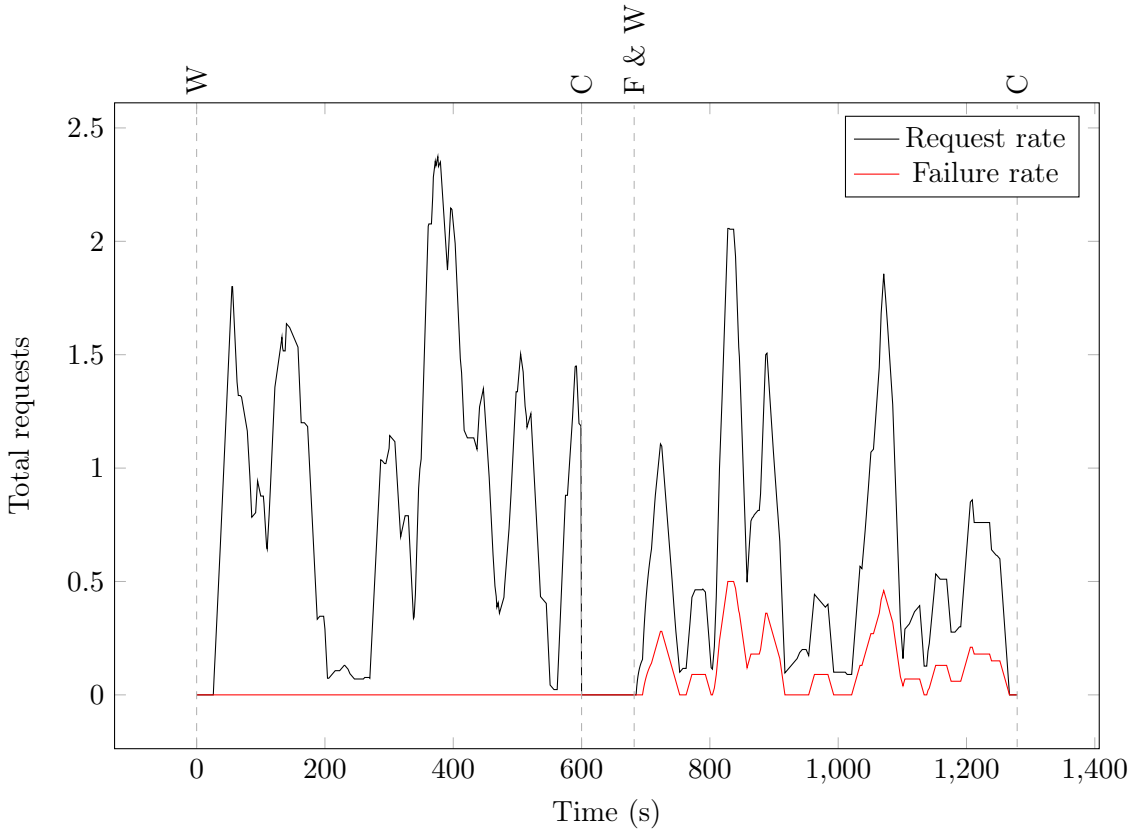


Figure 7.3: Sample HTTP Abort injection campaign, with typical injection stages: Run **(W)**orkload; **(C)**ollect Data; Inject **(F)**ault.

To better understand the impact of service failures (i.e., services returning an 5xx *status* code), practitioners could vary the fault activation probability. We carried out four experiments with fault activation probabilities of {25%, 50%, 75%, 100%} in the *cart* service, displayed in Figure 7.4. This figure shows, for each probability of *cart* service failure, the quartiles, median, maximum, minimum and outliers. An experiment like this demonstrates the dependency of the overall application on a specific service, for a given mixture of requests. As our experiment revealed, the observed failure rate is not linear with respect to the fault activation probability. Given the set M of all mean error rates M_a, for a given activation probability a, if we normalise around the 100% activation probability or $a = 1$, following Equation 7.1, we get $\{36.8\%, 66.1\%, 86.4\%, 100\%\}$, shown as a percentage. This step is necessary because we are injecting a single service, *cart*, but the workload exercises the whole system, diluting the failure rate.

$$M'_a = \frac{M_a}{max(M)}, \quad \forall\, a \tag{7.1}$$

This shows that for activation probabilities under 100% we see a higher failure rate than expected, which can be explained by failure propagation. As an example, a product is added to the cart and then another request is made that depends on it; that new request now has a reduced probability of success, as it is conditioned by the success of the former *i.e.*, it might fail as the result of the injected fault or it might fail because the insertion request failed, and the product is not on the cart.

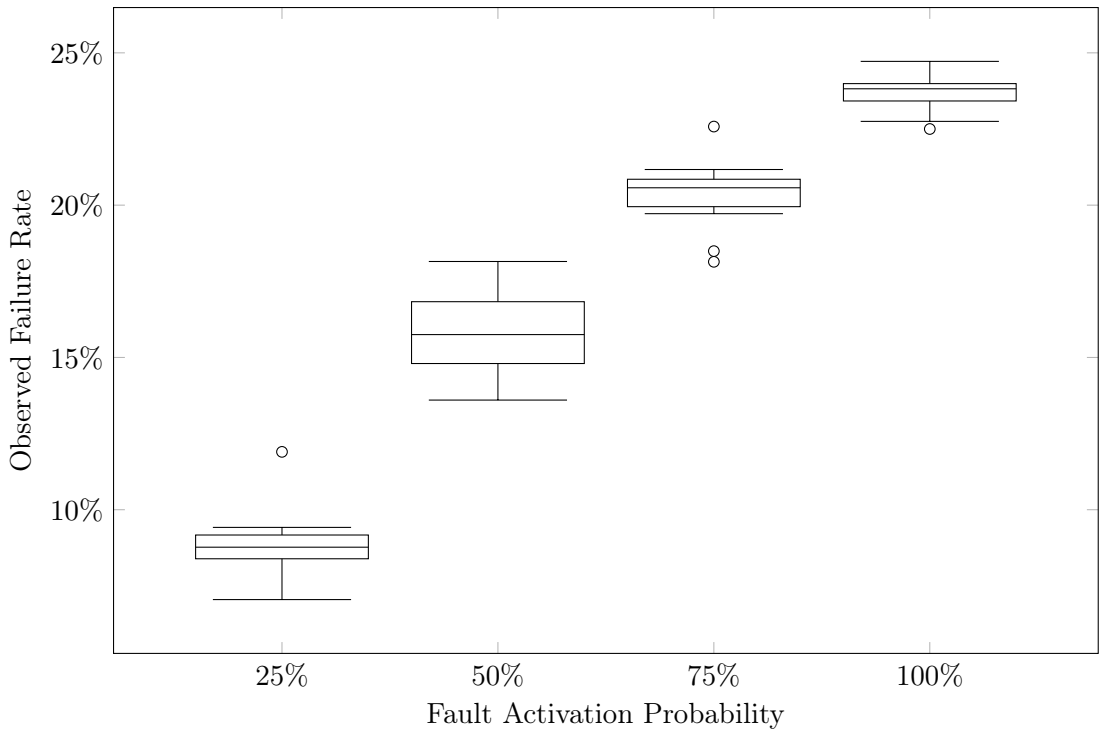


Figure 7.4: Observed system failure rate, when *cart* service is injected with HTTP Abort fault with different activation probability.

This page is intentionally left blank.

Chapter 8

Planning

In this chapter an analysis will be made of the planning of this project. For now, only the first semester will be approached with a describing planning analysis.

8.1 First Semester Planning

This section will analyse the tasks planned and developed during the first semester of this project. To support it, a Gantt diagram, in Figure 8.1 will then be presented containing information on the planned tasks and the tasks performed during the first semester.

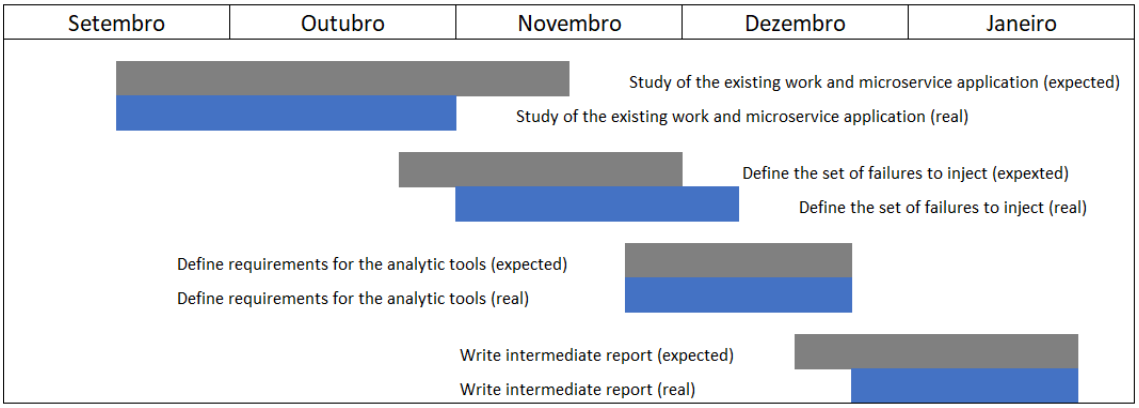


Figure 8.1: First semester Gantt diagram

In the diagram shown in Figure 8.1, it is possible to find, in grey, the **expected** tasks and, in blue, the corresponding tasks that were **real**istically performed. The first half of the project started September 16, 2020 and it has its ending in 18 January, 2021.

During this time four main tasks were performed:

- Study of the existing work and microservice application;
- Define the set of failures to inject;
- Define requirements for the analytic tools;
- Write intermediate report;

The first task was mainly about getting used to terminologies and tools of microservices and distributed systems. It was used *Kubernetes* to overcome this steep learning process. However, due some background with other container orchestrators, that task was accomplished earlier that was planned.

As the first task was about to be finished, an investigation about chaos engineering and distributed traces' tools and techniques were performed. The duration was similar to the one planned. Yet, there was no overlay between the first and second tasks.

The identification of requirements and the designing of the architecture were done in simultaneous with the second task. This was because December was a month with a busy agenda in terms of assignments. Therefore, there was the need to try to fit this investigation part and still learning about the tooling necessary for the project.

Lastly, the intermediate report was started to be written later than it was expected to the same reasons enunciated in last paragraph. Nevertheless, all the tasks were completely accomplished, yet with some deviations according to what was the initial planning.

8.2 Second Semester Planning

This section will analyse the tasks planned and developed during the second semester of this project. To support it, a Gantt diagram, presented in Figure 8.2, contains information on the planned tasks and the tasks performed during the second semester.

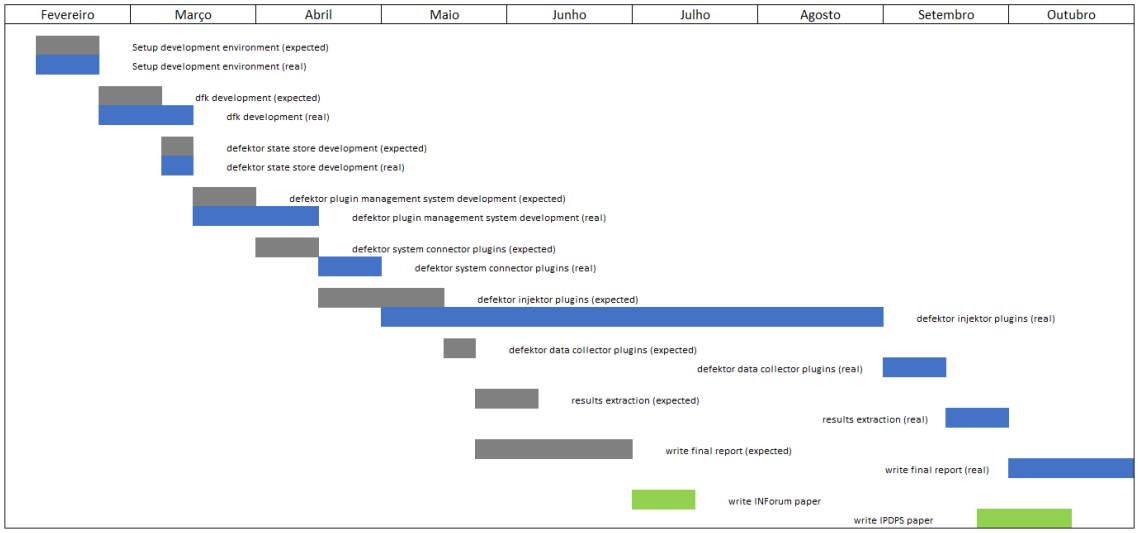


Figure 8.2: Second semester Gantt diagram

In the diagram shown in Figure 8.2, it is possible to find in grey the **expected** tasks, in blue the corresponding tasks that were **real**istically performed, and in green the tasks that were not planned but were completed due to the delayed report submission. The first half of the project started February 12, 2021 and it has its ending in 31 of October, 2021.

During this time eleven main tasks were performed:

- Setup development environment
- Development dfk
- Development Defektor State Store

- Development Defektor Plugin Management System
- Development system connector plugins
- Development injektors
- Development data collectors
- Extract results
- Write INForum paper
- Write IPDPS paper
- Write final report

The first task was mainly about setting up the IDE for our project and deploy and configure a *Kubernets* cluster to test some *Defektor* functionalities during their development.

The next three tasks were about to develop the three main *Defektor* components: dfk, state store, and plugin management system.

After these tasks were performed, the plugins started to be developed. Here we can note that some problems appeared. We can highlight that the task "Development injektors" took much longer time than expected. This circumstance was beyond our control, since we lost the connection to the infrastructure where our *Kuberentes* cluster was deployed. It took several weeks to find an alternative. However, the alternative we found had too many security constraints that made the development of some features impossible. Finally, we had the connection back to the first infrastructure and resumed the work normally.

Since the thesis submission was delayed, we decided it would be interesting to write two papers.

Lastly, the final report was started to be written. Nevertheless, all the tasks were completely accomplished, yet with some deviations according to what was the initial planning.

This page is intentionally left blank.

Chapter 9

Conclusion and Future Work

This Chapter aims to give some reflections on what we did, what was developed during this project, and what future work seems to be promising.

It was crucial to get some background knowledge and review the literature about fault injection, microservices, chaos engineering, and distributed tracing. We learnt that microservices tend to be complex in terms of their telemetry and predicting behavior when a failure event occurs in one specific service. Thus, two disciplines could be employed to mitigate these problems: distributed tracing—for telemetry—, and chaos engineering—for predicting behavior in a failure occurrence.

In what regards development, we presented *Defektor*, a tool for fault injection campaign management, its requirements, main architectural drivers, and its architecture. In addition, we presented a experimental setup and results analysis of its application to a microservice-based application.

By using plugins, *Defektor* is extensible, thus being able to target a large spectrum of systems, including microservice architectures, a paradigm where industry leaders rely heavily on fault injection for resiliency testing. *Defektor* departs from the previous state of the art, by automating the fault injection workflow, reducing the human element, and leveraging a plugin-based extension system that allows integration with third-party tools, speeding up data collection, and ensuring consistency among runs.

So far, for validation purposes, we have implemented two *System Connectors*, for *Kubernetes* and SSH-enabled Linux machines, as well as four *Injektors*, Process Terminator, Machine Shutdown, HTTP Delay and HTTP Abort. The tool and plugins are currently available as open source in a public repository [68].

As future work, we plan to write *Injektor* plugins for state-of-the-art fault injectors as well as improve the syntax for accepting vector parameters (*e.g.*, `workers: [1, 10, 100]`), making it easy to succinctly describe multiple runs with different parameterisations. Furthermore, we will move towards making the plans more expressive, allowing, for example, the concurrent injection of multiple faults.

Bibliography

[1] J. Aidemark et al. "GOOFI: generic object-oriented fault injection tool". In: *2001 International Conference on Dependable Systems and Networks*. 2001, pp. 83–88. DOI: 10.1109/DSN.2001.941394.

[2] Apache. *Cassandra*. URL: https://cassandra.apache.org/ (visited on 01/17/2021).

[3] Apache. *Maven – Welcome to Apache Maven*. URL: https://maven.apache.org.

[4] J. Arlat, Y. Crouzet, and J.-C. Laprie. "Fault injection for dependability validation of fault-tolerant computing systems". In: *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*. 1989, pp. 348–355. DOI: 10.1109/FTCS.1989.105591.

[5] J. Arlat et al. "Fault injection for dependability validation: a methodology and some applications". In: *IEEE Transactions on Software Engineering* 16.2 (1990), pp. 166–182. DOI: 10.1109/32.44380.

[6] Simon Brown. *The C4 model for visualising software architecture*. URL: https://c4model.com/ (visited on 12/23/2020).

[7] Tammy Butow. *Chaos Engineering: the history, principles, and practice*. 2018. URL: https://www.gremlin.com/community/tutorials/chaos-engineering-the-history-principles-and-practice/ (visited on 12/30/2020).

[8] Canonical. *Ubuntu*. URL: https://ubuntu.com/.

[9] Joao Carreira, Henrique Madeira, and Joao Gabriel Silva. "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers". In: *IEEE Trans. Softw. Eng.* 24.2 (Feb. 1998), 125–136. ISSN: 0098-5589. DOI: 10.1109/32.666826. URL: https://doi.org/10.1109/32.666826.

[10] *Chaos Monkey, the tool that causes minor faults in order to prevent greater ones*. URL: https://www.bbvaapimarket.com/en/api-world/chaos-monkey-tool-causes-minor-faults-order-prevent-greater-ones/ (visited on 01/07/2021).

[11] Cloud Native Computing Foundation. *What is Kubernetes*. 2019. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (visited on 01/17/2021).

[12] *CNCF SURVEY 2019: Deployments are getting larger as cloud native adoption becomes mainstream*. 2019. URL: https://www.cncf.io/wp-content/uploads/2020/08/CNCF_Survey_Report.pdf.

[13] Docker. *docker images*. URL: https://docs.docker.com/engine/reference/commandline/images/.

[14] Docker. *What is a Container?* 2020. URL: https://www.docker.com/resources/what-container (visited on 01/15/2021).

[15] Nicola Dragoni et al. "Microservices: yesterday, today, and tomorrow". In: *Present and Ulterior Software Engineering*. Ed. by Manuel Mazzara and Bertrand Meyer. Springer, Sept. 2017. URL: https://hal.inria.fr/hal-01631455.

[16] Durães, J. and Vieira, M. and Madeira H. "Dependability Benchmarking of Web-Servers". In: *Lecture Notes in Computer Science*. Vol. 3219. SAFECOMP 2004. Berlin, Germany, 2004, pp. 297–310. ISBN: 978-3-540-23176-9. DOI: 10.1007/978-3-540-30138-7_25.

[17] Elastic. *What is Elasticsearch?* URL: https://www.elastic.co/what-is/elasticsearch (visited on 01/17/2021).

[18] Python Software Foundation. *PyPI*. URL: https://pypi.org/.

[19] Martin Fowler and James Lewis. *Microservices, a definition of this new architectural term*. URL: https://martinfowler.com/articles/microservices.html/ (visited on 12/28/2020).

[20] *Git*. URL: https://git-scm.com/.

[21] Romana Gnatyk. *Microservices vs Monolith: which architecture is the best choice?* URL: https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/ (visited on 12/28/2020).

[22] Instana. *Robot Shop: Sample Microservice Application*. 2021. URL: https://github.com/instana/robot-shop.

[23] Istio. *Bookinfo Application*. 2021. URL: https://istio.io/latest/docs/examples/bookinfo/.

[24] Istio. *Docs*. URL: https://istio.io/latest/docs/ (visited on 01/14/2021).

[25] Istio. *Fault injection*. URL: https://istio.io/latest/docs/tasks/traffic-management/fault-injection/.

[26] Ravishankar K. Iyer. "Experimental Evaluation". In: *Proceedings of the Twenty-Fifth International Conference on Fault-Tolerant Computing*. FTCS'95. Pasadena, California: IEEE Computer Society, 1995, 115–132. ISBN: 0818671467.

[27] Jaeger. *Architecture — Jaeger documentation*. URL: https://www.jaegertracing.io/docs/1.21/architecture/ (visited on 01/17/2021).

[28] Jankotech. *MapDB*. URL: https://mapdb.org/.

[29] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham. "FERRARI: a flexible software-based fault and error injection system". In: *IEEE Transactions on Computers* 44.2 (1995), pp. 248–260. DOI: 10.1109/12.364536.

[30] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham. "FERRARI: a tool for the validation of system dependability properties". In: *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*. 1992, pp. 336–344. DOI: 10.1109/FTCS.1992.243567.

[31] W.-I. Kao, R.K. Iyer, and D. Tang. "FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults". In: *IEEE Transactions on Software Engineering* 19.11 (1993), pp. 1105–1118. DOI: 10.1109/32.256857.

[32] J. Karlsson et al. "TWO FAULT INJECTION TECHNIQUES FOR TEST OF FAULT HANDLING MECHANISMS". In: *1991, Proceedings. International Test Conference*. 1991, pp. 140–. DOI: 10.1109/TEST.1991.519504.

[33] Kubernetes. *Kubectl Reference Docs (kubectl apply)*. URL: https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#apply.

[34] Kubernetes. *Kubectl Reference Docs (kubectl delete)*. URL: https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#delete.

[35] Kubernetes. *Namespaces*. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.

[36] Kubernetes. *Nodes - Kubernetes*. 2019. URL: https://kubernetes.io/docs/concepts/architecture/nodes/ (visited on 01/15/2021).

[37] Kubernetes. *Organizing Cluster Access Using kubeconfig Files*. URL: https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/.

[38] Kubernetes. *Pods - Kubernetes*. 2020. URL: https://kubernetes.io/docs/concepts/workloads/pods/ (visited on 01/15/2021).

[39] Kubernetes. *Service*. URL: https://kubernetes.io/docs/concepts/services-networking/service/.

[40] Litmus. *ChaosHub*. URL: https://hub.litmuschaos.io/ (visited on 01/14/2021).

[41] Litmus. *Getting Started with Litmus · Litmus Docs*. URL: https://docs.litmuschaos.io/docs/getstarted/ (visited on 01/14/2021).

[42] Juval Lowy. *Programming WCF Services*. 2010, pp. 543–553. ISBN: 978-0-596-80548-7.

[43] Madeira, Henrique and Rela, Mário Zenha and Moreira, Francisco and Silva, João Gabriel. "RIFLE: A General Purpose Pin-Level Fault Injector". In: *Proceedings of the First European Dependable Computing Conference on Dependable Computing*. EDCC-1. Berlin, Heidelberg: Springer-Verlag, 1994, 199–216. ISBN: 3540584269.

[44] Laura Mauersberger. *Microservices: What They Are and Why Use Them*. URL: https://www.leanix.net/en/blog/a-brief-history-of-microservices/ (visited on 12/20/2020).

[45] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583.

[46] *Microservices Architecture – The Definitive Guide*. URL: https://www.leanix.net/en/microservices-architecture/ (visited on 12/20/2020).

[47] Moraes, R. and Duraes, J. and Barbosa, R. and Martins, E. and Madeira, H. "Experimental Risk Assessment and Comparison Using Software Fault Injection". In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. 2007, pp. 512–521. DOI: 10.1109/DSN.2007.45.

[48] *MySQL*. URL: https://www.mysql.com/ (visited on 01/14/2021).

[49] Heather Nakama. *Inside Azure Search: Chaos Engineering*. URL: https://azure.microsoft.com/en-us/blog/inside-azure-search-chaos-engineering/ (visited on 01/07/2021).

[50] Netflix. *Home - Chaos Monkey*. URL: https://netflix.github.io/chaosmonkey/ (visited on 01/14/2021).

[51] Netflix. *The Netflix Simian Army*. URL: https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116 (visited on 12/30/2020).

[52] Sam Newman. *Building Microservices*. 2015, p. 280. ISBN: 978-1-491-95035-7.

[53] OpenTracing. *The OpenTracing Semantic Specification*. 2020. URL: https://github.com/opentracing/specification/blob/master/specification.md (visited on 01/11/2021).

[54] OpenTracing. "What is Distributed Tracing?" In: URL: https://opentracing.io/docs/overview/what-is-tracing/.

[55] Claus Pahl et al. "Cloud Container Technologies: a State-of-the-Art Review". In: *IEEE Transactions on Cloud Computing* (2017). ISSN: 21687161. DOI: 10.1109/TCC.2017.2702586.

[56] Pallets. *click*. URL: https://click.palletsprojects.com/en/8.0.x/.

[57] Cloud Platform. "Distributed Tracing". In: July (2015). URL: https://microservices.io/patterns/observability/distributed-tracing.html.

[58] Chaos I N Practice and Advanced Principles. "Principles of chaos engineering". In: (2018), pp. 1–2. URL: https://principlesofchaos.org/.

[59] Chris Richardson. *What are microservices?* URL: https://microservices.io/ (visited on 12/20/2020).

[60] Z. Segall et al. "FIAT - Fault injection based automated testing environment". In: *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'*. 1995, pp. 394–. DOI: 10.1109/FTCSH.1995.532663.

[61] Benjamin H. Sigelman et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010. URL: https://research.google.com/archive/papers/dapper-2010-1.pdf.

[62] snyk. *JVM Ecosystem Report 2020*. URL: https://snyk.io/wp-content/uploads/jvm_2020.pdf.

[63] *Spinnaker*. URL: https://spinnaker.io/ (visited on 01/14/2021).

[64] Spring. *Spring Boot*. URL: https://spring.io/projects/spring-boot.

[65] D.T. Stott et al. "NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors". In: *Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000*. 2000, pp. 91–100. DOI: 10.1109/IPDS.2000.839467.

[66] Yevgeniy Sverdlik. *Facebook Turned Off Entire Data Center to Test Resiliency*. 2014. URL: https://www.datacenterknowledge.com/archives/2014/09/15/facebook-turned-off-entire-data-center-to-test-resiliency (visited on 01/07/2021).

[67] Swagger. *Documentation*. URL: https://swagger.io/docs/.

[68] *SysOBs/defektor: A fa[ult | ilure] injector for μServices*. URL: https://github.com/SysOBs/defektor.

[69] *SysOBs/defektorOpenAPISpec*. URL: https://github.com/SysOBs/defektorOpenAPISpec.

[70] *SysOBs/dfk: A command-line tool for defektor*. URL: https://github.com/SysOBs/dfk.

[71] Johannes Thönes. *Microservices*. 2015. DOI: 10.1109/MS.2015.11.

[72] Vieira, Marco and Madeira, Henrique. "A Dependability Benchmark for OLTP Application Environments". In: *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*. VLDB '03. Berlin, Germany: VLDB Endowment, 2003, 742–753. ISBN: 0127224424.

[73] *What Is Chaos Monkey? A Complete Guide for Engineers, DevOps & SREs*. URL: https://www.gremlin.com/chaos-monkey/ (visited on 12/30/2020).

[74] *What is REST*. URL: https://restfulapi.net/.

[75] Zipkin. *Architecture*. 2017. URL: https://zipkin.io/pages/architecture.html (visited on 01/17/2021).

This page is intentionally left blank.

Appendices

Appendix A

```yaml
openapi: 3.0.0
servers:
  - description: SwaggerHub API Auto Mocking
    url: https://virtserver.swaggerhub.com/jaimelive/defektorOpenAPISpec
    ↪ /1.0.0
info:
  description: defektor controll aplication interface
  version: 1.0.0
  title: defektor OpenAPI Specification
  contact:
    email: sob@dei.uc.pt
paths:
  /slave:
    get:
      summary: list slave machines
      operationId: slaveList
      description: List available slave machines
      responses:
        '200':
          description: slaves listed
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Slave'
    post:
      summary: add slave machine
      operationId: slaveAdd
      description: Add slave machine
      responses:
        '201':
          description: slave created
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Slave'
        '400':
          description: invalid input, object invalid
        '409':
          description: slave already exists
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Slave'
        description: Added Slave
```

```yaml
    delete:
      summary: delete all slaves
      operationId: slaveDeleteAll
      description: Delete all slaves
      responses:
        '200':
          description: all slaves deleted
  /slave/{slaveId}:
    get:
      summary: slave machine info
      operationId: slaveGet
      description: Get slave machine information
      parameters:
        - in: path
          name: slaveId
          description: Slave machine identifier
          required: true
          schema:
            type: string
            format: UUID
            example: d290f1ee-6c54-4b01-90e6-d701748f0851
      responses:
        '200':
          description: slave information
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Slave'
        '400':
          description: slave does not exist
    delete:
      summary: delete slave machine
      operationId: slaveDelete
      description: Delete slave machine
      parameters:
        - in: path
          name: slaveId
          description: Slave machine identifier
          required: true
          schema:
            type: string
            format: UUID
            example: d290f1ee-6c54-4b01-90e6-d701748f0851
      responses:
        '200':
          description: slave deleted
        '400':
          description: slave does not exist
  /target:
    get:
      summary: list targets
```

```
99        operationId: targetList
100       description: List available targets
101       responses:
102         '200':
103           description: slaves listed
104           content:
105             application/json:
106               schema:
107                 type: array
108                 items:
109                   $ref: '#/components/schemas/TargetType'
110   /target/{target}:
111     get:
112       summary: list target instances
113       operationId: targetGet
114       description: List available targets
115       parameters:
116         - in: path
117           name: target
118           description: target type
119           required: true
120           schema:
121             type: string
122       responses:
123         '200':
124           description: slaves listed
125           content:
126             application/json:
127               schema:
128                 type: array
129                 items:
130                   $ref: '#/components/schemas/Target'
131   /ijk:
132     get:
133       summary: list ijk (injektors)
134       operationId: ijkList
135       description: List available injektors
136       responses:
137         '200':
138           description: list of injektors
139           content:
140             application/json:
141               schema:
142                 type: array
143                 items:
144                   $ref: '#/components/schemas/Ijk'
145   /plan:
146     get:
147       summary: list plans
148       operationId: planList
149       description: List plans
```

```yaml
    responses:
      '200':
        description: list of plans
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/Plan'
  post:
    summary: add plan
    operationId: planAdd
    description: Add plan
    responses:
      '201':
        description: plan created
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Plan'
      '400':
        description: invalid input, object invalid
      '409':
        description: plan already exists
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Plan'
      description: Added Plan
  delete:
    summary: delete all plans
    operationId: planDeleteAll
    description: Delete all plan
    responses:
      '200':
        description: all plan deleted
/plan/validate:
  post:
    summary: validate plan
    operationId: planValidate
    description: Validate plan
    responses:
      '200':
        description: plan is valid
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Plan'
      '400':
```

```
201           description: invalid input, object invalid
202       requestBody:
203         required: true
204         content:
205           application/json:
206             schema:
207               $ref: '#/components/schemas/Plan'
208         description: Validated Plan
209   /plan/{planId}:
210     get:
211       summary: plan info
212       operationId: planGet
213       description: Get plan information
214       parameters:
215         - in: path
216           name: planId
217           description: Plan identifier
218           required: true
219           schema:
220             type: string
221             format: UUID
222             example: d290f1ee-6c54-4b01-90e6-d701748f0851
223       responses:
224         '200':
225           description: plan information
226           content:
227             application/json:
228               schema:
229                 $ref: '#/components/schemas/Plan'
230         '400':
231           description: plan does not exist
232     delete:
233       summary: delete plan
234       operationId: planDelete
235       description: Delete plan
236       parameters:
237         - in: path
238           name: planId
239           description: Plan identifier
240           required: true
241           schema:
242             type: string
243             format: UUID
244             example: d290f1ee-6c54-4b01-90e6-d701748f0851
245       responses:
246         '200':
247           description: plan deleted
248         '400':
249           description: plan does not exist
250   /system/config:
251     get:
```

```
252       summary: list system configs
253       operationId: systemConfigList
254       description: List available system configs
255       responses:
256         '200':
257           description: list of system configs
258           content:
259             application/json:
260               schema:
261                 type: array
262                 items:
263                   $ref: '#/components/schemas/SystemConfig'
264     post:
265       summary: Submit system and its configurations
266       operationId: systemTypeConfigure
267       description: Submits a system type and its configuration
268       requestBody:
269         required: true
270         content:
271           application/json:
272             schema:
273               $ref: '#/components/schemas/SystemConfig'
274         description: Configure system
275       responses:
276         '200':
277           description: submits a system type
278           content:
279             application/json:
280               schema:
281                 $ref: '#/components/schemas/SystemConfig'
282   /campaign:
283     get:
284       summary: list campaigns
285       operationId: campaignList
286       description: List campaigns
287       responses:
288         '200':
289           description: list of campaigns
290           content:
291             application/json:
292               schema:
293                 type: array
294                 items:
295                   $ref: '#/components/schemas/Campaign'
296   /campaign/{campaignId}:
297     get:
298       summary: campaign info
299       operationId: campaignGet
300       description: Get campaign information
301       parameters:
302         - in: path
```

```yaml
          name: campaignId
          description: Campaign identifier
          required: true
          schema:
            type: string
            format: UUID
            example: d290f1ee-6c54-4b01-90e6-d701748f0851
      responses:
        '200':
          description: campaign information
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Campaign'
        '400':
          description: campaign does not exist
    delete:
      summary: delete campaign
      operationId: campaignDelete
      description: Delete campaign
      parameters:
        - in: path
          name: campaignId
          description: Campaign identifier
          required: true
          schema:
            type: string
            format: UUID
            example: d290f1ee-6c54-4b01-90e6-d701748f0851
      responses:
        '200':
          description: campaign deleted
        '400':
          description: campaign does not exist
components:
  schemas:
    TargetType:
      required:
        - name
      properties:
        name:
          type: string
          example: container
    Target:
      required:
        - type
        - name
      properties:
        type:
          $ref: '#/components/schemas/TargetType'
        name:
```

```yaml
        type: string
        example: istio_ingress_2314234h21345
    SSHCredentials:
      required:
        - username
        - key
      properties:
        username:
          type: string
          example: debian
        key:
          type: string
          example: ooooohhhh this is a super secret private key...
    Slave:
      required:
        - address
        - port
        - credentials
      properties:
        id:
          type: string
          format: UUID
          example: d290f1ee-6c54-4b01-90e6-d701748f0851
        address:
          type: string
          example: example.org
        port:
          type: integer
          format: int32
          default: 22
          example: 22
        credentials:
          $ref: '#/components/schemas/SSHCredentials'
    Ijk:
      required:
        - name
        - params
      properties:
        name:
          type: string
          example: HoleyBoat
        params:
          type: array
          items:
            $ref: '#/components/schemas/KeyValue'
    DockerImage:
      properties:
        user:
          type: string
          example: sob
        name:
```

```
405           type: string
406           example: mangodb
407         tag:
408           type: string
409           example: latest
410     WorkLoad:
411       required:
412         - image
413       properties:
414         image:
415           $ref: '#/components/schemas/DockerImage'
416         cmd:
417           type: string
418           example: sh shesellsshellsbytheseashore.sh
419         env:
420           type: array
421           items:
422             $ref: '#/components/schemas/KeyValue'
423         replicas:
424           type: integer
425           format: int32
426           default: 1
427           example: 1
428         slaves:
429           type: integer
430           format: int32
431           default: 1
432           example: 1
433         duration:
434           type: integer
435           format: int32
436           default: 120
437           example: 120
438           description: Duration of the workload in seconds. If the
              ↪  container terminates earlier it gets restarted.
439     DataCollector:
440       properties:
441         name:
442           type: string
443         configs:
444           type: array
445           items:
446             $ref: '#/components/schemas/KeyValue'
447     Injektion:
448       required:
449         - ijk
450         - target
451       properties:
452         totalRuns:
453           type: integer
454         ijk:
```

```
        $ref: '#/components/schemas/Ijk'
      workload:
        $ref: '#/components/schemas/WorkLoad'
      target:
        $ref: '#/components/schemas/Target'
  Plan:
    required:
      - name
      - system
      - injektions
    properties:
      id:
        type: string
        format: UUID
        example: d290f1ee-6c54-4b01-90e6-d701748f0851
      name:
        type: string
        example: Order 66
      system:
        $ref: '#/components/schemas/SystemType'
      injektions:
        type: array
        items:
          $ref: '#/components/schemas/Injektion'
  SystemType:
    properties:
      name:
        type: string
        example: kubernetes
  SystemConfig:
    properties:
      configs:
        type: array
        items:
          $ref: '#/components/schemas/KeyValue'
      systemType:
        $ref: '#/components/schemas/SystemType'
  SystemTarget:
    required:
      - name
      - type
    properties:
      name:
        type: string
        example: kubernetes
      targetTypes:
        type: array
        items:
          $ref: '#/components/schemas/TargetType'
  KeyValue:
    properties:
```

```
506         key:
507           type: string
508         value:
509           type: string
510     Campaign:
511       properties:
512         id:
513           type: string
514           format: UUID
515         currentRun:
516           type: integer
517         totalRuns:
518           type: integer
519         status:
520           type: string
521         message:
522           type: string
```

Appendix B

Paper submitted to IPDPS is displayed in the next page.

Defektor: An Extensible Tool for Fault Injection Campaign Management in Microservice Systems

[Omitted for Double-blind Review]

***Abstract*—To achieve dependability, system designers often resort to fault-tolerance mechanisms. The evaluation of these mechanisms requires the observation of failures, which typically are rare events. To increase the failure rate, practitioners use fault injection techniques, leading to an increased occurrence of failures and allowing the assessment of the systems dependability properties. While many fault injection tools exist for this end, they are usually limited in scope, applicability and in their configuration abilities for microservice applications.**

We propose a generalist and extensible tool named "Defektor" capable of controlling a fault injection campaign on different types of applications, especially suited for microservice-based applications, compatible with different container orchestration technologies and different fault injection tools. The Defektor configuration follows a high-level approach, based on an injection campaign plan specifying the instructions for the Defektor operation and the parameters of the fault injection campaign. Defektor automates the entire workflow, consisting of defining the campaign plan, generating a workload, specifying and injecting the faults, and collecting data, aiding the experiment repeatability, improving the consistency of results, and saving a considerable amount of time.

***Index Terms*—Fault injection, Microservices, Cloud-native**

I. Introduction

Microservice-based architectures are currently a very relevant design strategy that breaks down applications into many small and focused components—the microservices. This paradigm offers many advantages including increased scalability, platform independence, load balancing and redundancy-based dependability. Each microservice is designed to provide a very specific functionality, thus being simpler, possibly better implemented.

Despite its many advantages, microservice-based architectures pose difficult problems related to verification and validation. Although each microservice individually may be very focused and well implemented, the increased interaction between many discrete components increases the possibility of integration issues. Also, because the system is now distributed among many loosely coupled components, fault localisation and causality analysis becomes very hard, making the evaluation the system dependability properties very hard.

Fault tolerance mechanisms are an important aspect of modern systems, to handle fault activation occurring in the operational phase and avoid or mitigate failures. Thus, during development, it is very important to understand the system behaviour when some components fail, in order to better design mitigation and recovery mechanisms. Because the activation of hidden faults and consequent failures are relatively rare (otherwise those fault would have been found and corrected), practitioners use fault injection techniques to increase the rate of fault activation in order to observe failures and be able to characterise the system behaviour.

Several large companies using microservice-based systems developed their own tools for fault injection, such as Simian Army [25]. These tools support different fault models and follow diverse injection strategies, but are mainly based on the idea of interfering with processes, resources, network connections, and containers to achieve the goal of causing failures.

These tools can help system developers to fine-tune fault detection, diagnosis and fault tolerance actuation. However, the entire process is, as far as we know, mostly manual, being up to developers to manualy execute many of the necessary steps, such as preparing the system, running it for a specified amount of time, collecting the data, and injecting faults or errors using specialised tools. Moreover, these tools are typically independent from one another and have heterogeneous interfaces and control logic, further increasing the difficulty and work necessary to run fault injection experiments. This is especially true in microservices, because their intrinsic diversity implies the need for more tools.

To automate the entire process and help integrating and dealing with the diversity of the tools involved, we propose the Defektor tool. Defektor is configurable for different types of systems, failures and data sources and can interface with diverse injectors. Defektor uses a high level language for experiment configuration where the user specifies the set of components, such as plugins, third-party tools, and Docker containers that connect to the target system. The tool is responsible the orchestrations of these components in the correct order and collect data from the experiments.

Defektor serves as a meta-tool, reusing other already available tools, and automating the entire fault-injection experiment, helping developers and integrators to identify weak parts of the system, guiding their efforts for a faster development cycle of detection, diagnosis, and correction. Defektor is well suited for microservice-based architectures, making it up-to-date with the current trends in industry. We show the feasibility and usefulness of Defektor using this tool in a case-study involving a microservice-based application.

The remainder of this paper is organised as follows: we present the key concepts of fault injection and related work in section II. In section III we show an overview of Defektor and Section IV describes the architecture of Defektor. The case study is presented in V and in Section VI we show and discuss the results. We conclude the paper in Section VII.

II. Concepts and Related Work

Testing robustness and fault tolerant properties of systems is an important part of the validation of said systems. This requires the observation of the system under evaluation in the presence of faults. However, fault activation are usually rare events. To address this, fault injection has been used to accelerate fault activation by inserting artificial faults into a given component of the system to observe how other components or the overall system behave. The object of the observation is not the target itself but another part (or the entire system). Fault injection has been used for decades to evaluate system dependability properties [16], [7], [28], [10] and risk assessment [24].

Early work of [3] and [2] proposed the initial frameworks defining the conceptual components of a fault injection experiment: the set of faults (faultload), the set operations to activate the system (the workload), the set of raw measurements (system observation) and the model to convert the raw measurements into meaningful properties concerning the system behaviour. Fault injection experiments are controlled by a set of typical tools which include the fault injector (actually inserts the intended faults into the target), the workload generator (submits the work to the system), a monitor (observes the system), and a controller (orchestrates the experiment), as described in the early work of [15]. Usually, the system is exercised first without any faults injected—the "golden run"—and then again, one or more times, with faults injected in a given component according to the faultload specification. Realistic faultloads representing real faults occurring in the operational phase are particularly hard to define, and even harder for complex software systems, where it is very difficult not only to understand which faults are realistic, but also how to inject them.

The fault injector is, by far, the most difficult component to implement, due to the complex nature of the faults (*e.g.*,, software faults), and also due to to reachability and control issues, which are technically very challenging, often having to bypass the normal semantics and behaviour of the platform and operating system. Besides the technical challenges involved in building fault injectors, another issue surrounding fault injection experiments is the ratio of fault activation. Once inserted into the target system, the fault must be activated to eventually cause an error (an internal wrong state in the system) that may or may not cause a failure (the unwanted behaviour that is observed). In particular, for software faults, to activate the fault, the workload must ensure that the execution path covers the inserted fault. To overcome this difficulty, many fault injectors insert not a fault but an error. The premise here is that the errors are a representative consequential state of the intended fault.

Fault injection has been used for several decades both in academia and industry and many early fault injectors were developed for more or less specific target types and scenarios. Examples of early tools are: specific for hardware systems ([2], [19], [22], among many others), specific for simulating hardware memory-related faults, such as bit-flips and stuck-at faults (FIAT [26], FERRARI [17], FINE [18]), and specific for given target systems, such as Online Transaction Processing (OLTP) systems [28] or web servers [10]. The specificity for a given target system or experimental scenario ties the tools to specific platform mechanisms and capabilities, limiting the fault models the tool is able to inject. Several initiatives addressed this problem by proposing modular fault injection tools. Examples include Xception [7], NFTAPE [27], Goofi [1]. The success of such modular tools is moderate given that there remains some dependency from the underlying system or target and new scenarios have become relevant for which these tools were simply not prepared to address, such as microservice architectures.

Microservice-based architectures have become relevant in the industry and several initiatives proposed fault injectors addressing these architectures. Following the approach of fault injection and borrowing some ideas from robustness testing, Chaos Engineering is a recent discipline that evolved from fault injection and is aimed at proactively discover possible hidden problems in microservice systems [5]. It follows the general approach of fault injection and helps improving the resilience of systems by creating adverse operational conditions and errors allowing developers and integrators to observe the system behaviour and proactively identify locations where improvement is needed.

The Gremlin framework [12] is a systematic resilience testing tool that allows the operator to design and execute tests by manipulating inter-service messages. The use of this tool is subject to a fee. The Simian army [25] is a comprehensive set of tools to cause errors and failures in services to assess the resilience of distributed applications following the ideas of Chaos Engineering [4]. Each tool is specialised on a specific type of failure (*e.g.*,, message delay—delay monkey, availability zone drop—Chaos Monkey, etc.). Wu *et al*. present an extensible fault tolerance testing framework for microservice-based cloud applications [29]. This framework acts mainly over the Hypertext Transfer Protocol (HTTP) communication between the services, including changing the HTTP status codes and delaying messages. Litmus is a modern multi-type fault injection tool for cloud-native systems [21]. This tool supports several types of failures related to distributed systems (*e.g.*, pod deletion, pod CPU exhaustion, container shut down, etc.) and can collect results from experiments, however it is specifically tied to the *Kubernetes* orchestrator.

Modern tools improved support for microservices scenarios, however a completely platform-independent tool for resilience testing of distributed systems is not available.

III. Defektor Overview

In this section, we present the main features of Defektor and explain the interaction of the tool with the underlying systems to control the fault injection experiments and manage the target application execution.

A. General Description

Defektor consists of a server application that exposes a REpresentational State Transfer (REST) Application Programming Interface (API) used by a command line client to manage injection campaigns. Operation of Defektor requires a number of components: the application, an injection plan, a workload generator, a fault injector and a data collector mechanism, to extract the resulting data from a specified data storing systems. Defektor interacts with all the involved tools to exercise and inject faults on the target application, while running on separate infrastructure, avoiding resource contention. The workload generator can be instantiated on multiple worker machines and the plan supports the details specifying this aspect. The integration of Defektor with the other tools involved in the experiment, in particular the fault injection tools, is done through adaptor plugins.

The user interacts with the command line client to provide an injection plan containing high-level instructions to perform a fault injection campaign. Asynchronously, Defektor server will run that plan and collect the data, leaving it available for the user to download later through the command line client. The plan describes all aspects of the campaign, including the following information: system type, targets, fault injectors, workload generators, and data collectors. Based on the plan it receives, the Defektor connects to the supporting platform or infrastructure and allows the fault injector to manipulate the target application.

The user controls all aspects of the plan, however, typically, the plan will first define a golden run, and then the run with faults. Thus, the behaviour of the system without any fault or interference is observed beforehand, as is typical in the well established fault injection, robustness testing and, later, Chaos Engineering experiments [8]. For example, a plan making two runs in addition to the golden run would be as follows:

1) Run workload generator for a predetermined amount of time;
2) Collect data (traces/metrics);
3) Apply fault 1 to a given target component A;
4) Run workload generator for a predetermined amount of time;
5) Collect data (traces/metrics);
6) Remove fault 1 injected on component A
7) Apply fault 2 to a second target component B;
8) Run workload generator for a predetermined amount of time;
9) Collect data (traces/metrics);
10) Remove fault 2 injected on component B.

B. Architectural Drivers

Enabling the use of a high-level plan and make Defektor extensible for different types of microservice-based systems were the two main design drivers.

The tool is made generic by allowing extension by plugin, enabling the addition of new fault injectors, data collectors and the ability to interact with new systems and platforms. All these plugins are referenced in a generic fashion in the plan, keeping it focused on the campaign definition, freeing both Defektor and the user from the implementations details. For example, shutting down a machine may appear in the plan, but the details about how that is done are encapsulated in the fault injector, which talks to the target system, by means of a system connector plugin (refer to Section IV).

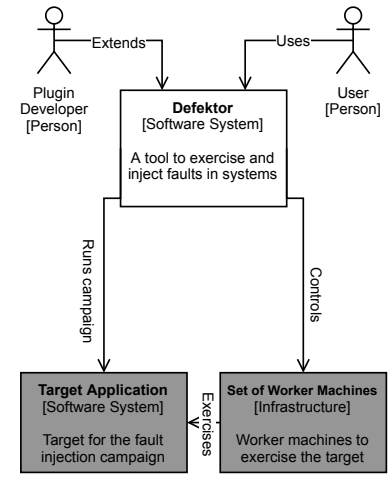
C. Functionality Overview

Defektor was developed with a fairly large number of functional requirements in mind in order to maximise its usefulness. We briefly present here a list of Defektor functionalities. To conserve space we focus on the most important functionalities, which are the plan management, the state store, the workload generation, the fault injection, and the data collection:

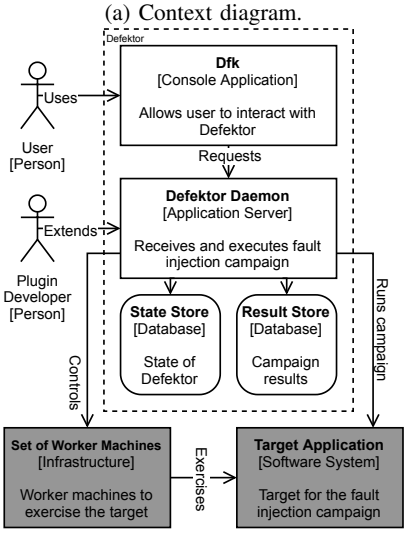
- **Plan Management:** the tool receives, validates, stores and deploys the injection plan, orchestrates the necessary steps for its execution and returns the resulting data to the user. Furthermore, since the user interacts with the tool in an asynchronous fashion, it allows the user to query the plan and its state.
- **State Store:** to keep the data about submitted plans and plan execution, Defektor employs some state store mechanism. For example, the state store can keep track of access and credential information of worker machines used for workload generation. The state store can also enable Defektor to resume an injection campaign after a crash.
- **Workload Generation:** the system uses an artificial workload to exercise the target application. To do so, Defektor gathers the necessary properties from the submitted plan to properly orchestrate an arbitrary number of worker machines. These worker machines resort to configurable Docker containers [23] to exercise the target application for a requested duration and severity.
- **Fault Injection:** Defektor injects faults with different characteristics, severities, and target types, resorting to third-party fault injectors if they are available. To accomplish this, the injection plan describes the necessary properties, *i.e.*, injector plugin, target instance, and some relevant parameters, *e.g.*, the identifier of a process to kill.
- **Data Collection:** the data collection mechanism allows the user to extract the resulting data from a specified data store of the target system. To enable Defektor to interact with any data store and fetch the information in an arbitrary format, our tool accesses data through plugins.

IV. Architecture

We present the Defektor architecture following Simon Brown's C4 Model [6]. This model specifies the architecture in up to 4, progressively finer detailed diagrams: 1 - Context Diagram, 2 - Container Diagram, 3 - Component Diagram and 4 - Code Diagram. Here, we present the first three diagrams as the last one closely resembles the source code structure.



(a) Context diagram.



(b) Container diagram.

Fig. 1: High-level architecture, C1 and C2 diagrams.

Defektor is meant to be a generic, high-level, extensible fault injection campaign management tool that can be adapted, through plugin addition, to interact with and inject faults on any system and application. It is designed according to a client-server paradigm so that the process of managing a campaign is asynchronous to the practitioners and does not depend on the state of their local machine.

A. Concepts

To make the architecture easier to understand, this subsection lists and explains the concepts and abstractions used by Defektor.

- **Defektor:** a generic, high-level, extensible fault injection campaign management tool, designed according to a client-server paradigm.
- **dfk:** the command line client used to control Defektor.
- **Plan:** a high level description of the injection campaign, containing a list of injection steps, and respective parameters.
- **Injection:** each individual injection step, where a fault is injected with an *Injektor* and the application exercised with a *Workload generator*.
- **Injektor:** a plugin that implements the fault injection logic, or connects to an external fault injector.
- **Workload Generator:** a docker container encapsulated, application-specific, workload generator used to exercise the target application.
- **Worker Machine:** a generic, docker-enabled machine, used to run the *Workload Generator* and exercise the target application.

B. Context Diagram

The context diagram has the least scope of detail. It provides an high-level view of our system, including the relationships with users and external systems. It thus provides a clear view of all the external dependencies that the system must manage. Figure 1a shows the interaction between Defektor and two classes of users and two external systems. The *User* is the practitioner deploying and starting fault injection campaigns, while the *Plugin Developer* is the person responsible for developing and maintaining plugins enabling failure injections, connection to new system types and data collection from different data stores. Once every configuration is ready, Defektor may run an injection campaign targeting the *Target Application*. To make the experiment more reliable, our application can use a set of *Worker Machines*, to generate load and exercise the *Target Application*.

C. Container Diagram

The container diagram, in Figure 1b, further details our system, breaking the single box in the previous diagram into different containers, each representing an executable and deployable sub-system. From top to bottom, we have a console application, *dfk*, providing the user some abstraction and helpful hints regarding the interaction with the Defektor REST API. This API is encapsulated in a container that we called *Defektor Daemon*. This component contains the core of the program. To persist the crucial data, Defektor uses two different databases: one persists Defektor state (injection plans, worker machines information, status of an experiment, etc.), while the other stores the injection campaign results, making them available to the user.

D. Component Diagram

This diagram gives a more detailed view of the system, decomposing every container into a group of related functionalities encapsulated behind a well-defined interface. Figure 2 portrays a more detailed vision of Defektor Daemon.

The plugins are the components that deserve the most emphasis are the plugin ones. This is because

We designed Defektor with two main principles in mind: make the tool agnostic to the cloud system and avoid recompiling the core, whenever a new type of target system is

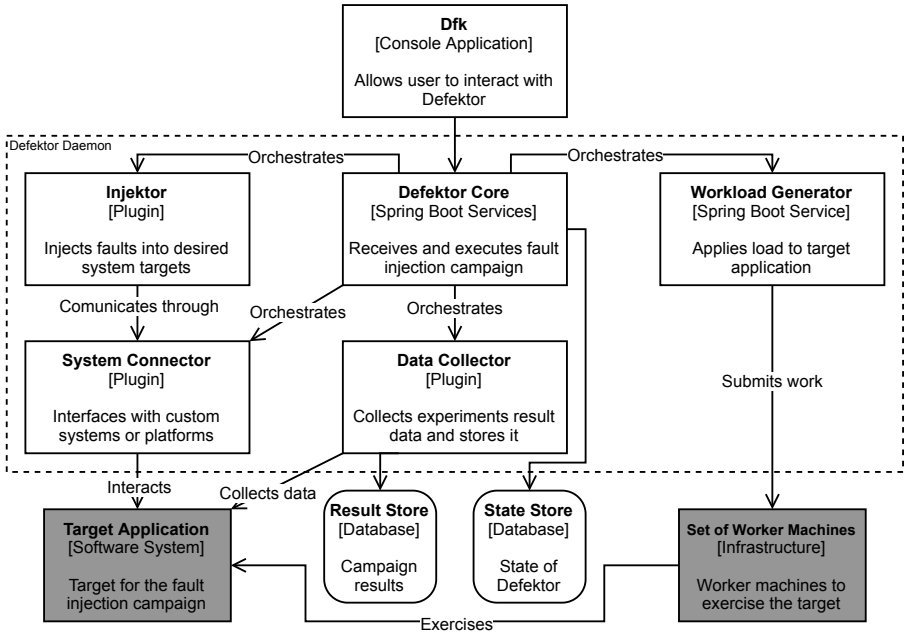


Fig. 2: Component diagram.

added. To achieve these goals, we designed Defektor following a plugin architecture. This system can be split into two different groups: core and plugins. The core parts are statically loaded and are responsible for generic functions, like serving the API, handling plans and managing and orchestrating the plugin modules. The plugins are run-time loadable, stand-alone components that provide specialized functionalities, such as fault injectors and connectors, to interface with supporting systems and platforms. This plugin architecture will be further detailed in the next subsection.

E. Plugin System

We use the following types of plugins:

- *Injektors*, which are responsible for implementing the fault injection logic.
- *System Connectors*, which are responsible for interfacing with the platforms supporting the target application; this may for example be the Operating System (OS), an hypervisor, or a container orchestration system, such as Kubernetes.
- *Data Collector* plugins, which provide the specialized logic to extract the resulting data from the target system or application; common examples would be Prometheus, OpenZipkin or other data stores typically used for monitoring data.

Even though they are designed to be independent modules, Injektor plugins require System Connector plugins to interface with the system where the target application is running. The choice of splitting injection and system connection in two different types of plugins, comes from the expectation of having multiple injectors per system type, thus preventing the duplication of logic. In this fashion, when the interface for a system type changes, only the respective System Connector will require update.

Once a plugin is imported, its functionalities become available to be used by Defektor. The communication between both components must follow an abstract interface that must be implemented by the plugin.

A final, but not less important, consideration is usability. Since plans will have to differ based on the plugins used, each plugin should implement the necessary functions for inspection. As examples, from the *dkf* command line, the user should be able to determine which are the valid targets for a specific type of injection, or which targets are provided by a given system connector, as well as which configurations are needed by each injection type.

1) System Connector *Plugins:* The *System Connector* type plugin is a bridge between Defektor and the target application or its supporting platform, *i.e.*, OS, *hypervisor* or *Kubernetes*. The Defektor Core can query the *System Connector*, to get information of the target system, while the *Injektor* type plugin uses it to inject failures. This plugin must implement the following interface:

- `help`: returns a brief introduction and some details on the plugin interaction with the target system and how *injektor* plugins should send their instructions to be performed.
- `configure`: a function that can be called to assign some configuration parameters to the plugin object.
- `getTargetTypes`: returns a list of all target types the connector can interface with, *i.e.*, machine, virtual machine, container, pod or process.

Nevertheless, some functions must be added to the plugin to properly connect and interact with the target application. For instance, to perform a fault injection in a single Virtual

Machine (VM), the first question to arise would be: "How can I interact with the target system?". As an example, one could write an Secure Shell Protocol (SSH) *System Connector* that would need to be parameterized with the appropriate SSH credentials, by means of the `configure` function. Once the connection is established, there should be a function (*i.e.*, called *sendSSHCommand*) that could be used by an *Injektor* plugin, to send commands to the target system, to force its malfunction.

2) Injektor *Plugins:* The *Injektor* type plugin is responsible for injecting faults in the target system. *Injektor* plugins depend on *System Connector* plugins, to mediate the interaction with the target system. The latter plugins encapsulate the system, by allowing the interface to stay unchanged, while encouraging sharing of the same *System Connector* code.

Injektor plugins must implement the following interface:

- `performInjection`: receives injection parameters and performs the injection.
- `stopInjection`: stops or removes the failure injection.
- `getTargetTypes`: this returns a list of targets where this particular injector can perform a fault injection.
- `getTargetInstancesByType`: it returns all the instances, with an identifier, of a type where this particular injector can perform a fault injection.
- `getInjectionStatus`: returns the status of the injection (running, stopping, or stopped).

Following the same environment that was previously given as an example, the VM, we can assume that a user wants to perform the most basic failure injection: shutdown the instance.

Considering that virtually all cloud-native system instances are running some *Linux* distribution, and having the plugin access to the *System Connector* `sendSSHCommand` function, it becomes trivial to achieve this goal, by sending the string `sudo shutdown` to the target system via SSH.

3) Data Collector *Plugins:* The *Data Collector* plugin is responsible for collecting the data generated during each run. Effectively, this is the portion of the system that returns the data for analysis. As data is system, application and purpose specific, the tool returns it in some generic format, *i.e.*, an array of bytes or file, and it is up to the practitioner to interpret and analyse it. The plugin must implement the following interface:

- `configure`: a function that can be called to assign some configuration parameters to the plugin object.
- `getData`: function that returns data or some Uniform Resource Identifier (URI) to it.

As an example, since *prometheus* is a widely used data store for monitoring data, a *Data Collector* could be written for it that would return user-selected metrics for each run.

V. Case Study

In this section we illustrate the kind of experiment one could perform with Defektor. By doing this, we demonstrate that Defektor can integrate with third-party fault or failure injection tools. We also show the ability of Defektor to apply and remove injections according to a declarative plan; manage a swarm of workers to generate load for the application; and collect data from the experiments, with and without failures.

Since Kubernetes is a widely known container orchestrator, well suited for microservice-based applications [20], we developed some *Injektors* and a *System Connector* plugin for Kubernetes to demonstrate our solution. We opted for Kubernetes primarily due to its relevance for the industry. Additionally, Kubernetes has complementary tools, like service meshes, which provide control and observability mechanisms. This is the case of *Istio*, an open-source service mesh, which has been one of the most used service meshes both for evaluation and production purposes [9]. Among other functionalities, *Istio* provides two primitive fault injection mechanisms that we may leverage for our own case: HTTP delay and HTTP abort faults.

A. *The* System Connector *Plugin*

The *System Connector* plugin includes two functions, to provide *Injektors* the means to operate the target application environment. As an example, it provides the primitives *applyManifest* and *removeManifest*, to add and remove resources via YAML Ain't Markup Language (YAML) declarative manifests. These manifests will be generated by the *Injektor* plugins, and will contain instructions to parameterize and toggle the *Istio* fault injection mechanisms. Some example parameters may namespace, target service, fault occurrence probability, or request delay.

B. *The* Injektor *Plugins*

The *Injektor* plugins control the system through the *System Connector* Plugin. In our case study, we resort to delay and abort faults, because *Istio* offers both types of faults out of the box.

1) HTTP Delay Fault: This fault injection inserts a delay in HTTP packets for a target service. For example, if this failure is injected in Service B and Service A requests something from Service B, there is a probability $P(D)$ that Service B will delay its response x seconds. The practitioner may configure both, the probability and x in the injection plan.

2) HTTP Abort Fault: This fault injection enables the possibility to insert an error in HTTP packets destined for a specific service. For instance, if this failure is introduced in Service B and Service A requests something from Service B, there is a customisable probability that Service B will respond with a configurable HTTP code. For example, if we want to cause a failure in the target service, sending `5xx` HTTP status code as a response to a request will be handled as an error by the service that requested the faulty service. The practitioner may configure both the failure probability and the HTTP status in the injection plan.

C. Application

The design of Defektor gives the practitioner the ability to perform consistent fault injection campaigns on different microservice-based applications. As we mentioned before, to

achieve this goal in a fully generic manner, Defektor resorts to plugins, to interface with the supporting infrastructure and platforms, collect data, and to implement fault injection mechanisms.

To run our experiments, we needed an application compatible with the plugins we had already created. We thus required an open-source Kubernetes-based microservice application with the *Istio* service mesh enabled. We found two applications that met these criteria: *Stan's Robot Shop*, by *Instana* [13] and *Bookinfo Aplication*, by *Istio* [14]. The latter was developed with the goal of serving as a sample application for testing out all of *Istio*'s features. It is, however, a very basic application with only four services bearing little resemblance to any real-world product.

Stan's Robot Shop, on the other hand, is an e-commerce sample application with a higher degree of complexity and a closer resemblance to a real-world deployable product. It depends on eleven containerized services running on Kubernetes: *cart*, *catalogue*, *dispatch*, *mongodb*, *mysql*, *rabbitmq*, *ratings*, *redis*, *shipping*, *user*, and *web*. These are in charge of handling user actions, such as logging in, browsing the catalogue, adding items to the cart, paying, and shipping the desired products. Given the size and contents of the application, we selected *Stan's Robot Shop* to perform the experimental evaluation. Furthermore, *Instana* provides a load generator to exercise its own application. We can take advantage of this generator for our own experiments.

D. Experiment setup

In our case study, we ran injection campaigns, setting parameters for both injectors, HTTP Delay and HTTP Abort, in the injection plan.

This use-case is exemplified in Listing 1. A plan can have a list of different fault injections but for brevity the listing only shows one of them, HTTP Abort, in the `injektions` array on line 4. Note that even though we only specify 1 run, Defektor will automatically do a "golden run" without any faults; it then injects a fault that stays active until the end of the run. In the end, the faults are removed and the application returns to its normal state. We collect data at the end of each run.

In the Delay and Abort experiments we target the *cart* service. The time for each run (golden run and fault injection run) was set to 600 seconds and the load generator was configured with one client exercising the system, by performing arbitrary tasks in the e-commerce website.

For the HTTP Delay fault, we want to analyse what happens to the average response time of the server, while for the HTTP Abort fault, we want to analyse what happens to the failure rate and if this fault propagates to other services.

VI. Results

We now present and analyse the results of the experiments for the delay and abort faults.

```
1  name: AbortCart100Percent
2  system:
3    name: kubernetes
4  injektions:
5    - totalRuns: 1
6      ijk:
7        name: httpabort
8        params:
9          namespace: robot-shop
10         service: cart
11         host: cart.robot-shop.svc.clust⌋
             ↪  er.local
12         httpStatus: '555'
13         faultOccurrence: '100'
14     workLoad:
15       image:
16         user: robotshop
17         name: rs-load
18         tag: latest
19       env:
20         host: http://system.example.com/
21         numberClients: '1'
22         silent: '1'
23         error: '1'
24       replicasPerWorker: 1
25       workers: 1
26       duration: 600
27     dataCollector:
28       name: jaeger
29       params:
30         host: http://example.com:16686
```

Listing 1: HTTP Abort Failure Injection Campaign Plan.

A. HTTP Delay

We start by evaluating if the injector is operating properly. For this, we delay the responses of the *cart* service. Figure 3 shows an injection campaign sample, where the delay occurs 100% of times. It starts the load generator at $x = 0$, exercising the target application. At $x = 600$, Defektor stops the load generator and cools down the system to collect data without interference. The time interval $x \in [0, 600]$ represents the golden run.

At $x \in]600, 689]$ the system uses the *Jaeger* data collector plugin to extract the traces and metrics generated by the golden run and store the data in *Jaeger* [11]. Due to the infrastructural limitations on writing and reading this data on storage, this process takes more time than usual.

The time interval $x \in]689, 1289]$ represents the fault injection run. The HTTP Delay injection starts at $x = 690$. From this instant until $x = 1289$ the system is again exercised and the *cart* service will take exactly 5 seconds to respond every single request it receives. During this run, we can observe that the average response time of the entire application dramatically increases, indicating that the failure was successfully deployed.

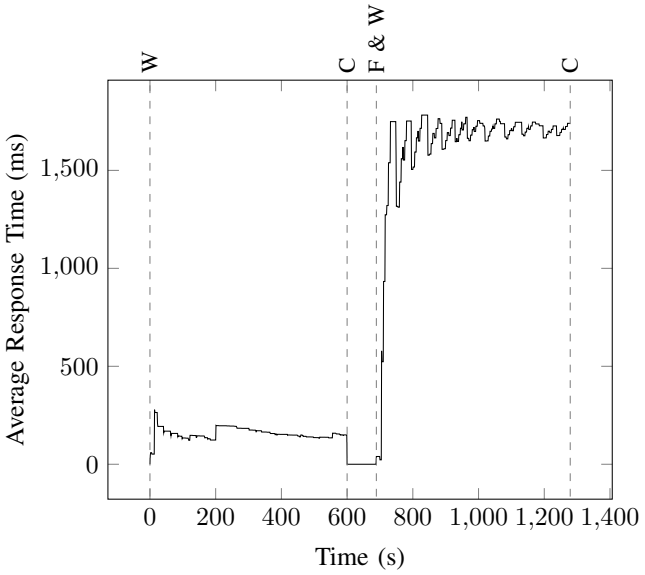


Fig. 3: Sample HTTP Delay injection campaign, with typical injection stages: Run **(W)**orkload; **(C)**ollect Data; Inject **(F)**ault.

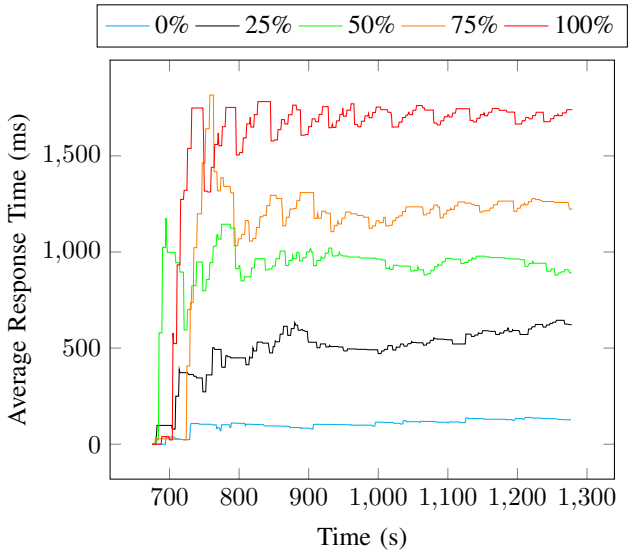


Fig. 4: HTTP Delays injection campaigns for different failure activation probabilities.

To determine the impact of a slow service on the overall application or to exercise a load balancer or some fault-tolerance mechanism, one could exercise varying levels of delay in this same *cart* service. This would involve carrying out multiple experiments, with different fault activation probabilities. We set these fault activation probabilities to $\{0\%, 25\%, 50\%, 75\%, 100\%\}$, still in the *cart* service, and display the results in Figure 4.

For visualization purposes, we omitted the golden run phase in all experiments and kept only the fault injection run in the

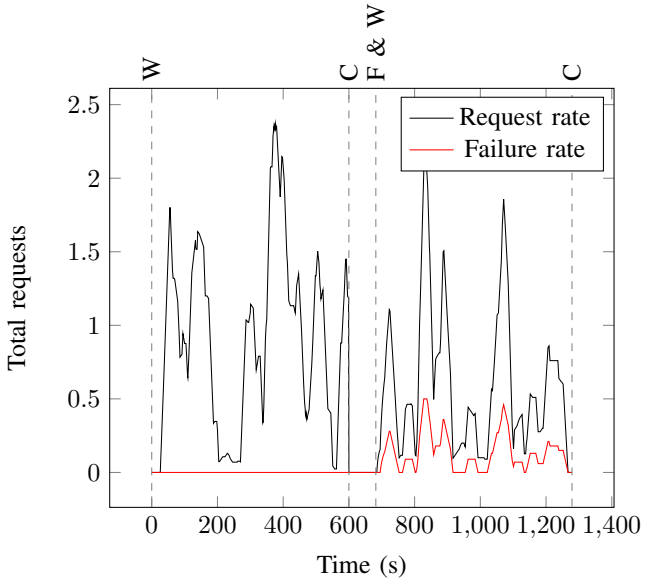


Fig. 5: Sample HTTP Abort injection campaign, with typical injection stages: Run **(W)**orkload; **(C)**ollect Data; Inject **(F)**ault.

interval $x \in [680, 1280]$ (for a total of 10 minutes per run). The plots show that the higher the fault activation probability is, the higher the total average response time is. This demonstrates that Defektor is capable of injecting this fault with varying degrees of severity in the application.

B. HTTP Abort

We now discuss the experiments with HTTP Abort faults. These faults affect 100% of the requests to the *cart* service. We study Defektor's capability to manage an HTTP Abort fault campaign and its impact on the system's overall failure rate, whenever a service is responding with HTTP error status codes to other services.

Figure 5 portrays an injection campaign sample (simple moving average n=30). This figure shows the number of requests the application serves per second. We may compare Figure 5 to Figure 3, as it covers the entire experiment, first in the golden run, then in the faulty run. It starts the load generator at $x = 0$, exercising the target application in the golden run. At $x = 600$ Defektor stops the load generator. At $x \in]600, 682]$ the system uses the *Jaeger* data collector plugin, to extract the traces and metrics generated by the golden run. The time interval $x \in]682, 1282]$ represents the fault injection run. The HTTP Abort injection is injected at $x = 683$. From this moment until $x = 1282$ the system is again exercised and the *cart* service returns the 555 HTTP status code for 100% of the requests it receives. During the golden run, we can observe that the entire system does not have failures, but that these start as soon as the abort failure is injected.

To better understand the impact of service failures (i.e., services returning an 5xx *status* code), practitioners could vary the fault activation probability. We carried out four

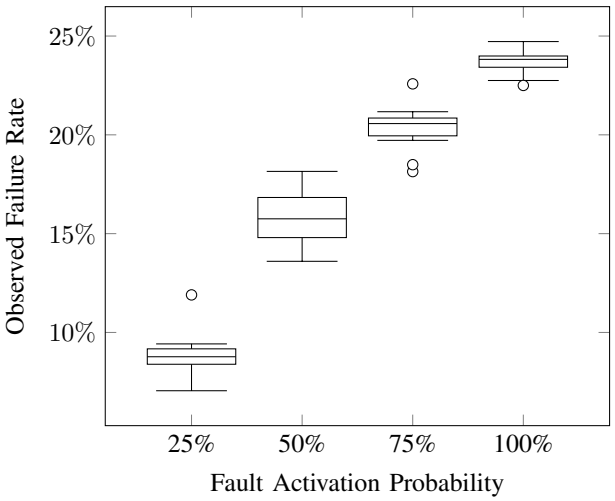


Fig. 6: Observed system failure rate, when *cart* service is injected with HTTP Abort fault with different activation probability.

experiments with fault activation probabilities of {25%, 50%, 75%, 100%} in the *cart* service, displayed in Figure 6. This figure shows, for each probability of *cart* service failure, the quartiles, median, maximum, minimum and outliers. An experiment like this demonstrates the dependency of the overall application on a specific service, for a given mixture of requests. As our experiment revealed, the observed failure rate is not linear with respect to the fault activation probability. Given the set M of all mean error rates M_a, for a given activation probability a, if we normalise around the 100% activation probability or $a = 1$, following Equation 1, we get $\{36.8\%, 66.1\%, 86.4\%, 100\%\}$, shown as a percentage. This step is necessary because we are injecting a single service, *cart*, but the workload exercises the whole system, diluting the failure rate.

$$M'_a = \frac{M_a}{max(M)}, \quad \forall\, a \tag{1}$$

This shows that for activation probabilities under 100% we see a higher failure rate than expected, which can be explained by failure propagation. As an example, a product is added to the cart and then another request is made that depends on it; that new request now has a reduced probability of success, as it is conditioned by the success of the former *i.e.*, it might fail as the result of the injected fault or it might fail because the insertion request failed, and the product is not on the cart.

VII. Conclusion

In this paper, we presented Defektor, a tool for fault injection campaign management, its main architectural drivers and its architecture. In addition, we present a case-study of its application to a microservice-based application. By using plugins, Defektor is extensible, thus being able to target a large spectrum of systems, including microservice architectures, a paradigm where industry leaders rely heavily on fault injection for resiliency testing. Defektor departs from previous state of the art, by automating the fault injection workflow, reducing the human element, and leveraging a plugin-based extension system that allows integration with third-party tools, speeding up data collection and ensuring consistency among runs.

So far, for validation purposes, we have implemented two *System Connectors*, for *Kubernetes* and SSH-enabled Linux machines, as well as four *Injektors*, Process Terminator, Machine Shutdown, HTTP Delay and HTTP Abort. The tool and plugins are currently available as open source in a public repository **[Omitted for Double-blind Review]**.

As future work, we plan to write *Injektor* plugins for state of the art fault injectors as well as improve the syntax for accepting vector parameters (*e.g.*, `workers: [1, 10, 100]`), making it easy to succinctly describe multiple runs with different parameterisations. Furthermore, we will move towards making the plans more expressive, allowing for example for concurrent injection of multiple faults.

Acknowledgments

[Omitted for Double-blind Review]

References

[1] Aidemark, J., Vinter, J., Folkesson, P., Karlsson, J.: Goofi: generic object-oriented fault injection tool. In: 2001 International Conference on Dependable Systems and Networks. pp. 83–88 (2001). https://doi.org/10.1109/DSN.2001.941394

[2] Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.C., Laprie, J.C., Martins, E., Powell, D.: Fault injection for dependability validation: a methodology and some applications. IEEE Transactions on Software Engineering **16**(2), 166–182 (1990). https://doi.org/10.1109/32.44380

[3] Arlat, J., Crouzet, Y., Laprie, J.C.: Fault injection for dependability validation of fault-tolerant computing systems. In: [1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers. pp. 348–355 (1989). https://doi.org/10.1109/FTCS.1989.105591

[4] Basiri, A., Behnam, N., De Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., Rosenthal, C.: Chaos engineering. IEEE Software **33**(3), 35–41 (2016)

[5] Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., Rosenthal, C.: Chaos engineering. IEEE Software **33**(3), 35–41 (2016). https://doi.org/10.1109/MS.2016.60

[6] Brown, S.: The C4 model for visualising software architecture, https://c4model.com/

[7] Carreira, J., Madeira, H., Silva, J.G.: Xception: A technique for the experimental evaluation of dependability in modern computers. IEEE Trans. Softw. Eng. **24**(2), 125–136 (Feb 1998). https://doi.org/10.1109/32.666826, https://doi.org/10.1109/32.666826

[8] Chaos Community: Principles of chaos engineering (2018), https://principlesofchaos.org/

[9] CNCF SURVEY 2019: Deployments are getting larger as cloud native adoption becomes mainstream (2019), https://www.cncf.io/wp-content/uploads/2020/08/CNCF_Survey_Report.pdf

[10] Durães, J. and Vieira, M. and Madeira H.: Dependability benchmarking of web-servers. In: Lecture Notes in Computer Science. SAFECOMP 2004, vol. 3219, pp. 297–310 (2004). https://doi.org/10.1007/978-3-540-30138-7_25

[11] Foundation, T.L.: Jaeger: open source, end-to-end distributed tracing, https://www.jaegertracing.io

[12] Heorhiadi, V., Rajagopalan, S., Jamjoom, H., Reiter, M.K., Sekar, V.: Gremlin: Systematic resilience testing of microservices. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). pp. 57–66 (2016). https://doi.org/10.1109/ICDCS.2016.11

[13] Instana: Robot shop: Sample microservice application (2021), https://github.com/instana/robot-shop

[14] Istio: Bookinfo application (2021), https://istio.io/latest/docs/examples/bookinfo/

[15] Iyer, R.K.: Experimental evaluation. In: Proceedings of the Twenty-Fifth International Conference on Fault-Tolerant Computing. p. 115–132. FTCS'95, IEEE Computer Society, USA (1995)
[16] Kanawati, G., Kanawati, N., Abraham, J.: Ferrari: a tool for the validation of system dependability properties. In: [1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing. pp. 336–344 (1992). https://doi.org/10.1109/FTCS.1992.243567
[17] Kanawati, G., Kanawati, N., Abraham, J.: Ferrari: a flexible software-based fault and error injection system. IEEE Transactions on Computers **44**(2), 248–260 (1995). https://doi.org/10.1109/12.364536
[18] Kao, W.I., Iyer, R., Tang, D.: Fine: A fault injection and monitoring environment for tracing the unix system behavior under faults. IEEE Transactions on Software Engineering **19**(11), 1105–1118 (1993). https://doi.org/10.1109/32.256857
[19] Karlsson, J., Gunneflo, U., Liden, P., Torin, J.: Two fault injection techniques for test of fault handling mechanisms. In: 1991, Proceedings. International Test Conference. pp. 140– (1991). https://doi.org/10.1109/TEST.1991.519504
[20] What is Kubernetes?, https://kubernetes.io
[21] Litmus: Chaos engineering for your kubernetes, https://docs.litmuschaos.io/docs/introduction/what-is-litmus
[22] Madeira, Henrique and Rela, Mário Zenha and Moreira, Francisco and Silva, João Gabriel: Rifle: A general purpose pin-level fault injector. In: Proceedings of the First European Dependable Computing Conference on Dependable Computing. p. 199–216. EDCC-1, Springer-Verlag, Berlin, Heidelberg (1994)
[23] Merkel, D.: Docker: Lightweight linux containers for consistent development and deployment. Linux J. **2014**(239) (Mar 2014)
[24] Moraes, R. and Duraes, J. and Barbosa, R. and Martins, E. and Madeira, H.: Experimental risk assessment and comparison using software fault injection. In: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07). pp. 512–521 (2007). https://doi.org/10.1109/DSN.2007.45
[25] Netflix: Simianarmy, https://github.com/Netflix/SimianArmy
[26] Segall, Z., Vrsalovic, D., Siewiorek, D., Ysskin, D., Kownacki, J., Barton, J., Dancey, R., Robinson, A., Lin, T.: Fiat - fault injection based automated testing environment. In: Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'. pp. 394– (1995). https://doi.org/10.1109/FTCSH.1995.532663
[27] Stott, D., Floering, B., Burke, D., Kalbarczpk, Z., Iyer, R.: Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors. In: Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000. pp. 91–100 (2000). https://doi.org/10.1109/IPDS.2000.839467
[28] Vieira, Marco and Madeira, Henrique: A dependability benchmark for oltp application environments. In: Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29. p. 742–753. VLDB '03, VLDB Endowment (2003)
[29] Wu, N., Zuo, D., Zhang, Z.: An extensible fault tolerance testing framework for microservice-based cloud applications. In: Proceedings of the 4th International Conference on Communication and Information Processing. pp. 38–42 (2018)