Instituto Superior
de Engenharia
Politécnico de Coimbra

DEPARTMENT OF SYSTEMS AND COMPUTER ENGINEERING

Defektor: A Chaos Engineering campaign orchestrator

Project Report to fulfill the Master's degree in Informatics Engineering

Specialization in Software Engineering

Author

Rui Jorge da Silva Neves

Supervisor

João António Pereira Almeida Durães

Co-Supervisor

Filipe João Boavida de Mendonça Machado de Araújo

Coimbra, April 2025

INSTITUTO POLITÉCNICO
DE COIMBRA

INSTITUTO SUPERIOR
DE ENGENHARIA
DE COIMBRA

Resumo

O aspeto de confiabilidade dos sistemas modernos tornou-se num dos mais importantes em termos de qualidade de software, com o aumento do número de sistemas e utilizadores, um desafio à sua estabilidade. De forma a preparar para estes desafios, os desenvolvedores dedicam uma quantidade de tempo considerável a criar mecanismos e técnicas para melhorar a resiliência e a robustez do sistema em desenvolvimento. Isto geralmente inclui técnicas de injeção de falhas para garantir que o sistema possa ser testado num ambiente com adversidades e provocar estados internos inválidos, em vez de esperar que estes ocorram naturalmente, já que geralmente estes problemas são raros.

Apresentado neste trabalho, o Defektor é uma ferramenta já em desenvolvimento que foi aprimorada e testada para demonstrar a possibilidade de se aplicar a qualquer projeto. A ferramenta atua como um gestor de campanhas, configurado para controlar múltiplas máquinas, usar diferentes ferramentas externas, interagir com sistemas alvo e obter dados que podem ser analisados. O Defektor recebe uma série de parâmetros que lhe permitem automatizar a execução da campanha, o que torna os resultados mais consistentes e facilita a replicação de experiencias. O Defektor executa essas ações por meio de integrações com ferramentas de injeção de falhas que interferem nos sistemas alvo, ferramentas de tracing e logging para obter dados e containers Docker que executam workloads, mantendo-se agnóstico em relação a ferramentas, sistemas e infraestruturas.

Palavras-chave: Chaos Engineering, Injeção de falhas, Cloud, Microserviços

Abstract

The reliability and dependability aspect of modern systems has become one of the most important in terms of software quality, with an increase in the number of systems and individual users, challenging their stability. To better prepare for these challenges, developers spend a considerable amount of time creating mechanisms and using techniques to improve the resilience and robustness of the system in development. This usually includes techniques of fault injection to ensure that the system can be tested in an adverse environment and provoke invalid internal states, instead of waiting for them to naturally occur, as it can take much longer.

Presented in this work, Defektor is a tool already in development that has been improved and tested to show its real-world applicability. It acts as a campaign manager, which is configured to control multiple machines, use different external tools, interact with a target system, and collect that that can be analyzed. It receives a series of parameters that allow Defektor to completely automate the campaign execution, which makes results more consistent and easier to replicate experiments. Defektor performs these actions through integrations with fault injection tools that interfere with the target systems, tracing and logging tools to collect the data, and Docker containers that execute workloads, staying agnostic regarding external tools, systems, and infrastructures.

Keywords: Chaos Engineering, Fault Injection, Cloud, Microservices

Index

Table index

Figure index

Acronyms list

AI	Artificial Intelligence
API	Application Programming Interface
AWS	Amazon Web Services
CI	Continuous Integration
CLI	Command-Line Interface
CISUC	Centre for Informatics and Systems of the University of Coimbra
CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete
FIS	Fault Injection Service
FR	Functional Requirement
HTTP	Hypertext Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
JAR	Java Archive
JSON	JavaScript Object Notation
MELT	Metrics, Events, Logs, and Tracing
UI	User Interface

1 Introduction

This document presents the dissertation in the context of a Master's in Informatics Engineering, specialization in Software Engineering, advised by Professor João Durães from Instituto Superior de Engenharia de Coimbra and Professor Filipe Araújo from the University of Coimbra, presented to the Politechnic Institute of Coimbra, Coimbra Institute of Engineering, Department of Systems and Computer Engineering.

1.1 Context

The ever-changing world of software systems has forced developers to research and develop new methods and improve existing ones to address growing challenges. With the spread of mobile devices with internet connections and the overall number of users increasing, a need for more scalable software capable of handling higher workloads and more concurrent users arose. New software architectural kinds were developed to help handle these challenges. Systems started to be developed with small pieces decoupled from each other, each with its own technological stack and development life cycle, commonly known as Microservices architecture. On the other hand, sensing a new business opportunity, some companies started renting their infrastructure, known as Cloud, which became a very common way to deploy these new independent system parts.

Both the microservices architecture and the cloud infrastructure have drastically changed how software is developed and deployed, especially at scale. Microservices architecture has proved to create systems that can handle more concurrent users and a higher amount of requests, by creating systems easier to distribute over many machines, distributing the load, but also in the development part, it is easier to create each of the small applications that form the entire system. On the other hand, cloud infrastructure has made it easier to have infrastructure with the requirements to handle the requests, by making the necessary number of machines available, and providing other services that facilitate the deployment and maintenance of these distributed systems, such as workload distribution. But it is not without challenges. Microservices make systems harder to test end-to-end, it is harder to have a complete view of the data flow within the system. Also, in terms of service delivery, network delays can impact response times to client requests, as there are more internal requests between services. Regarding cloud, it is still a considerable expense, and depending on the hardware from a different identity carries a certain level of risk of simply losing those machines

or the services made available by the provider.

To overcome the challenges imposed by the microservices architecture, mechanisms and methodologies have been developed or improved to achieve better quality processes and results with these systems. Logging evolved, and tracing was introduced to make sure that developers can track the entire flow of data and service activations for a specific client request, for example. Quality assurance processes evolved, and techniques such as robustness and resilience testing started to be applied. These methodologies, already used broadly in software development, were applied to understand how a system would handle the presence of errors or in adverse environments. Fault injection started to be used to ensure that these faulty states would happen without the need to wait for them to occur naturally, since they can be very rare. A more specific technique, Chaos Engineering, also started to be used; it also revolves around fault injection, but in this case, it works on a production level, on software being used by the clients, with a certain level of randomness to asses how the system handles it and what the impact to the users was. These new quality processes now form a common base for how the quality of these microservices applications is assessed.

1.2 Goals

This work revolves around Defektor, a Chaos Engineering and fault/failure injection campaign manager, which coordinates and uses different machines and tools that interact with a target system and collect data.

The objectives of this work were to provide evidence that Defektor is capable of working with third-party tools, different infrastructures, and diverse target systems. Deriving from this, also to analyse Defektor, identify weak points it might have, and improve it following the findings of this analysis.

The expected result is a more complete Defektor, shown to work with multiple tools, with technical relevancy, targeting systems developed independently of Defektor itself.

1.3 Results

Tests of Defektor have shown its ability to integrate with other tools and influence external systems. It was shown to interfere with the correct functioning of a system running on different machines and networks. Through different injection tools, it communicated with multiple machines to activate the system and to collect data regarding the influence exercised, making the tests a success.

1.4 Project stages

The project has 4 steps. The first one is the analysis of Deketor, with two objectives: to understand how it works and its structure, and also to understand where it can be improved. The developments made to improve and complete Defektor are the second step of the project. Then, the third step is the necessary integrations with tools picked to test Defektor. Finally, the forth step is the validation of Defektor, with multiple tools and the integrations developed in the previous step.

1.5 Document Structure

The remainder of this document is structured in the following way.

The Concepts and Related Work chapter presents the underlying concepts that support the work made in this project, as well as the related work that aided in the decision-making processes of the work. Defektor is the chapter that originates from the initial analysis of Defektor. It presents the tool in the initial state before the changes performed in this project. Development is the chapter that explains the improvements made to Defektor and how other tools were integrated with Defektor. Experimental Validations explains the strategy used to test Defektor in regards to its applicability to the real world, as well as working with the integrations, explains the tests made, and the results. Finally, Conclusions has the final remarks regarding this project, going over the entire process, improvement points, results, and possible future directions.

2 Concepts and Related Work

This chapter introduces the concepts used within the remainder of the document. It also presents the related work present in the literature.

The chapter is organized as follows: Microservices architecture, Key Concepts, Resilience and Robustness testing, Common Fault models, Injection Techniques, and Tools.

2.1 Microservices architecture

This section focuses on the architectural components of microservice architecture and the functionality and challenges that this kind of architecture brings.

There are multiple definitions of a microservice. In this work, we will be using the definition of a microservice as a program that can be deployed, tested, and scaled independently, with a single responsibility [1]. This positions a microservice as a simple building block of a larger system. This simplicity is one of the reasons that microservices architecture is popular, and often preferred over monolithic architecture nowadays. This distributed architecture improves system performance, scalability, and operation.

In a microservices architecture, a system is comprised of these small applications, with dedicated data storage when necessary, that are responsible for a single part of the whole system. This makes them easier to understand and, therefore, simpler to maintain and develop. The applications communicate through the network to pass data around and achieve different tasks. Architectural components add certain behaviors that facilitate the usage of the system, such as an API Gateway (further explained in section 2.1.1) and support the functionality of the services themselves.

Microservices architecture's main objective is to improve scalability and overall performance. Scalability refers to the ability of the system to adapt to fluctuating load circumstances (traffic, data volumes, etc.). One of the most important concepts that improve quality is the concept of single responsibility. This facilitates development and maintenance by ensuring that each aspect of the entire system has a single service handling it, which also provides a certain level of autonomy. Common ways to assess and define the desired performance of the system are response time, error rates, and availability. Response time represents how long it takes for the system and services to respond to a request; error rates measure how prevalent errors are for a specific period; and availability is a measurement of how long the system or service is working correctly during a certain time frame, usually in the form of a percentage. These

types of metrics are commonly used when specifying Service Level Agreements and Service Level Objectives. The agreements are contracts with clients that the system and company must abide by, and the consequences when they don't. The objectives represent the metric values the teams must strive to achieve to comply with the agreements. These agreements usually involve availability, responsibilities/requirements, and support operations.

There are some challenges associated with the usage of this architecture. Its distributed nature makes it harder to test, network communication can create performance issues because of latency, and data consistency can be harder to ensure ([2]). Development techniques and frameworks, such as Metrics, Events, Logs, and Tracing (MELT), have been developed to help handle these problems.

2.1.1 Components

Systems developed following this architecture can have many architectural components. These components ensure that system functionality is maintained and enable a series of aspects that allow inspection, integration, and configuration. It is possible to have microservices without these components, but they greatly improve the delivery and maintenance of the system.

These components include the service Layer (or mesh), Service Discovery, Load Balancer, API Gateway, Service Registry, Circuit Breaker, and Configuration Server. The Service Layer is the core of the whole system; it is the services themselves. Service Discovery and Service Registry enable the system to work as a whole. It is very common for microservices applications to be deployed via the cloud, which means that the network path to a microservice can change over time. The Service Discovery component with the Service Registry ensures that every microservice is available, holding the value of each microservice's network path. The API Gateway eases the integration process with clients using the system, and it provides a single entry point, as every client request is made to this component. It can then map to a specific service and call multiple services when necessary. It works with Service Discovery to find the path of the services and with the Load Balancer. This latter component ensures that the requests are spread across the multiple servers serving the services, keeping in mind that this can also be done on the client side. The Circuit Breaker helps to handle faults, preventing requests to a service that is in a faulty state, usually putting it in a sort of cooldown timeout after which requests can be made again. Finally, the Configuration Server is a component that centralizes configuration information that is used by multiple services, providing a way to quickly configure any common aspect.

Figure 2.1 shows the flow of data and influence between these different components in a microservices system that employs these components. A request from a client comes into the API gateway, which uses the Load Balancer and Service Discover and Registry

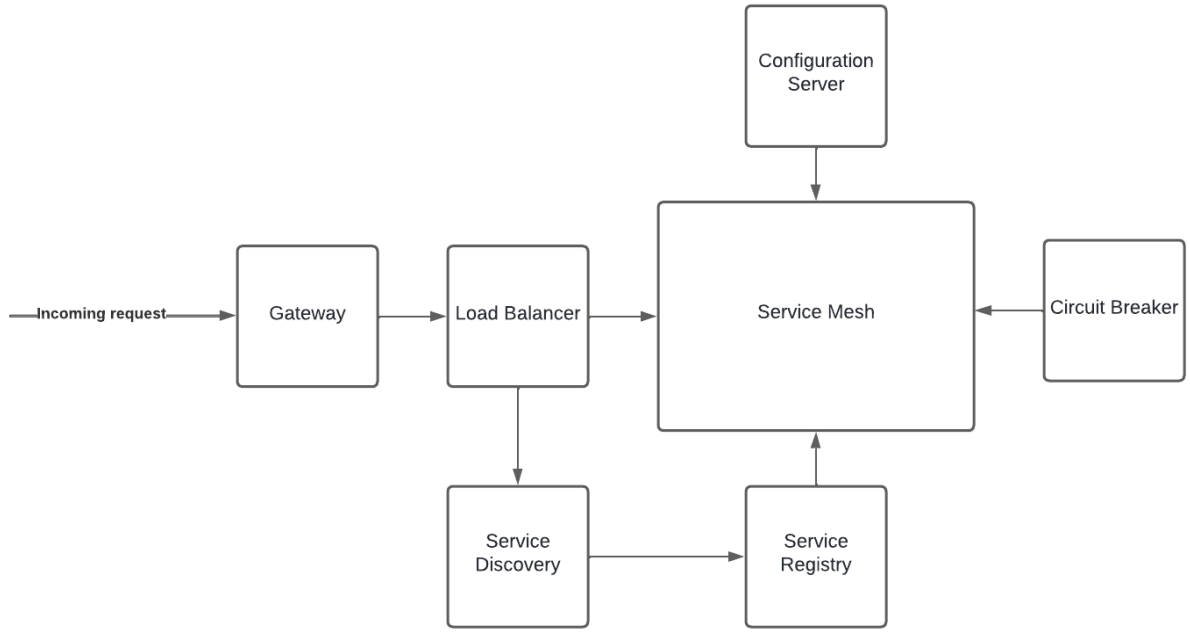

Figure 2.1: Microservices components

to get the service the request should be routed. The Configuration Server and Circuit Breaker influence the services in the mesh by providing settings and helping handle failures.

Beyond this, components such as Service Monitoring and Service Orchestration greatly improve the system's maintenance operations. The Service Monitoring component, as the name indicates, allows the development team to monitor the services to understand their performance and overall health. The service Orchestration component makes calls to microservices so that they work together. This happens when a certain (user) action triggers some functionality that requires multiple services, guaranteeing the order of execution and that the correct information is shared between the services.

2.1.2 Operation of Microservices Systems

The distributed nature of the microservices architecture makes software communication mechanisms a central point of the development of the system. The most common ones are: REST APIs, message-based systems (such as Kafka), and Remote Procedure Call (RPC). REST APIs usually use Hypertext Transfer Protocol (HTTP) and are synchronous; the client sends the request and then waits for a response. Message-based systems, also known as event-based systems, such as Kafka, are asynchronous, as the client publishes a message that the server will eventually handle without any feedback. Finally, RPC is one of the older ways to communicate via the network, where machines can call methods to be executed on a different machine. For this, a contract interface is developed that describes what methods are available on each machine.

Operating a system with a microservices architecture does have some challenges: data consistency, network latency, tracing and logging, and overall complexity. To overcome these difficulties, a series of methodologies and tools were developed. Part of this was the components mentioned previously, but one of the most important parts of working with this type of system is the MELT framework. MELT stands for Metrics, Events, Logs, and Tracing, and it denotes 4 important aspects to improve the visibility and operation of a distributed system. Metrics are measurements of certain aspects or occurrences within the system, such as memory usage and error rates. They provide a high-level view of the system state and can be helpful for things such as forecasting. They provide a quick way to understand how the system is behaving and what it is trending to. Events are occurrences within the system at a certain point in time. Events help contextualize other things, such as metrics. Examples of events are alerts or specific user actions, such as login or purchase. Logs are records of what is happening within the system and are a key part of the quality assurance process, as they provide critical information when dealing with problems by providing visibility over aspects that cannot be analyzed in other ways. For example, we can check how many purchases happen of a product on a database,e but a log is enough to understand how many times users open a product. Finally, Tracing is a collection of operations that describe a workflow or transaction in the system. Each unit of work is a span, representing an operation, and it is the basic unit of tracing. This provides a better understanding of the data flow within the system and the performance of a distributed system. Tracing is the hardest of these concepts to implement, as it requires additional development to register tracing data.

Correct usage of deployment tools can also help with these issues. Proper platform and load balancing configuration improve system metrics by ensuring that each service is not overloaded.

2.2 Resilience and Robustness Testing

Resilience testing is a way of testing that attempts to evaluate a system regarding its capacity to handle and recover from faulty states. Robustness testing is very similar; the difference is that the expected behavior from the system is different. In resilience testing, the system's ability to recover from an issue promptly is enough to consider it successful, but robustness requires the system to always stay in a correct state.

Fault injection is a technique that revolves around causing a real or simulated fault on a system to induce an error. This ensures that development teams can evaluate the system behavior without having to wait for the injected fault or the error to occur naturally. To correctly understand this, it is important to define what an error or a fault is. Following the definition from [3], a fault is the cause of an error, for example, a faulty

piece of code that does not perform the correct action (e.g. an incorrect condition on an if state). When an error causes the system to not behave in the intended way this is called a failure. Recovery is the transition from this faulty state to a correct one.

- Fault - a hypothesized cause of an error
- Error - a deviation from the system's expected state
- Failure - when the incorrect state from an error causes the functionality of the system to be incorrect as well

To inject a fault, it is important to understand what faults make sense given the system and what circumstances it operates under. So part of the process is defining what should be injected. This fault or group of faults is called a fault load, further explained in section 2.3. It is also necessary to exercise the system in a way that may activate the failure. This is done via a workload, which is usually processed automatically and simulates real-world usage.

Fault Injection can be used for Robustness and Resilience testing, as a way to quickly understand the real impact of an error or failure in a system and the recoverability of that system. Defektor works with these concepts in mind to make the developer work easier and faster, especially in terms of replication and implementation.

This type of testing is usually done to ensure that the system under development performs according to the performance and functionality required under certain circumstances and to minimize security issues and failures, which is very useful when dealing with things such as SLAs.

For microservice systems, this is usually made through Chaos Engineering, a concept created by Netflix [4], and fault injection. Fault injection has some other use cases, such as the evaluation of Artificial Intelligence (AI) models, usually related to the automatic detection of failures in a service mesh [5] [6] [7], where fault injection guarantees that at least some runs will have problems. While the work from those papers is not directly related to the work developed in the context of this one, these works provide a practical example of possible Defektor usage.

Chaos Engineering

Chaos Engineering consists of injecting realistic faults and failures into production environments to analyze the system's response, firstly referred to with this name by Netflix [4], and is defined by them as "experimenting on a distributed system to build confidence in its capability to withstand turbulent conditions in production". Chaos Engineering is a specific type of resilience or robustness testing with a group of ideas that can only be implemented in certain kinds of systems. These specific aspects are:

- Define what a steady-state is

- Use real-world disruptions
- Perform it in production environments
- Automate the process

These aspects ensure that real value can be obtained from the experiments by performing them on production with realistic inputs and having a good understanding of what is considered expected from the system, both at the functional and non-functional level, framed accordingly with the project SLAs and SLOs.

2.3 Common fault models and fault loads

Fault models are models that represent a failure in a certain process. In the context of this work, a fault model represents a fault that can be injected into a system and that represents something failing within a process of that system. Some examples of models could be simulating high levels of disk and memory usage, a complete shutdown of an entire machine, changing error messages and states, and manipulating access permissions or security settings.

There are multiple ways to categorize faults and failures, based on non-functional requirements [8], on the cause [9] or the moment when the fault occurs [10]. Most importantly, these taxonomies provide examples of real-world and commonly used faults when used for resilience testing and Chaos Engineering. From this, we can see that request data and access are common faults on distributed systems communicating via Application Programming Interfaces (API) [9], both [10] and [8] give an abundance of examples of multiple types of faults, such as network latency, total machine shutdown, and service discoverability.

Fault loads apply a model to a system. They are used to define the faults to be injected and are part of the planning of an injection experiment.

2.4 Injection techniques

There are two main high-level injection techniques, intrusive and non-intrusive. Non-intrusive techniques revolve mostly around proxies interacting with network communications between services [11]. This technique has the advantage of making services, in most cases, a black box, whoever sets up the experiment does not need to know the internals of each service. It usually involves injecting information into requests and responses, changing the contents of the messages, and altering response codes/statuses.

Intrusive techniques include changing the source code [6] and interfering with the infrastructure the code is running on, such as machines, physical or virtual [5]. While

this requires more information about the services themselves, especially when changing source code, it provides a greater range of possible faults and covers different possible fault points when compared with non-intrusive techniques.

In most of the published works, fault injection was automated and not done in every run of the experiment [12], with some cases of faults being injected by manual interaction with virtual machines [5].

2.5 Tools

There are two main types of tools relevant to this work. Tools that are similar to Defektor and tools that can inject faults/failures into systems.

2.5.1 Tools similar to Defektor

Frisbee [13] is a similar tool to Defektor, as a campaign manager capable of performing experiments on microservice-based applications running on Kubernetes. Experiments are defined on YAML files, and Frisbee exposes a set of actions that allow for managing resources and faults. The main advantage of Defektor in comparison to Frisbee is the flexibility due to the plugin system, meaning that the action and target lists are not fixed, like in Frisbee. ORCAS [14] is a tool for the generation of fault injection tests based on knowledge of the system architecture, patterns, antipatterns, and past test results. It has less overlap with Defektor than Frisbee, but they play a similar role in the campaign as an orchestrator. While Defektor operates through integrations with targets and other fault injection tools, ORCAS uses system information from tracing and a microservice architecture modeling language to generate tests and simulations that ORCAS itself executes. IntelliFT [15] is similar to ORCAS, as it uses already-known information to create new tests. It uses integration tests to create injections at the level of the network with Istio, causing crashes, disconnects, hangs, and overload. It also mutates its tests to generate new ones. Like Defektor, IntelliFT integrates with a different tool that performs the injection instead of performing the injection itself, but it does not use plugins like Defektor; instead, it always uses Istio.

Table 2.1: Tool comparison

Tool	**Target kinds**	**Mecanism of interaction**
Defektor	Agnostic	Plugins
Frisbee	Kubernetes environments	Direct access
ORCAS	Microservices applications	Direct access
IntelliFT	Microservices applications	External tool

Table 2.1 visually shows this comparison. It shows, for each tool, what kinds of targets they support and how they manage to interact with the target, i.e., how they inject

faults into the target system.

2.5.2 Injectors

Injectors are the tools mentioned in this work that are capable of performing injections into systems.

Gremlin [16] is one of these tools, capable of injecting network-based faults on microservices non-intrusively. It affects the networking interfaces of the microservice being tested and works based on recipes written in Python. Gremlin exposes 3 interfaces that can be used to create more complex faults, allowing for delays, aborts, and modification of messages.

Fault Injection Service (FIS) is Amazon Web Services (AWS) [17] fault/failure injector for their environments. It handles the injection, stopping experiments, and tracing. Works via templates that define actions from an already defined list by AWS itself, and each action is associated with a target group. Stopping configuration works based on metrics and time, to ensure that the experiments do not cause actual harm. FIS has a group of SDKs that enable programmatic access to the service.

Pumba [18] is a chaos engineering tool created by Alexei Ledenev. At the time of this research, it was the 5th repository on GitHub's Fault Injection topic, ranked by stars (2.6k), behind Litmus (4k), ChaosBlade (5.6k), Chaos Mesh (6.1k), and Istio (34.1k). It is built in Go and targets Docker containers, allowing basic Docker interference such as stopping containers, network interference, and container stress. Its main advantage lies in the simple setup and relatively simple requirements to function. In opposition to the other tools mentioned with more stars, it does not require Kubernetes in any way (despite supporting it), with some very basic requirements for the stressing and network commands. This makes it the most popular lightweight chaos-inducing tool. Pumba is used either through a Docker container or a CLI, with a command for each type of fault supported.

Chaos Toolkit [19] is a command-line tool written in Python that executes Chaos Engineering experiments on a wide range of platforms, enhanced by community-created extensions. These experiments are defined in JavaScript Object Notation (JSON) format, and the only requirement is to have Python, the *chaos* command-line tool, and the necessary extension installed. Similarly to Pumba, this tool is relatively lightweight regarding prerequisites, with the added advantages of extensibility and a wider target range, but less popular on the GitHub topic of Chaos Engineering with 1.8k stars. It also works via Command-Line Interface (CLI) like Pumba.

Toxiproxy [20] is a TCP proxy for chaos purposes, developed by Shopify, and has been used since 2014 by the creators. It's highly versatile as it can interfere with any endpoint, independent of the target type (Docker, Kubernetes, etc). The tool has many

clients in different languages, such as Java, Go, and Node. Beyond the client libraries, there is also a CLI that allows the usage of Toxiproxy without the need to change the source code. It was designed to be used mostly in testing, continuous integration (CI) pipelines, and development environments to prevent single points of failure.

3 Defektor

Defektor is a tool for automating and executing fault injection campaigns, with a focus on cloud environments and microservice applications. It originated in the Centre for Informatics and Systems of the University of Coimbra (CISUC) when a lack of tools with this kind of role and capabilities was noted [21]. This led to the development of a tool that integrates with different tools and machines to interact with a target system and with the goal of improving reproducibility and automation of this kind of campaign.

This chapter explains Defektor as a tool, as well as its objectives, functionality, usage, inner workings, and architecture.

3.1 Mission and functionality

Defektor was designed to work without being restricted by the target system or injection tools. For example, the AWS injection tool already allows for good reproducibility, but only works on AWS environments. To achieve this, there are two requirements: the ability to represent experiments independently of tools and target systems, and the ability to store these representations and reuse them.

Defektor represents experiments as plans, JSON objects that define the target system, injection tools configuration, injection configuration, workload, machines to execute the workload, and tracing configuration. These JSON objects are saved in a database to be used later in experiments with the same targets and faults. To stay agnostic regarding tools and targets, Defektor has a plugin system that handles the interaction with external elements. Figure 3.1 shows this aspect of Defektor, interacting with different kinds of tools, external to itself, through the plugins. The image shows Defektor using the Plugins to reach each of the external elements, namely the target, tracing tools, and injection tools.

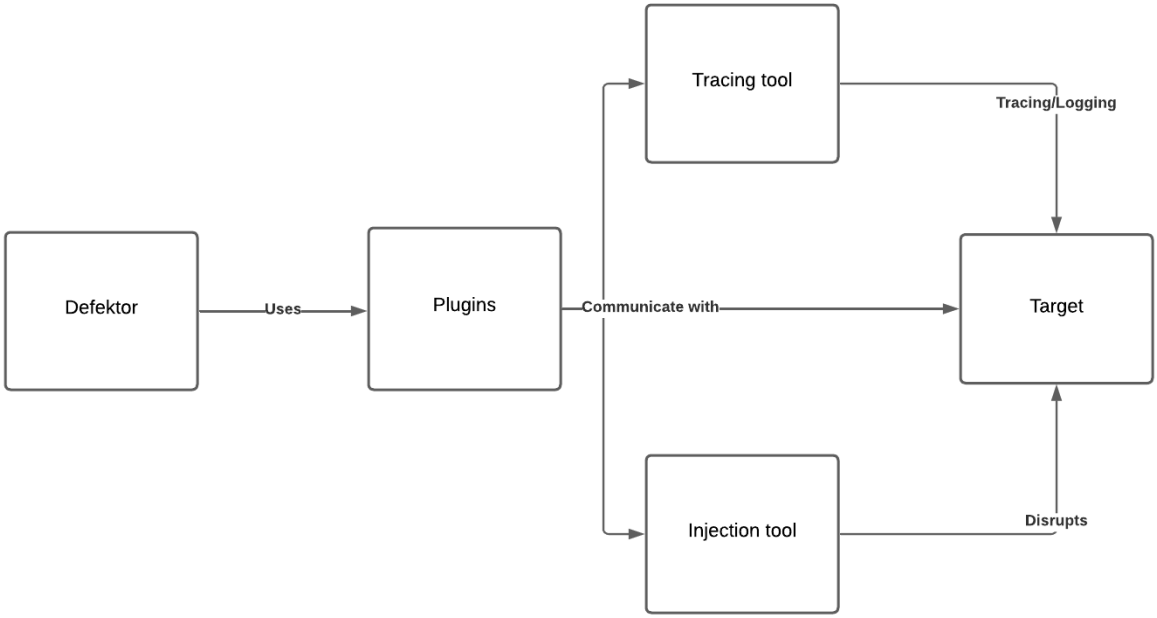

Figure 3.1: Representation of how Defektor interacts with external tools

Defektor is interactable via a console application that makes requests to a REST API exposed by Defektor. This makes the management of plans, target systems, and slaves available. This API is also responsible for executing plans and retrieving information about installed plugins and previous campaigns.

3.2 Requirements

Defektor's main objective is to enable better reproducibility and automation of fault injection campaigns. As an orchestrator, Defektor integrates with other tools that are capable of performing the actions necessary for a complete experiment. This includes fault injection, workload execution, logging and tracing, and communication with remote machines. Defektor is integrated with tools that perform fault injection to interfere with target systems and tracing/logging tools to collect data of an experiment. For example, Defektor can be integrated with a tool such as Pumba (described in section 2.5.2) to inject a fault such as latency to the responses of a service that is within a system, exercise the system and that service with the workload present on different machines and then, with an integration to something like Jaeger it can collect data.

Defektor has a set of 22 functional requirements that are already defined and prioritized, ranging from High to Low, as defined prior to this work. The 10 high ones define the ability to create and perform campaigns, encompassing plan creation and validation, the ability to inject faults, collect data, and run workloads. It also includes some management functionality, such as listing elements and campaign management.

Finally, the ability to extend how Defektor performs fault injection (i.e., plugin development) is also a high functional requirement. Medium requirements expand on this, such as support of other kinds of plugin, for data collection and communication with target systems, the ability to use an already existing plan for a campaign, and management functionality for other elements such as slave machines and plugins (i.e., Create, Read, Update and Delete (CRUD) operations on the various elements Defektor uses. Finally, the low-priority requirements focus on the parallelization of tests and injections and improvements to how development is made with Defektor.

Table 3.1 shows these requirements [21].

Table 3.1: Defektor's requirements

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

A High priority requirement is considered mandatory to consider Defektor a working tool. Medium requirements represent a significant improvement in the tool. Low priority requirements are only enhancements for already developed functionality.

This was the initial set of requirements that led to the creation of Defektor and has mostly been achieved, but FR-13, FR-15, FR-20, FR-21, and FR-22 are not implemented. This is addressed in further detail in section 4.3.

3.3 Architecture

Defektor's API can be separated into 3 parts:

- Core - handles the requests, the database access
- Campaign management - the configuration and execution of a campaign
- Plugins System - handles plugin detection, initialization, and interaction

Figure 3.2 shows the architecture of Defektor at the time this work started. It shows the relationship between the core and the campaign running and management portion of Defektor, as well as how the campaigns then get and activate the different plugins needed. On the plugin system part, it shows relationships between the different kinds of plugins and the 2 types that Injektors must define to help in the tool picking process of a campaign.

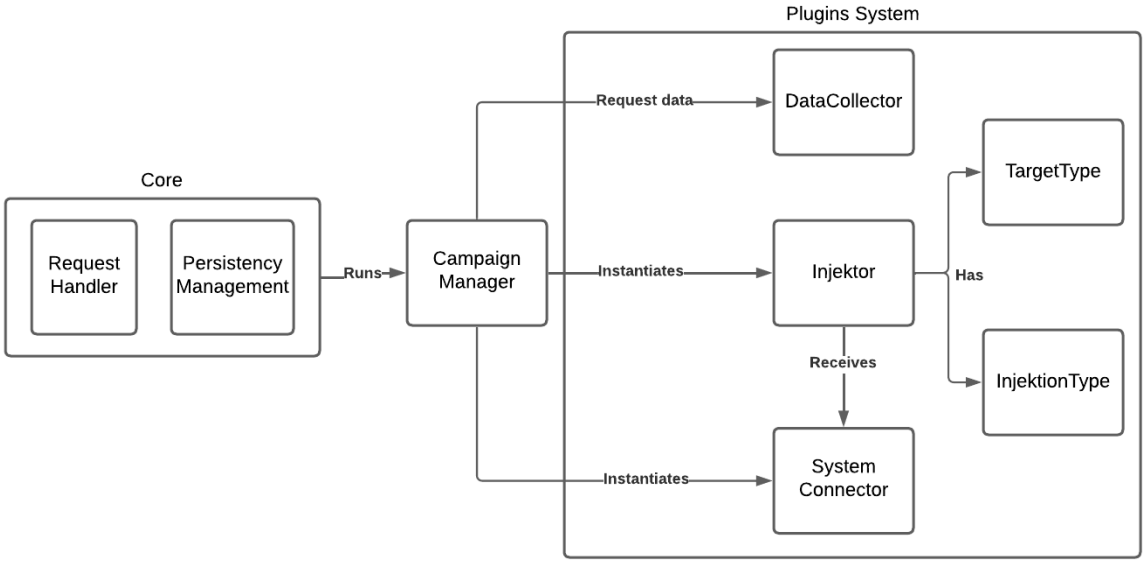

Figure 3.2: Defektor's previous Architecture

The execution of a new plan starts with a request that saves the request to the database. The saving of a new plan triggers a new campaign to be configured and run. A complete flow would be:

1. Receive request to add a plan
2. Save the plan - returns information about the added plan to the client
3. Begin a new campaign (configure)
4. Configure plugins with the information from the plan
5. Run the green run with the workload configured from the plan
6. Activate the plugin for injection

7. Rerun the workload with the injection performed
8. Gather information from tracing via a plugin
9. Finish campaign

Figure 3.3 shows the example of a plan that would trigger the steps listed above. For the sake of space, it was transformed into YAML, as JSON and YAML are convertible to each other.

3.3.1 Core

The core of Defektor is a Springboot application with a generated model from an OpenAPI spec file. It is responsible for dealing with HTTP requests, providing endpoints to manage the different kinds of objects within Defektor, such as plans, plugins, and slave machines. This also includes a repository implementation that handles the storage logic used by each of the endpoints.

3.3.2 Campaigns

Plans configure a set of plugins, target systems, and workloads to initiate a campaign. Defektor uses Docker to represent the workloads, which gives it greater portability in terms of workload runners but requires these machines to have the Docker image already present. Defektor instantiates the plugins and runs the workload once before injecting and then one more time after the injection has been performed. Finally, it ends the campaign cycle by using the data collector to gather the traces or logs from the system and saves them into a file. If at any point in this process, there is an error, the campaign is aborted.

3.3.3 Plugins

When Defektor is initialized, it gathers Java Archive (JAR) files from a plugins folder. Each JAR contains a class that represents a plugin following the contract defined by Defektor. This class is registered for later instantiation, which is done via a factory that creates instances of plugins.

There are 3 types of plugins:

- Injektors - Integration method with 3rd party fault/failure injection tools
- SystemConnectors - Abstract communication logic with the target machine/environment
- DataCollectors - Collect data from tracing and logs generated by the workload and injection

```yaml
name: <experiment_name>
system:
  name: <system_type>

injektions:
  - totalRuns: <integer>
    ijk:
      name: <fault_name>
      params:
        - key: <param_key>
          value: <param_value>

    workLoad:
      image:
        user: <image_user>
        name: <image_name>
        tag: <image_tag>
      cmd: <command>
      env:
        - key: <env_key>
          value: <env_val>
      replicas: <integer>
      slaves: <integer>
      duration: <duration_seconds>

    dataCollector:
      name: <collector_name>
      params:
        - key: <collector_param_key>
          value: <collector_param_val>
```

Figure 3.3: Plan example - transformed to YAML

4 Development

Being already in development, Defektor has a different set of requirements as a project. In this project, the objectives are focused on maturing the tool and ensuring that it is fit to be applied outside of the experimental space. This chapter delves into these objectives and how to achieve them.

4.1 Objectives

The main target at this stage of Defektor is to prove its ability to work with third-party tools, namely injectors, tracing tools, and target systems. If Defektor can be integrated with any injecting tool, then every target system is accessible. As of the start of this work, Defektor had been tested with a sample microservices application (Roboshop) and with Istio and Jaeger.

The goal of this project was then to test Defektor with third-party injection tools and systems, developed independently from Defektor itself. This is achieved through case studies where Defektor is used to orchestrate a fault injection campaign. To achieve this, it is also intended to develop Defektor and ensure it can be extended effectively with plugins, and provides the value it proposes: repeatable and consistent fault injection campaigns.

There are then three steps to achieve this: analyse Defektor for any shortcomings regarding its value proposition and possible improvements, fix the aspects found in the analyses, and mount and execute one or more case studies that attest Defektor's functionality and agnosticism over tools and targets.

4.2 Technical analysis

At the beginning of the work described here, Defektor was analyzed with the intent of understanding some technical aspects. Understanding how Defektor works internally, what architecture it follows, and how it was implemented are important aspects to assess before making changes to the project, and this was part of the technical aspects analyzed here. It was also through this analysis that missing features and possible improvements were detected.

The outputs of this technical evaluation are presented in Table 4.1.

Table 4.1: Improvement opportunities found

Improvement Point	**Description**
Improve Reproducibility	Allow users to run a campaign from an already existing plan
Simplify Injektors	Make Injektor plugins simpler to develop and maintain
Improve plan structure	Restructure the plan object to facilitate reading and writing

These improvement possibilities were identified in two ways: by comparing the requirements (presented on section 3.2) with the functionality present on the code base and by analyzing the goals of this work and what would be necessary to achieve this.

The first improvement, improving reproducibility, comes directly from a requirement found not to be met when analyzing the code, as there was no functionality implemented that supported re-running an existing plan, other than submitting the same plan again, creating duplication. To fulfill this requirement, a new endpoint was developed that would take in an identifier of an existing plan and start a campaign with it.

The latter two improvements are more focused on the development of plugins, an integral part of making Defektor a more relevant tool. For Defektor to support more systems, more cloud providers, and different deployment methods, the necessary changes should not be on Defektor itself; it is achieved through the development of more plugins. With this in mind, the development of a plugin for Defektor should be as simple as possible and free of excessive obstacles.

4.3 Improvements

After the technical analysis, developments were made to improve the topics from table 4.1.

4.3.1 Simplify Injektors - Make SystemConnectors optional

This improvement was made to make the development and maintenance of Injektor plugins easier. Injektors are the plugins that integrate Defektor with a fault injection tool, and therefore are a crucial part of Defektor's agnostic aspect and ability to support new technologies.

Injektors required a SystemConnector to be provided to them as a dependency. The developer of the plugin would define the type of connector required to use the Injektor, but had no guarantee that this type of plugin would be passed at runtime, as Defektor would look at the target system in the plan and pick a connector associated with it, as

seen in section 3.3. This made Injektors harder to develop and forced multiple projects when an Injektor had a very specific form of communication that was reusable.

The change was to make SystemConnectors a part of the Injektor plugin. Instead of receiving a SystemConnector from Defektor, which means that the developer of the Injektor directly references the SystemConnector desired, if any. Since the SystemConnector is encapsulated in the Injektor, the configuration is also made within the Injektor part of the plan.

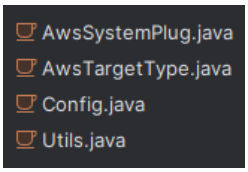

Figure 4.1: Files required for a SystemConnector

Figure 4.1 shows the files used for an initial AWS SystemConnector, developed before the start of the work described in this document. It shows the need for multiple files, although some are not mandatory (namely, Utils.java); it adds a lot of unnecessary code and information in comparison to what is necessary. Figure 4.2 shows the interface of an adapter (used in the Injektor developed and explained in section 4.5.3, with exactly what this Injektor needs to achieve its objectives, making this communication logic have a very high degree of coupling with the Injektor plugin, and making it not suitable for reuse.

```
public class FISAdapter {

    private final AWSFIS client;
    private final FISData data;

    public FISAdapter(FISData data)

    private HashMap<String, CreateExperimentTemplateTargetInput> getTemplateTarget()

    private HashMap<String, CreateExperimentTemplateActionInput> getActions()

    public String createExperiment()

    public void runExperiment(String id)

    public void stopExperiment(String id)

    public void deleteExperiment(String id)
}
```

Figure 4.2: AWS Adapter for Injektor

4.3.2 Improve plan structure - Plugin Registry

A new registry was added to track Injektor plugin configurations, mapping object instances to a name specified in the plan. This enhancement allows the plan to generate a sequence of plugins, which can be easily accessed by name. Campaigns leverage this registry to retrieve specific instances of an Injektor as needed.

Comparing figure 4.3 to 4.4, it is possible to see how the big object for the injections was broken into two, with the information regarding the plugin and how to inject the fault extracted to a different object that is then referenced, making the injections easier to understand.

There are two parts to this. First, reading the plan is different now. The campaign had to be adapted to read a new object that has all the Injektor configurations, and then store them in the registry. And also, the registry itself, which is a map of maps, that relates injection type and name to an instance of an Injektor. It is then a map of fault types, each with its list of Injektors. Each list is also a map for faster search based on the configuration name, which is set in the plan. Beyond registering new configurations, this class allows the fetching of items to be used for injection in 2 ways:

- Via name - it searches a specified Injektor map of Injektors, based on the fault type, to find one with the provided name

- Via target type - it searches the specified Injektor list to find one that supports the specified target, returning the first found

If no suitable item is found, both methods return null, which will cause an error further ahead in the campaign, causing it to stop.

This update simplifies the injection definition, removing fields from an already large object, which now has a static amount of thirteen fields. It also enables future development, such as automatic Injektor selection based on the type of injection, a mechanism allowing Defektor to automatically select an Injektor from the ones configured in the plan based on the supported injection types. This has changed Defektor's architecture, as shown in Figure 4.5. In the figure, it is possible to see the changes in the plugin system regarding how the campaign management module gets Injektor plugins and how these are instantiated via the newly added Plugin Registry. It also shows how the relationship between the Injektor plugins and the SystemConnector plugins changed, now being directly managed by the Injektor.

```
{
    "name": "Order 66",
    "system": {
        "name": "kubernetes"
    },
    "injektions": [
        {
            "totalRuns": 1,
            "ijk": {
                "name": "httpabort",
                "params": []
            },
            "workLoad": {
                "image": {},
                "cmd": "",
                "env": [],
                "replicas": 1,
                "slaves": 1,
                "duration": 240
            },
            "dataCollector": {
                "name": "jaeger",
                "params": []
            }
        }
    ]
}
```

Figure 4.3: Old plan structure

```
{
    "name": "Pumba!",
    "system": {
        "name": "docker"
    },
    "injektors": [
        {
            "name": "PumbaDocker",
            "pluginName": "PumbaInjektor",
            "params": "'action': 'kill','containers':..."
        }
    ],
    "injektions": [
        {
            "ijk": "PumbaDocker",
            "totalRuns": 1,
            "workLoad": {
                "image": {},
                "replicas": 1,
                "slaves": 1,
                "duration": 240,
                "env": []
            },
            "dataCollector": {
                "name": "jaeger",
                "params": []
            }
        }
    ]
}
```

Figure 4.4: New plan structure

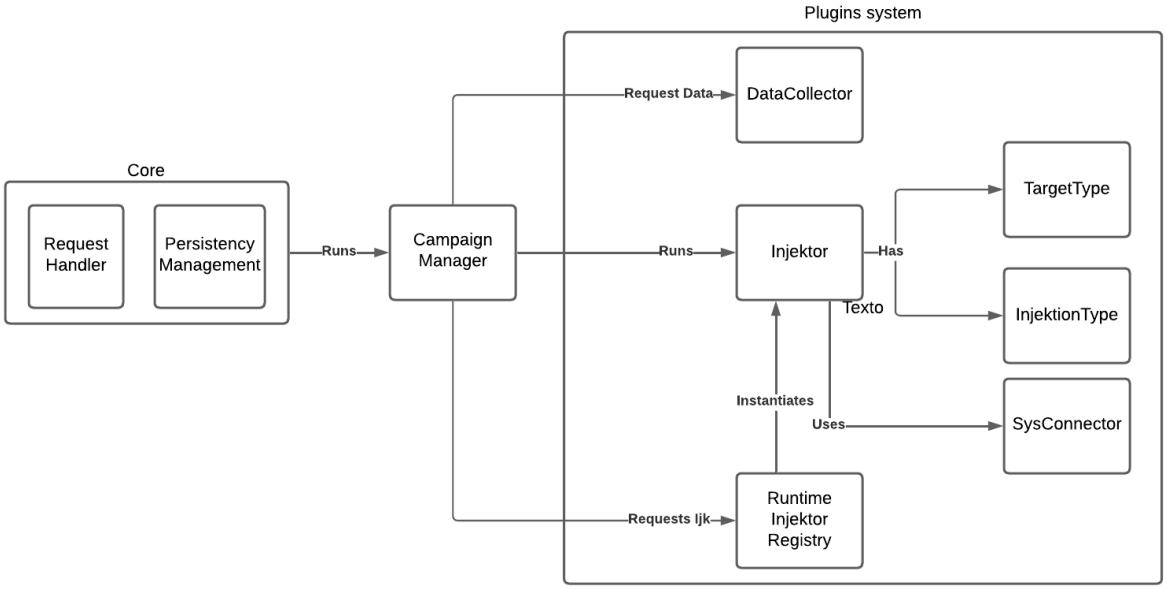

Figure 4.5: Defektor's new Architecture

4.4 Other developments

There were two minor developments made to facilitate development and fix certain issues found during development, not related to Defektor directly.

Firstly, the existing implementation of a Jaeger Data Collector plugin had to be updated to work with a more recent version of Jaeger as it had multiple updates after the initial development of the plugin. This plugin makes an HTTP request to get the tracing logs from the machine running Jaeger. This endpoint has been updated to set the timeframe to be obtained using a lookback mechanism instead of a start and end mechanism. Note that this is a compatibility change and does not impact the plugin functionality.

Secondly, to ease the development of more complex plugins, the Injektor object definition in Defektor's API, used in the plan, was updated to have the parameters be a string representing a JSON object instead of a list of strings. This facilitates the development of plugins that have a high parameter count, with different types, by making the structure of the parameters easier to track in the plan JSON.

4.5 Plugins developed

Integrating Defektor with other tools is a vital part of this project as it provides the means to show Defektor in action with 3rd party tools. The process of integration with Defektor consists of the development of plugins that make the tool's functionality available to Defektor. Here is an explanation of what can be achieved with these plugins and how they were developed.

4.5.1 Injektor scope

Injektor plugins developed earlier were focused on a single fault or failure on not on an injection tool. For example, there was a ShutdownMachine Injektor that did not even use any tool, it simply sent a shutdown command via SSH session to the target machine. Another example, attempted to integrate the shutdown machine action from FIS, but that was the only action that it supported, so, for example, to enable the reboot machine, a whole new plugin would be necessary, plus the repetition of code to accommodate other configurations, such as alarms that create alerts and stop chaos experiments.

The plugins developed in the scope of this work were focused on tools and not on specific failures or faults. For example, a plugin that is developed for FIS intends to enable the usage of as much of FIS functionality as possible, instead of a single action. Note that this is just a change in the way plugins are idealized and developed, but there was no actual change to the way Defektor works and handles plugins.

4.5.2 Tooling selection criteria

To achieve the case studies showing Defektor working with external tools, it has to be integrated with such external tools. There were multiple possibilities, namely AWS FIS, Chaos Toolkit, Pumba, and Toxiproxy. Tools such as Istio were discarded due to their complexity, specifically the requirements for target system deployment, including the use of Kubernetes, which was not necessary for testing Defektor but had significant implications for costs and time.

The criteria used to select which tools to use were the following:

1. Compatibility - the tools selected must be integrable with a Java application
2. Target systems - the tools selected should provide access to different kinds of systems
3. Supported faults - the tools selected should provide a wide range of possible faults/failures
4. Popularity - the tools should be relevant in the fault/failure injection and Chaos Engineering spheres

These requirements don't have to apply to a single tool; multiple ones can be selected to provide a wider range of possible targets and faults.

Looking at the four possibilities, already analyzed in the section 2.5.2, there are two that provide more value upfront, namely FIS, providing direct access to cloud infrastructure-based faults and failures, and Pumba, one of the most popular tools of this kind, that has more possible faults and is generic enough that any target system using Docker, a very common practice in microservices applications, is a possible target.

FIS is fairly unique given the fact that it is native to a cloud provider. It provides quick and simple access to multiple faults and failures on systems hosted on AWS's services, which puts it in a position that does not compete with the other, more generic, tools.

Pumba, on the other hand, competes with Toxiproxy and Chaos Toolkit. Toxiproxy, despite also being quite popular and generic, only provides faults based on networking, such as delays and simulated error responses, which makes it a little shallow. For Chaos Toolkit, it's very similar to Pumba, but less popular, and it also depends on extensions developed by the community. In comparison with Pumba's closed package of possible faults that englobes an entire category of possible faults for Docker containers, Chaos Toolkit's more extensible environment raised more challenges without a benefit that justified it.

Together, FIS and Pumba provide a wide range of possible faults and failures. FIS provides direct access to the cloud and hardware failures, while Pumba provides a multitude of Docker-based generic faults applicable to nearly any system. As a consequence of the improvements made to Defektor, present previously, the development of these plugins is very simple and they are very lightweight.

4.5.3 AWS Fault Injection Service Plugin

AWS FIS was integrated via an Injektor that communicates with an environment to run experiments, a type of asset within the AWS ecosystem that is used to perform chaos experiments/fault injection on AWS environments. It was developed with the Java SDKs provided by AWS that allow these experiments to be managed and run programmatically.

The plugin has 3 main parts: the glue with Defektor (implementation of the Injektor interface), a JSON parser for the properties passed via the plan, and a communication module that sends the data parsed from the plan to the environment. The Injektor interface has the functionality of giving Defektor a way to activate the tool and to provide information on what targets and faults are supported. When executed, the plugin can create an experiment or use an already existing one. In the case of creation, there is also a setting that deletes the created experiment. This is all made through commands sent to the remote environment in AWS.

The minimum information that is needed to execute is the credentials, the region, and either the existing or the data, such as action names and target names, to create a new experiment template.

4.5.4 Pumba Plugin

Pumba's injektor is a small application that communicates with a target machine to send commands via SSH that execute Pumba when activated by Defektor. This deve-

lopment was made without any SDKs, making use only of an SSH library. There was an attempt at using an already existing SSH SystemConnector, but it was outdated, and upgrading it was not successful in a reasonable amount of time.

The plugin is divided into 3 parts: the injektor interface, which provides the injektor's functionality to Defektor; the parser that parses the data from the plan; and the communication part, which sends the command to the machine via an SSH session. The plugin has very little setup on the plan, needing only 5 fields: the Pumba's command to be sent, the address of the target machine and the port, and the credentials, user, and path to the SSH key.

In the case of Pumba, the prerequisite of it already being installed on the target machine was assumed to be true, as without this, the command would not run.

5 Experimental Validations

As per the objective of this work, we want to verify the ability of Defektor to be used for Chaos Engineering and fault injection campaigns, by orchestrating third-party tools to interact with microservices applications, possibly in different networks, and be able to exercise them, and then collect data that shows the effectiveness of the injection and how the system handled it. This chapter presents the strategy and case studies used to validate Defektor, the results of the experiments, and how the changes made to Defektor impacted the development, preparation, and execution of these case studies.

5.1 General Validation Strategy

To effectively test Defektor, several case studies were set up using a microservices application and various forms of deployment, each presenting different opportunities for fault injection with different tools. These case studies enable the use of other tools and different infrastructures to ensure that Defektor is tested in ways that demonstrate its ability to be agnostic of the third-party aspects of a fault injection campaign, namely the target system and injection tools.

Each case study coincides with a campaign, with the following set up: there is a target application which has been instrumented to have logging of tracing of metrics that represent the health of the application, such as latency; a workload that activates the system or some parts of it is defined and is spread along clients that can then execute it; a fault model is defined, stating the faults to be inject; the system is then exercised without any injection, to set what the normal behavior is (known as a golden run), the injection and another run of the workload follow this.

To achieve this, an experiment setup was organized, where we used a microservices application developed by a third-party, deployed in two different ways, one in a local network setup and one in a cloud environment of AWS, both using just Docker containers and networks, which provides an example for both cloud-based hosting and on-premises solutions. This application has both requirements of having a microservices-based architecture and being developed independently of Defektor. The application used simulates an online marketplace, including a bank and an advertising platform. It is divided into 3 parts: the bank, the advertising platform, and the marketplace itself, and was developed in the scope of a different master's project [22]. Each of these subsystems has a series of services that communicate with each other and with their

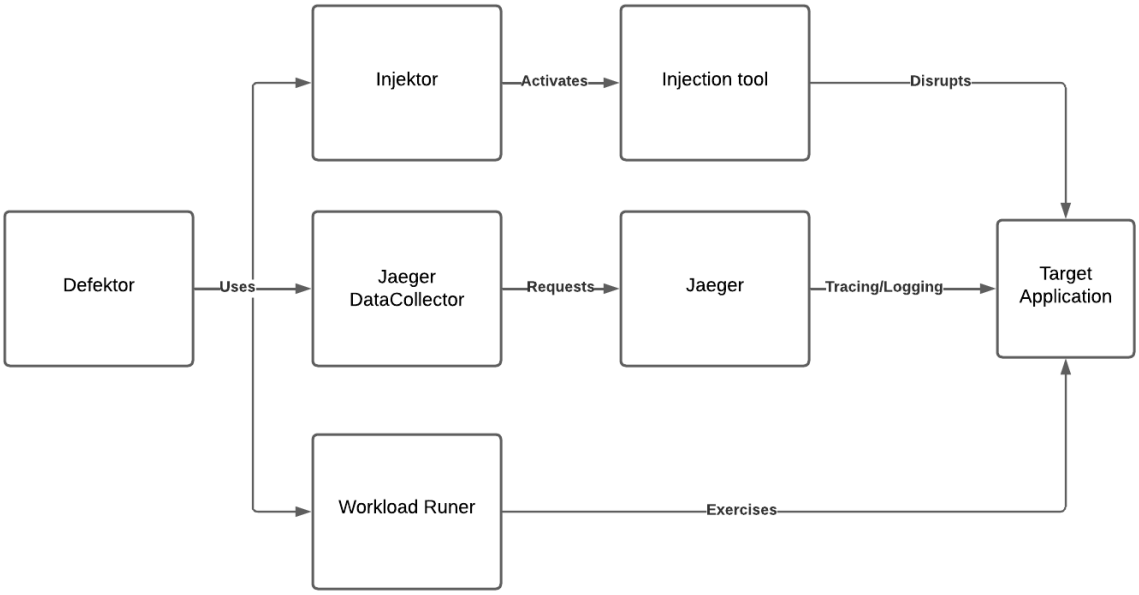

Figure 5.1: Campaign elements

supporting databases. In total, there are nine services, each with its own Docker container and developed using its specific technology stack. The application had to be instrumented to support Jaeger's OpenTelemetry-based tracing and logging.

Figure 5.1 shows a generic diagram for both case studies. The Injektor and the Injection tool change depending on the case study setup, between Pumba and FIS. The figure shows how Defektor reaches the target workload runners (to exercise the target) and the Injketor plugin and injection tool to disrupt it. It also shows how Defektor collects data back through Jaeger and a Jaeger DataCollector plugin. It is also possible to see how Defektor can work without the presence of a SystemConnector between the Injektor plugin and the target system.

Regarding fault models used, they were adapted to each deployment environment. The cloud case study makes use of the infrastructure the target is running on by turning off the machine, simulating an outage of the cloud provider. In the case of the local network example, the chosen fault was to inject a delay into the responses of one of the services, simulating a degradation. These are 2 realistic examples of possible faults that can disrupt the delivery of functionality by a microservices application on each of these deployment environments. To achieve this, the cloud case study makes use of the native injection tool provided by AWS, FIS, which can interact with the machines where the system is running. For the on-premises one, Pumba was chosen as it is very focused on disrupting Docker-based systems, regardless of the environment.

5.2 Local network case study

The local case study performs the campaign with the target application running on a physical machine in the same local network as Defektor.

In this example, the Pumba injektor plugin is used to inject a delay of three seconds into the response of a database Docker volume, which makes the requests to that service also take longer to be responded to. In this case, the notifications service was selected, which can be activated by having clients request their notifications.

The workload repeatedly requests the notifications, simulating multiple clients requesting their notifications over time, executed from a single client machine.

The timespan of this campaign is one minute per run, meaning that the system is exercised for 60 seconds without any fault, and then another 60 seconds with the fault. In accordance with the fault model selected, it is expected that at some point, the request duration goes from milliseconds to more than three seconds. The data collected by Defektor, generated by the application and Jaeger, showed 60 seconds of normal behavior, with each request taking around 3 milliseconds to be responded to. After around 60 seconds, it is possible to see that requests start to take more than 3 seconds to be responded to, in line with the expectations. This shows that Defektor, through Pumba, the client machines, and Jaeger, was able to interfere with the target, was able to exercise it, and then collect the data showing what happened within the system.

It can then be concluded that Defektor was able to perform an entire fault injection campaign on a target system running on the same local network with success. Moreover, it was possible to visually verify in Jaeger UI the impact on the response time, although this is not considered in the validation of Defektor, as it is not a replacement for Defektor collecting this data.

Figure 5.2 shows a trace where the delay was not applied, where the duration is still around 3 milliseconds, while figure 5.3 shows a later trace, of the a request to the same container, but now the duration being a bit more than 3 seconds, the delay injected (both images were cut and do not show the entire trace for the sake of space). These traces were taken from a log file downloaded from Jaeger, which was set up for tracing in the target application.

```
"traceID": "677d3fffc4bdb97c814e61e12860e2db",
"spanID": "a06cdb1c388e3d67",
"operationName": "GET /notifications-admin/",
"references": [],
"startTime": 1729629294348000,
"duration": 3265,
"tags": [
  {
    "key": "http.url",
    "type": "string",
    "value": "http://host.docker.internal:7001/notifications-admin"
  },
```


Figure 5.2: Trace of a request without response delay


```
"traceID": "8e804728a2e47309b23bbb3c4e87d4bc",
"spanID": "b24774fea8c5002e",
"operationName": "GET /notifications-admin/",
"references": [],
"startTime": 1729629306514000,
"duration": 3017411,
"tags": [
  {
    "key": "http.url",
    "type": "string",
    "value": "http://host.docker.internal:7001/notifications-admin"
  },
```


Figure 5.3: Trace of a request with response delay

5.3 AWS environment case study

In the remote example, the same target application is deployed to a cloud environment on AWS, with the databases and the services running on two different machines.

Here, FIS was used to turn off the machine where the databases were running, which should cause problems in the responses from the services themselves. As explained in 4.5.3, it works with experiments, composed of actions and targets. An experiment

was then created with the action to shut down the target machine, with the databases machine as the target.

Similarly to the local case study, the notifications service was selected to be exercised by the workload, with the workload, again, requesting the notifications repeatedly. The golden run takes 60 seconds, and after the injection is performed, it will be exercised for 60 more seconds, but the system will not revert to the original state, given the nature of the failure injected.

The data collected from Jaeger showed 60 seconds of normal responses to the requests, and after around 60 seconds, the requests started to have spans with errors. In this case study, it is expected that after the initial 60 seconds, the traces of the requests start including errors when trying to communicate the SQL query to the database volume. In the data collected via Jaeger, there are traces with spans that represent the communication from the service to the database volume. In these, it is possible to see that, according to the expectations, error spans started appearing, showing errors when trying to run SQL commands in the database volume, showing that Defektor was capable of conducting an entire campaign with a target hosted on a cloud environment and with different injection tools. Also, beyond visual validation on Jaeger UI, it is possible to validate the injection by checking that FIS has one more experiment run, and the state of the target machine is stopped. Figure 5.4 presents a trace that shows a successful request (image does not show the entire trace for the sake of space). The tags show that the request is an SQL request to the notifications database, confirming it is a request from the workload.

Contrary to the figure 5.4, figure 5.5 shows a trace where the request to the database failed. Through the tags, we can confirm that it fails when attempting to reach the same IP as the successful trace, and the start time shows that it is a bit after the successful trace, after the golden run ended, and the injection was performed.

Both these traces were taken from a trace log file downloaded by Defektor from Jaeger, which was set up on the target application.

5.4 Developments impact

The developments made to Defektor, explained in section 4.3, made the preparation, planning, and execution of the case studies easier. The development of the plugins, especially the FIS plugin, was made easier because the communication logic stayed close to the logic of the plugin, avoiding the need for explicit casting, the maintenance of multiple projects (in case of an SDK update by AWS, for example), and extra setup on the plan to configure an extra plugin. The changes made to the plan structure also helped with iterating faster and with fewer errors in the multiple attempts that these case studies took, effectively reducing the necessary time and costs to perform them. By

```
"traceID": "8f828a826a1b84e78cddaed28a7748d9",
"spanID": "6b15e47a503751aa",
"operationName": "pg.query:SELECT notifications_db",
"references": [
  {
    "refType": "CHILD_OF",
    "traceID": "8f828a826a1b84e78cddaed28a7748d9",
    "spanID": "58eee15e962229f8"
  }
],
"startTime": 1727461559693000,
"duration": 1452,
"tags": [
  { "key": "db.system", "type": "string", "value": "postgresql" },
  {
    "key": "db.name",
    "type": "string",
    "value": "notifications_db"
  },
  {
    "key": "db.connection_string",
    "type": "string",
    "value": "postgresql://172.31.7.170:7008/notifications_db"
  },
```


Figure 5.4: Trace showing a successful request

```
"traceID": "743da4935b0747b55dfa3ee5c227c189",
"spanID": "60b600c7287adf89",
"operationName": "tcp.connect",
"references": [
  {
    "refType": "CHILD_OF",
    "traceID": "743da4935b0747b55dfa3ee5c227c189",
    "spanID": "8f18b67f8c59a34d"
  }
],
"startTime": 1727461563079000,
"duration": 1195,
"tags": [
  { "key": "net.transport", "type": "string", "value": "ip_tcp" },
  {
    "key": "net.peer.name",
    "type": "string",
    "value": "172.31.7.170"
  },
  { "key": "net.peer.port", "type": "int64", "value": 7008 },
  { "key": "span.kind", "type": "string", "value": "internal" },
  { "key": "otel.status_code", "type": "string", "value": "ERROR" },
  { "key": "error", "type": "bool", "value": true },
  {
    "key": "otel.status_description",
    "type": "string",
    "value": "connect ECONNREFUSED 172.31.7.170:7008"
  },
```


Figure 5.5: Trace showing an unsuccessful request

having the injection tooling setup separated, it was easier to separate changes regarding injection logic, such as fault load, from aspects such as tracing or workload.

This provides an example of how those changes made the usage of Defektor and the development of plugins easier and faster, without losing any of the necessary functionality of the tool.

6 Conclusions

In this work, Defektor was presented, a tool that manages fault injection campaigns to achieve greater reproducibility, more consistent results, and more automation, while staying agnostic of injection tools, target systems, and infrastructures.

After analyzing Defektor, weak points were identified, as well as missing functional requirements and non-functional requirements. This led to updates and improvements on the tool to reach a higher level of completeness. The ability to re-run a plan was added, the structure of the plan was improved with the addition of a plan plugin registry, and the plugins to inject faults were simplified.

The developments made to Defektor improved the development of the plugins used and made the preparation of plans considerably easier to make and maintain. With special relevance in the development of the FIS plugin, it was much simpler to handle the communication with the AWS services when compared to the previous structure, where a SystemConnector had to be used, as it used fewer files, less code, and the components with higher levels of coupling got closer. In the case of the plan structure change, it made the multiple iterations of the plans for the case studies easier to work with. Having the injection tooling setup separate from the rest of the injection definition made it harder to make mistakes and easier to change what was injected and how it was injected.

With the upgrades made, Defektor was integrated with two new tools, Pumba and FIS from AWS, which allow Defektor to inject faults into a vast number of systems, through their infrastructure with FIS and container subsystems with Pumba. The FIS plugin allows Defektor to create an experiment or use an already existing one, which makes every possible configuration available and therefore complete integration of the native fault injection tool of one of the most used cloud providers right now. Pumba's plugin, on the other hand, sends a Pumba command via SSH to the target machine. By sending the entire command, every option is available to Defektor, providing full support to one of the most well-known Docker fault injection tools. And with these tools, it was possible to test Defektor in two scenarios that simulate two different use cases, a remote one, where the target system is running in a cloud, and therefore FIS was used, and a on-premises or local one, where Pumba was used. In both cases, Defektor was able to inject the fault, activate the system with the workload, and collect data that showed the results of the interference.

6.1 Future directions

In terms of developments, certain functionalities would benefit Defektor greatly, namely, a way to protect credentials used in the plan, as these are plain text in the plan.

Regarding the tests, Defektor would benefit from a wider range of tests, with different systems, cloud providers, different infrastructures, and injection tools. There are a lot of ways to develop and deploy software, and the two case studies do not cover such a range.

Bibliographic references

[1] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015.

[2] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., 2015.

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[4] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[5] A. Nandi, A. Mandal, S. Atreja, G. B. Dasgupta, and S. Bhattacharya, "Anomaly detection using program control flow graph mining from execution logs," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 215–224.

[6] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 683–694.

[7] M. Ma, J. Xu, Y. Wang, P. Chen, Z. Zhang, and P. Wang, "Automap: Diagnose your microservice-based web applications automatically," in *Proceedings of The Web Conference 2020 (WWW '20)*, 2020, pp. 246–258.

[8] F. Silva, V. Lelli, I. Santos, and R. Andrade, "Towards a fault taxonomy for microservices-based applications," in *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*, 2022, pp. 247–256.

[9] J. Aué, M. Aniche, M. Lobbezoo, and A. van Deursen, "An exploratory study on faults in web api integration in a large-scale payment company," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 13–22.

[10] S. Brüning, S. Weissleder, and M. Malek, "A fault taxonomy for service-oriented architecture," in *10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, 2007.

[11] N. Wu, D. Zuo, and Z. Zhang, "An extensible fault tolerance testing framework for microservice-based cloud applications," in *Proceedings of the 4th International Conference on Communication and Information Processing*, 2019, pp. 38–42.

[12] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, "Cloudseer," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 489–502.

[13] F. Nikolaidis, A. Chazapis, M. Marazakis, and A. Bilas, "Frisbee," in *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems*, 2021.

[14] A. van Hoorn, A. Aleti, T. F. Dullmann, and T. Pitakrat, "Orcas: Efficient resilience benchmarking of microservice architectures," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2018.

[15] Z. Long, G. Wu, X. Chen, C. Cui, W. Chen, and J. Wei, "Fitness-guided resilience testing of microservice-based applications," in *2020 IEEE International Conference on Web Services (ICWS)*, 2020.

[16] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: Systematic resilience testing of microservices," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[17] A. W. Services, "Aws fault injection simulator," 2022, available: https://aws.amazon.com/fis/.

[18] A. Ledenev, "Pumba: chaos testing tool for docker," 2016, available: https://github.com/alexei-led/pumba.

[19] C. Toolkit, "Chaos toolkit documentation," 2022, available: https://chaostoolkit.org/.

[20] Shopify, "Toxiproxy: A tcp proxy to simulate network and system conditions," 2014, available: https://github.com/Shopify/toxiproxy.

[21] G. Baptista, "Failure injection in microservice applications," M.Sc. thesis, Universidade de Coimbra, Coimbra, Portugal, 2021.

[22] T. Simões, "A microservices application for research purposes," M.Sc. thesis, Universidade de Coimbra, Coimbra, Portugal, 2023.

Anexxes

Anexx A - AWS FIS Shutdown EC2 machine plan

```
{
    "name": "AWS:FIS",
    "system": {
        "name": "AWS"
    },
    "injektors": [
        {
            "name": "AWS:FIS",
            "pluginName": "AWSInjektor",
            "parameters": "{\"id\":\"<ID>\",␣\"key\":␣\"<KEY>\",␣
              ↪ \"secret\":␣\"<SECRET>\",␣
              ↪ \"region\":\"<REGION>\"}"
        }
    ],
    "injektions": [
        {
            "ijk": "aws:fis:injektor",
            "type": "aws:fis:ec2",
            "totalRuns": 1,
            "workLoad": {
                "image": {
                    "user": "neves",
                    "name": "get-admin-notifs-aws",
                    "tag": "latest"
                },
                "dockerProps": [
                    "-e␣HOST='<HOST>'",
                    "-e␣DURATION=60"
                ],
                "delay": 0,
                "cmd": "",
                "replicas": 1,
                "slaves": 1,
                "duration": 60,
                "env": []
            },
```

```
            "dataCollector": {
                "name": "jaeger",
                "parameters":
                    {
                        "host": "<TRACES_HOST>",
                        "lookback": "30m"
                    }

            }
        }
    ]
}
```

Annex B - Pumba network delay plan

```
{
    "name": "Pumba!",
    "system": {
        "name": "docker"
    },
    "injektors": [
        {
            "name": "PumbaDocker",
            "pluginName": "PumbaCommandInjektor",
            "parameters": "{\"command\":␣\"pumba␣netem␣--duration␣
              ↪ 1m␣--tc-image␣gaiadocker/iproute2␣delay␣--time␣
              ↪ 3000␣tese-hotrod-1\",␣\"user\":␣\"neves\",␣
              ↪ \"host\":␣\"192.168.1.85\",␣\"port\":␣22,␣
              ↪ \"key\":␣\"C:/Users/ruizi/.ssh/id_rsa\"}"
        }
    ],
    "injektions": [
        {
            "ijk": "pumba-command-injektor",
            "type": "pumba-ntem-delay",
            "totalRuns": 1,
            "workLoad": {
                "image": {
                    "user": "neves",
                    "name": "get-hotrod-fe",
                    "tag": "latest"
                },
                "dockerProps": [
                    "--add-host=host.docker.internal:host-gateway"
                ],
                "cmd": "",
                "replicas": 1,
                "slaves": 1,
                "duration": 60,
                "env": []
            },
```

```
            "dataCollector": {
                "name": "jaeger",
                "parameters":
                    {
                        "host":
                            ↪ "http://192.168.1.85:16686/api/traces"
                    }

            }
        }
    ]
}
```

Instituto Superior de Engenharia
Politécnico de Coimbra