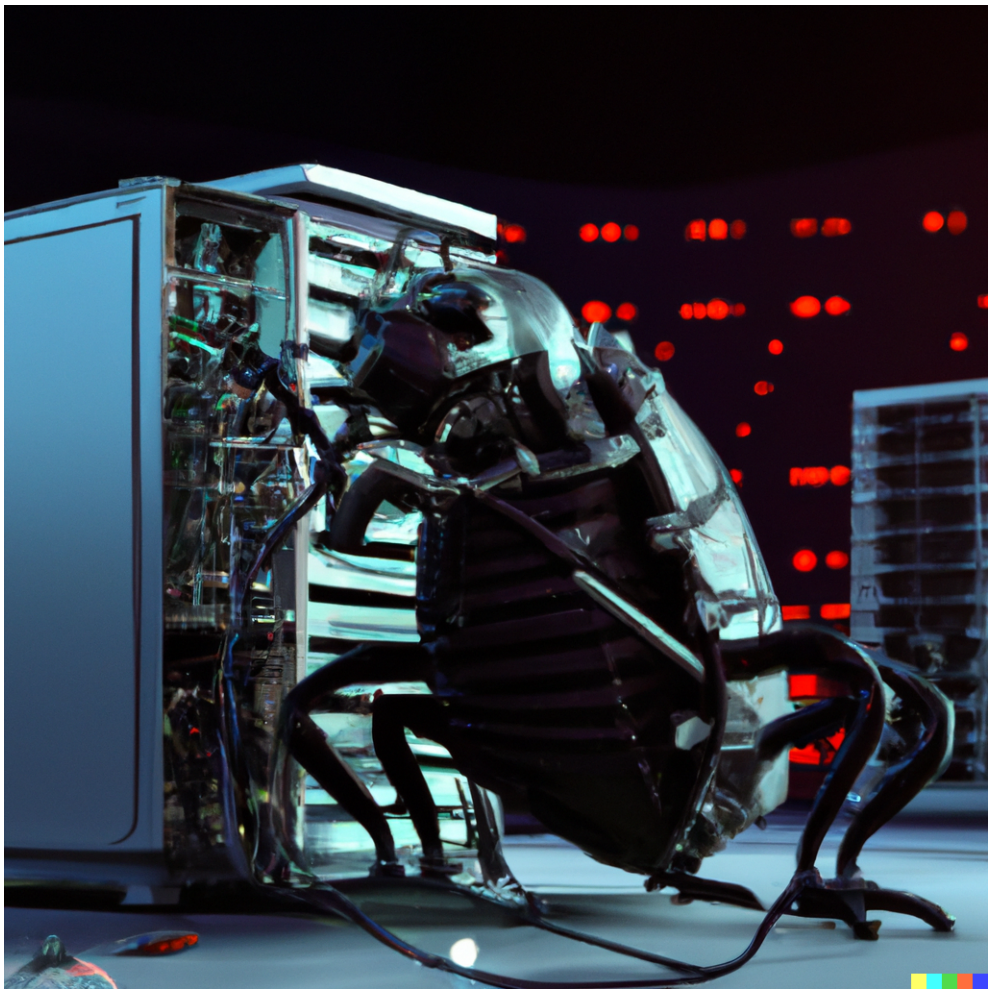
Bug Detection in Distributed Systems with Platform-independent Fault Injection: A Case Study at Adyen

Version of June 12, 2023



Nick Dekker

Bug Detection in Distributed Systems with Platform-independent Fault Injection: A Case Study at Adyen

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Nick Dekker
born in Roelofarendsveen, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Adyen
Simon Carmiggeltstraat 6-50
Amsterdam, the Netherlands
www.adyen.com

Bug Detection in Distributed Systems with Platform-independent Fault Injection: A Case Study at Adyen

Author: Nick Dekker
Student id: 4965388

Abstract

Fault injection has been a long-standing technique for testing software. Injecting faults into a system, either in production or development environments, offers unique opportunities to discover bugs that are difficult to reproduce using conventional testing methods. However, it is widely considered to have a high implementation threshold. Due to this threshold and out of skepticism of its effectiveness, many developers are resistant to the idea of injecting faults as a testing method. This thesis introduces "Yet Another Fault Injector": YAFI, a platform-independent fault injection framework designed for distributed systems. Our hope is that this framework is adopted to research future fault injection and causes the bar to apply fault injection on previously hard-to-test systems to be lowered. We perform a case study to evaluate YAFI and find that with minimal implementation of fault injectors and little developer input, bugs and flaws can be detected in a system by running fault injection experiments. A case study, performed at Adyen, shows that the system under test (SUT) is resilient in certain scenarios. Automatically generated failure plans have been shown to result in system behavior without requiring in-depth knowledge of the SUT. Injected faults were reflected in the response metrics when information from the experiments was used to generate additional failure plans. This emphasizes the need for gathering proper response data and system metrics to evaluate the system's behavior under different fault conditions. Additionally, YAFI has been executed on a project based on Apache ZooKeeper, to show portability to other systems.

By introducing YAFI and showcasing its effectiveness through the case study, this thesis contributes to the advancement of fault injection techniques and encourages wider adoption of fault injection for testing distributed systems.

Thesis Committee:

Chair:	Prof. Dr. A. Zaidman, Faculty EEMCS, TU Delft
University supervisor:	Prof. Dr. B.K. Ozkan, Faculty EEMCS, TU Delft
Company supervisor:	A. Daignan, Team Lead Platform Scalability, Adyen
Committee Member:	Prof. Dr. J. Decouchant, Faculty EEMCS, TU Delft

Preface

As the final part of my education for a master's degree at the Delft University of Technology, I present my master's thesis. Over the past two years, the focus of my studies has been on distributed systems and software testing. Due to the fantastic professors and peers at the TU Delft, I have been able to learn a lot about these topics. With the opportunity to do a case study at Adyen, I was able to combine these two fields of interest.

I would like to thank my thesis supervisors for their guidance. Antoine Daignan, for the effort put into making my time at Adyen conducive to my academic purposes. Burcu Kulahcioglu Ozkan, for the many creative academic insights and helpful recommendations throughout the thesis. And Andy Zaidman, for the thorough feedback and the enthusiasm about my topic.

I also want to express thanks to friends and family. First, my brother, who despite his hectic schedule, has made time to deal with my many obnoxious requests of brainstorms, sanity checks and academic ideas. Thanks to my father, for being open to assess my development heavy work, even though he already had his time occupied with copious amounts of developer issues, as an software engineer himself. I am grateful for my mother, for her support and attempts to understand my work. Thanks to my friends, for studying with me, being a listening ear, serving as a 'rubber ducky', and providing good support throughout the writing process (and beyond). A special thanks to Ariena and Cosmin, for spending **hours** reading my thesis and providing me with valuable feedback. Finally, thanks to all the amazing people at Adyen for being welcoming, helpful and inspiring during my internship.

Nick Dekker
Roelofarendsveen, the Netherlands
June 12, 2023

Contents

List of Figures

Chapter 1

Introduction

Since the early days of programming, people have struggled to make their applications reliable. These struggles stem from failures which can range from decreased performance to incorrect output, and even data loss [7]. Many attempts have been made to make applications as resilient to faults as possible [22]. Software testing includes various techniques with the goal of detecting as many bugs in the application as possible. Well-known techniques to ensure that applications remain reliable even in edge cases include unit tests, end-to-end tests, and similar methods. Despite the efforts that go into this goal of resilience, the economic cost of software bugs is significant [60]. Even big tech companies still suffer outages, for example, recent outages at Zoom and Microsoft were caused by traffic redirection and configuration problems [38]. Similarly, Google has had problems with configurations [3, 51], causing outages. These failure scenarios are difficult to emulate with testing techniques from the test pyramid [21].

1.1 Problem Statement

Distributed applications are complex and often need to be highly available. Testing these systems can be hard, especially in rare edge cases, where resilience is desired. When distributed applications are deployed as part of the core model of a business they often have to operate 24/7. This availability requirement imposes a level of fault tolerance to handle failures such as network outages, hardware component breakdowns, and other unexpected interferences. However, these scenarios are often not covered by standard test suites. Traditional testing methods, such as unit tests, evaluate the application flow and execution of code. These methods are subject to several limitations [39], including the difficulty in simulating external disturbances. These methods are also limited in efficiency by the amount of time that goes into developing them [33]. A method in testing called fault injection deals with these cases by simulating real-world problems in a system to see how it behaves. This method of testing has shown to detect previously unknown bugs in mature software such as Apache's Cassandra [1], Zookeeper [2] and others [48].

1.2 Case Study Context

This thesis aims to establish and execute a fault injection framework for a distributed cloud system at Adyen. With the injection of faults into the system, we can simulate realistic scenarios, test the recovery capability of the system, and identify potential weaknesses that may undermine its performance and reliability.

Adyen, one of Europe's leading payment solution providers, is the case study in focus here. The case study examines a distributed cloud system at Adyen, which facilitates in-store payments. This SUT is called the Nexo Router and handles real-time payments for many merchants, it is crucial for their systems to be highly available. Any disruptions or downtime in the payment processing system can have significant consequences for both Adyen and the merchants they serve, leading to financial losses, customer dissatisfaction, and potential damage to their reputation. As a result, continuous efforts are made to improve the resilience and reliability of the Nexo Router, ensuring that it can withstand various types of faults and maintain seamless performance in processing in-store payment transactions. The Nexo Router was suggested as a testing target by Adyen's engineers.

1.3 Terminology and Document Formatting

In the field of fault injection, several terms have been overloaded and have ambiguous meanings. We define the terminology in Appendix A and use the terms consistently throughout the text to avoid confusion. The glossary also contains concepts and abbreviations that are frequently used in the text and have contextual meanings.

Throughout the text, we also use consistent formatting. With important terminology in *italics* and code and data-structures formatted in a `monospace font` or listings. Mathematical formulas or variables are represented in $math$ mode.

1.4 Research Questions

In this research, our aim is twofold: to construct a platform-independent fault injection framework and to evaluate its efficiency. Our evaluation will primarily focus on the framework's ability to identify bugs in a distributed system, using a specific implementation of the designed framework.

We explore the design challenges of a generic fault injection framework and its effectiveness in detecting bugs. Having established the context and objectives of our research, we formulate two specific research questions to guide our investigation, both divided into subquestions.

RQ1: How can a platform-independent fault injection framework be designed to effectively test the resilience of distributed applications?

RQ1.1: What types of faults can cause failures in the executions of distributed applications?

RQ1.2: How can fault injection techniques be utilized to expose vulnerabilities in a system?

RQ1.3: What are the potential challenges in designing a platform-independent fault injection framework?

Recent works have targeted orchestration frameworks and have not been designed to be portable. With the lack of research on portable fault injectors and uniform architectures, existing research and architectures need to be analyzed to find information on this topic. We answer RQ1 with the help of a literature study, specifically RQ1.1 and RQ1.2 are answered by analysis of fault injection systems on distributed system.

In this thesis, we present a framework and an implementation capable of injecting faults into an application, as illustrated in a case study context. The third subquestion, RQ1.3, is answered using the results of RQ1.1 and RQ1.2, combined with information obtained from the case study. Evaluating the effectiveness of this framework prompts another essential question:

RQ2: How effective is the fault injection framework in detecting bugs?

RQ2.1 How can system metrics be used to validate system output for detecting unknown bugs?

RQ2.2 Is the system capable of detecting known bugs using targeted fault injection?

The effectiveness of our framework implementation is evaluated by looking at experimental results. The purpose of our framework is to identify bugs when they are present in the SUT. We do this by creating two experimental setups: one where the SUT is not guided toward a bug and another where the SUT is guided toward a bug. The experiments show how effective the framework is in each case.

1.5 Contributions

The following contributions are made through this thesis:

1. Design of a framework for platform-independent fault injection, presented in Chapter 4.
2. Implementation of a proof-of-concept application that adheres to the higher-level framework design, presented in Section 4.4 and Chapter 5.
3. Evaluation of the framework implemented, through a case study, in Chapter 5.

1.6 Thesis Structure

The structure of this document is as follows. First, in Chapter 2, we go over the related work. Background information is presented, and we classify faults and injection of faults

in distributed systems. Next, Chapter 3 explains methodology that was followed to evaluate our novel framework. Chapter 4 then presents this framework and highlights the challenges we faced as part of the design process. The framework is used on a case study application in Chapter 5. We perform this evaluation in a company context, using a distributed system as the target. Finally, we finish up with future work, a discussion and conclusions with our findings in Chapter 6.

Chapter 2

Related Work

The objective of this work is to establish a platform-independent framework capable of injecting faults and detecting bugs. Our focus is on fault injection, a technique used to test the correctness and reliability of systems. It is a process where faults and errors are deliberately introduced in an application to assess its behavior and ensure it handles such problems properly. In this chapter, a comprehensive summary of the related work from both literature and industry is provided. The strategies of the related work serve as fundamental building blocks or starting points for addressing our research question. Fault injection is part of software reliability engineering.

Having control over effects of faults is critical, especially when performing chaos engineering and when injecting faults into a production system. In case an injected fault causes severe outages and business critical operation is disturbed, the effect has to be quickly reversible. If this is not possible, testing in production poses serious risks. Even though these faults may impact live systems, performing controlled perturbations is still preferred over being in the dark when outages occur in a production system [69].

In this chapter, we will explore what a fault is and how faults are used in fault injection systems. Here, we specifically look at fault injection in distributed systems, which has received limited attention among published research, according to a study by Bertolino et al. from 2019 [18]. Their study categorizes 147 articles on distributed testing into three groups. They classify 63% as testing in the cloud (TiC) and 27% as testing of the cloud (ToC), while the remaining 10% is both: testing of the cloud, in the cloud (ToiC). We explore some of the most popular fault injection systems to gain an understanding of the state of research and the techniques that are used and avoided. With the help of literature we answer RQ1.1:
What types of faults can cause failures in the executions of distributed applications?
We also look at the effects of these faults, and the common techniques to inject them and answer RQ1.2:
How can fault injection techniques be utilized to expose vulnerabilities in a system?

In this chapter, we first look at classifications and concepts that are present in literature. Then, we look at how systems use fault injection techniques and common architecture ideas.

2.1 Classification and Taxonomy in Fault Injection Systems

As one of the steps in our research, we look at different types of symptoms of system failure. The reasoning behind this is to understand the underlying faults or problems that cause these system failures, which are injected by fault injection systems. Fault injectors are tools or software components that manifest faults in a system under test (SUT) when it is running. By understanding the specific faults that contribute to system failures, we can generate appropriate fault injectors. Initially, we prioritize the identification of the most crucial network faults. By solely focusing on the most important fault types, particularly those accountable for critical failures, we can construct fault injectors that efficiently explore software systems. These fault injection implementations can be tailored to the specific SUT if necessary. Acquiring knowledge of faults enables us to evaluate systems more effectively, as we simulate the problems that are most likely to occur. All of the properties of faults can be used to answer RQ1.2.

2.1.1 Definitions of Faults and Failures

Defining the distinction between faults and failures is important to understand concepts and results in fault injection. We follow standard definitions from the IEC[1] [23] to align with existing research and avoid confusion between other work. These definitions align with seminal work by Laprie et al. [47].

Avizienis et al. [12] define the relation between faults, errors, and failures. Errors can propagate between components, which makes the relationship of faults and failures circular, as a fault can be caused by a failure. The relation is shown in Figure 2.1. When a fault is introduced in a system, it may activate, causing an error state in the system. Then, one or more such errors can propagate into a failure.

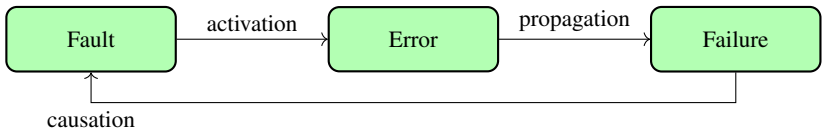


Figure 2.1: Relations of faults, errors, and failures.

Regardless of which fault occurs, they can manifest in multiple ways. Some literature only divides faults into two classes: *permanent* and *transient* faults [42, 49]. Permanent faults remain in the system as long as the responsible system part is not reset, whereas transient faults are resolved after a certain amount of time. Others go further by defining intermittent faults as a separate category [75]. *Intermittent* faults may be transient or permanent and produce failures at intervals. A fault is not guaranteed to activate in a system, if this is the case we call it dormant [58]. When a fault is activated, it does not necessarily produce a detected failure, if it does not, we call it a latent fault [56], this is not a class by itself, but a property. The counterpart of a latent fault is an activated and detected fault, called an active fault. With this property, we can sub-divide even further. Nikolaidis et al.

[1] https://iec.ch/homepage

[59] define five classes of faults. They identify *non-recoverable* and *recoverable* permanent faults as separate classes. The final class is a *transparent* fault, which is a latent, permanent fault.

2.1.2 Classification of Fault Types

Fault injection systems are built with a focus on a specific fault type. Our goal is to inject software faults into a distributed system. The most common separation of fault types is between hardware and software faults. The Internet of Things (IoT) is a field where fault injection is ubiquitous [30]. IoT devices are tested with hardware faults [16] and inject software faults that emulate hardware faults. In this work, hardware faults are out of scope, which also excludes fault injection on IoT devices from this research. In the context of our research on distributed systems, we prefer simulation-based software faults whenever possible, due to their ease of injection [11].

Hardware Faults

Hardware faults usually target low-level hardware components. There are many different techniques [28], often with the goal of causing bit flips [35] or power interruption [10, 13, 54, 66].

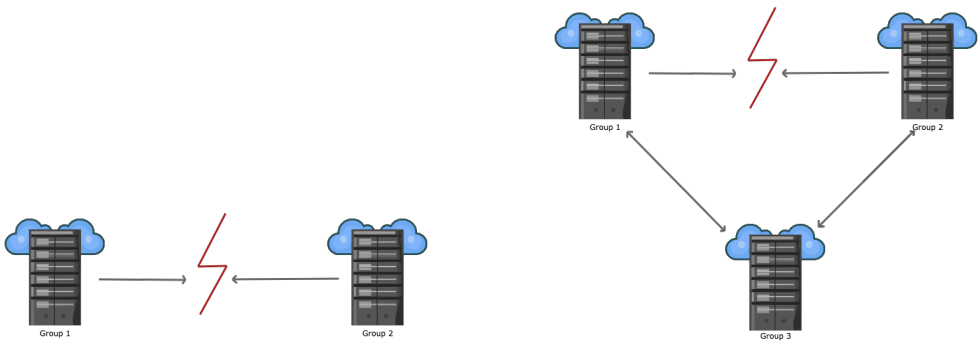
Software Faults

Software faults either emulate or simulate problems within software systems [41]. This includes the emulation of hardware faults [58, 36] and bugs found in software [53]. Simulation-based fault injection replicates scenarios where software faults occur, while emulation-based fault injection mimics hardware faults. An example of emulation can be seen in link failure in a network. A root cause of this can be a failure of a hardware device. The effect could also be caused by an incorrect firewall rule, which we can simulate. Simulation-based faults mimic the effects of common software occurrences that cause systems to misbehave. An example is CPU utilization, which can throttle parts of an application and disturb normal operation.

We look at two areas of faults that have large impact on distributed systems: Network faults and Exception faults. Distributed systems are complex systems, with these types of faults being common causes for failure [7, 73].

Network Faults

In distributed systems, a large part of the logic is reliant on messages sent over the network. Recent tools such as NEAT [7] and CoFI [19] have shown that network faults can cause critical failures in distributed systems. A set of network faults can be seen in Table 2.1. Network partitions (NPs) are instances of a link failure fault, where a group of nodes is segmented from another. In complete NPs, nodes in a group are not able to reach nodes in



(a) Complete network partitioning, group 1 and 2 cannot communicate.

(b) Partial network partitioning, group 1 and 2 cannot communicate directly. Group 3 is able to communicate with both.

Figure 2.2: Network partitioning examples.

another group, see figure Figure 2.2a. In partial NPs, this is still possible via another route, see Figure 2.2b.

Exception Faults

When unexpected events occur that cannot be handled by a part of the software, exceptions are thrown. One study finds that incorrect error handling is the cause of 35% of catastrophic failures of their samples. A recent study introduces ChaosMachine [73], where they define classes of error handling caused by exceptions, based on try-catch block resilience.

2.1.3 Classification of Fault Injection Systems

The separation of hardware and software faults is also represented in the types of fault injection systems. A system that injects hardware faults is called a Hardware-Implemented Fault Injection (HWIFI) system. Unsurprisingly, the software injection counterpart is known as Software-Implemented Fault Injection (SWIFI).

Recently, a variety of injection methods and platforms have been designed. Inside the context of SWIFI, there exist suites [59] and Fault Injection as a Service (FIaaS) [43, 26, 25].

All of these types of injectors have their own techniques and architectures with many shared components. NFTAPE [67] uses a technique in which the fault injection component is replaced by LightWeight Fault Injectors (LWFI). This allows the creation of a flexible architecture with replaceable fault injectors.

2.1.4 Properties of Faults

In the context of fault injection a fault has three properties that we need to keep in mind when designing a fault injector, as it can affect the choice of attack surface. These properties are often expressed as what-when-where (WWW) [58, 53]. It consists of the capability

Table 2.1: Faults and their corresponding bugs.

Fault	Consequence
link failure	data loss, stale read, split-brain
packet loss	data loss, reappearance of deleted data
high latency link	stale read, system crash, split-brain
partial NP	system crash, performance degradation
complete NP	system crash, performance degradation
packet corruption	data unavailability, data corruption
packet reordering	reappearance of deleted data
duplicate message	dirty read, system crash, data corruption

of injecting a specified fault type (what), at a certain point of the execution (when), and a specific point in code (where). This combination is an important idea in injecting faults, and is used under different terminology with the same intention across literature and implementations. It also is strongly related to three common classifications of faults, transient, intermittent, and permanent faults [75], since those can be emulated if WWW is properly implemented by starting and stopping faults accordingly.

2.2 Overview of Fault Injection Systems

Fault injection methods, which have been in existence since before 2000, have continued to evolve up to the present day. Hsueh et al. gave an overview of fault injection techniques as early as 1997 [40]. Fault injection can be seen as part of fault removal, a process to identify faults with confirmation and proof and "removing" them from the application [45, 6, 52]. Injecting faults can be a way of proving a particular fault results in a bug in an application. Fault removal falls in the dependability [29] classification according to the taxonomy of Alazawi & Al-Salam [6]. Closely related to fault injection is chaos engineering, see the work by Basiri et al. [15, 62]. Chaos engineering is known as the act of intentionally introducing faults specifically in the production environment, using fault injection [74]. It reduces the probability of diverging the environment configurations between live and test. The cost of setting up a test environment, equal to the scale of production is also mitigated. This section provides an overview of the most relevant state-of-the-art fault injection implementations. As part of the field of software testing and software reliability engineering, fault injection has been investigated to great depth [75, 22, 46, 58, 18, 49].

The rest of this chapter highlights the most relevant work of the past decade and identifies their novel techniques of bug classification and detection. First, a set of fault injection frameworks and models is listed, highlighting their main contribution and the way they operate. Chaos Engineering tools provide us with similar information about the systems they

test and how to design fault injection systems for them. As they are often directed at production systems, the scope of their attacks is more limited than that of general fault injection systems.

2.2.1 Injection Targets and Search Strategy

Two important common features of fault injection systems are injection target and search strategy. We identify a common target of fault injection systems: (container) orchestration systems. Orchestration systems provide a uniform layer on top of the operating system (OS) for any application running in this orchestration environment. The benefit of having this layer between the OS and the application is consistency in fault injection, regardless of the application. By implementing the injection features with support for this layer, one can build very general fault injection systems that inject many types of faults, without worrying about compatibility with the tested application built on top of it. Unfortunately, each container orchestration platform is only one of the cases that might be encountered. There are many different platforms for which an abstraction layer may have to be built. Building for popular platforms will maximize the coverage of applications, though application support will remain limited. Aside from applications running on such platforms, there are applications running on bare metal, which cannot support injection in this way.

Some fault injection systems only allow for manual definition of static failure plans [9, 4]. The approach of how the system runs tests, using generated scenarios or running manual scenarios, depends on the user's case. We define the following automated exploration techniques, that work by failure plan generation:

Unguided Random Test Generation Arbitrary faults or fault configurations are executed in a system during a fault injection campaign. The output of each experiment is not taken into account in following experiments.

Guided Random Test Generation Failure plans are procedurally generated, guided by heuristics. Output of each experiment may affect future executions.

Systematic Search Test Generation Failure plans are generated by exploring fault combinations or system states in a systematic way. Commonly used search algorithms include depth-first search and breadth-first search.

2.2.2 Established Concepts and Definitions

The design of fault injection systems, with the goal of presenting their novel ideas, brings new designs and architectures. Unfortunately, this also means they introduce new terminology for the approach, even when this is not required. SWIFI tools, such as those in the next section (Section 2.2.3), follow common practices and concepts. In these tools, we observe these commonalities, but we see inconsistent naming. Using well-defined concepts from literature provides benefits for future research. We define a common concepts concepts on a high level.

Fault injection tools run tests on an SUT in an *experiment*. The schema for the experiments are typically specified in a *fault model*, which contains all properties required to inject

the fault, defined in Section 2.1.4. The idea of fault models is not unique to fault injection, representing a logical model representation of a physical fault [37]. A fault model should be transferable between systems. In literature, instantiations of fault models can be templates [59], failure plans [19], or can be drawn from a fault library [26]. Experiments have to be performed during execution of the SUT, requiring a workload to be run. We call the combination of the planned workload and injectable faults together a *faultload*. According to its first definition, a faultload consists of a set of faults and their properties, combined with the workload [27]. In a comprehensive 2016 fault injection survey [58], the faultload is considered separate from the workload. However, we believe that this is a misclassification. The idea of a faultload is to have a repeatable configuration of a system and its execution environment for the fault injection process. For this to be consistent and reusable, we need a workload configuration and a set of faults. The execution of a set of failure plans is called a *fault injection campaign*. In a single campaign, multiple *experiments* are run in which the faults of a failure plan are injected into the *target system* (at the attack surface and the attack points). Before running a fault injection campaign, the system can run a workload without injecting faults. This is called a *fault-free run*, *golden run* or *reference run* [39, 17].

2.2.3 List of Fault Injection Systems

Here, we provide an overview of recently published fault injectors and comparable industry tools to compare the features they offer. In addition, we can take inspiration from their core ideas. There is considerable is a benchmarking suite for fault injection for overlap between the features of the solutions in this list as defined. Overlap is present in components, architecture, search strategy and injection target. This raises the question of whether it is necessary to build a new application for their respective features and research. As noted by Stott et al. in 2000: “each is to some extent specific to a particular, albeit novel and useful, fault injection method” [67]. We observe a continuation of this trend, using orchestration platforms as targets and including a unique method for detection of faults. What follows is a non-exhaustive list of fault injection systems, explaining their most important concepts.

Frisbee [59] is a benchmarking suite for fault injection for Kubernetes. It provides control over Kubernetes clusters and hosts and aims to be application-agnostic. Frisbee focuses on testing recovery behavior, showing visual output for users to assess application recovery times.

CoFI [19] is a fault injector that extracts invariants using a correct run, i.e., a run without any faults injected. It then injects faults and awaits an inconsistency. When an inconsistency is found, it isolates the nodes through network partitions, to inspect whether correct operation is resumed. CoFI can replay network partition scenarios and runs a test case for a single invariant. It loops over the Failure Plan, which generates new scenarios until it has covered all message types of the application it is testing. This does take a long time and they report a run-time of 585 hours to run 13.016 tests.

FaultSee [9] is a fault injector that injects faults according to its fault specification at pre-specified times in the process. The workload of an application is also part of the fault specification, which is defined in a configuration file. FaultSee uses a master controller on

the server running the experiments, and each server node running as a part of the SUT has a local controller with an orchestrator, responsible for interacting with Docker containers running locally. It extracts logs from the system and sends statistics back to the master controller. FaultSee runs using Docker containers, even when running multiple hosts.

Jepsen [44] focuses mainly on databases and consensus systems. While running random operations on the target system, a nemesis (fault injector) is used to perturb the system. The operation history is recorded during this process, to be analyzed by checkers. Checkers are Jepsen's way of analyzing this recorded history for correctness; they have to be written by developers and are highly specific to the tested system.

Fallout [43] is a fault injector that consists of two sections: ensembles and workloads. Ensembles represent node groups (cluster of nodes), and a node group can have a specific role. Per node group, there is one provisioner, providing faultload information to each node. Workloads consists of three stages: modules, checkers, and artifact checkers. Modules can generate faultloads, i.e., benchmarks and faults (using **Chaos Mesh**), **Jepsen** checkers can check for failure, and artifact checkers can validate if the output is correct. Performance reports using benchmarks are useful for comparison between versions.

ThorFI [26] is a fault injector that allows for minimal client effort to inject faults. The architecture is set up to have agents running on the target systems. This way, the injection tool is not intrusive and can be run regardless of the SUT as long as the environment is supported, which is now limited to Kubernetes and OpenStack. ThorFI is a recent implementation that takes into account a lot of previous frameworks.

FiliBuster [55] is a fault injector that targets service-level faults. The injector instruments network calls to discover the structure of the system and provide information to the fault injector. By creating additional tests when new service relations are discovered, the guided random testing method explores the system in a logical way. The random tests are combined with functional tests that will throw exceptions when an undesired response is returned. These events are also stored for reproduction in a counterexample file. The test generations can exponentially increase with complicated service structures, which is why (dynamic) test reduction is used.

NEAT [7] is a network partitioning framework that can be used to identify issues in distributed applications. The framework lets developers create a global order for nodes, create and heal partitions, and even cause node crashes. NEAT is useful for testing distributed systems for their resilience against network partitioning, but it does not provide any monitoring capabilities.

Chaos Mesh is a GUI based chaos engineering tool, where failure plans can be easily specified through the interface. They define Custom Research Definitions as plans for their experiments. Chaos Mesh is built to inject faults into Kubernetes. They allow injection types of network, disk, file system, operating system, and more. It shows events that occurred during the experiment in to the user through the interface.

We put emphasis on network faults, due to their importance in distributed system. Most of the fault injection systems, in the list above have network injection capabilities. Table 2.2 illustrates which of the tools are capable of injecting the set of network faults.

Table 2.2: Comparison of fault injection capabilities.

Name	Link Failure	Packet Loss	High Latency Link	Packet Corruption	Packet Reordering	Duplicate Packet
Jepsen	✓	✓	✗	✓	✗	✗
NEAT	✓	✗	✗	✗	✗	✗
FaultSee	✓	✗	✗	✗	✗	✗
ProFIPy	✓	✗	✓	✗	✗	✓
CoFI	✓	✗	✗	✗	✗	✗
Fallout	✓	✓	✓	✓	✓	✗
Filibuster	✓	✗	✗	✗	✗	✗
Zermia	✓	✓	✗	✓	✗	✓
ThorFI	✓	✓	✓	✓	✓	✗
Chaos Mesh	✓	✓	✓	✓	✓	✗
Chaos Monkey	✓	✗	✗	✗	✗	✗
Chaos Toolkit	✓	✓	✓	✓	✓	✓
Gremlin	✓	✓	✓	✓	✓	✗
Litmus	✓	✓	✓	✓	✓	✗

2.3 Commonalities of Fault Injection Systems

Fault injection systems often use various features and components which have similar behavior and roles in the system. While there are slight differences in how these features are implemented, it can be argued that they are classifiable as generic classes. First, an overview of a well-established model is given, followed by examples of how literature and industry implementations have designed their architecture.

2.3.1 The Fault Injection Environment

We identify common stages and procedures in fault injection tools and frameworks to see what role they play. For the definition of components and terminology, we mainly follow the conceptual model defined by Hsueh et al. and other seminal work. They provide a conceptual model for fault injection systems. Figure 2.3 shows the hierarchy of the components in the model. The model consists of eight components, which have their own responsibilities [75]. These exact components are still used to classify systems [65]. With this in mind, we describe the components of a fault injection system and their responsibility.

The component that handles the execution of a representative workload of the system is called a *workload generator*. Workloads are stored in the *workload library*. Workloads are executed while a *fault injector* component injects faults, coming from the *fault library*. The fault injector is traditionally responsible for creating the faults. A fault injection campaign is managed by a *controller* or *orchestrator* (not to be confused with orchestration systems), acting as a command center. The system has to gather information during experiments and campaigns. Again, this is done by a separate component, called a *monitor*. The monitor collects measurements, metrics and other data during and after the execution of a failure

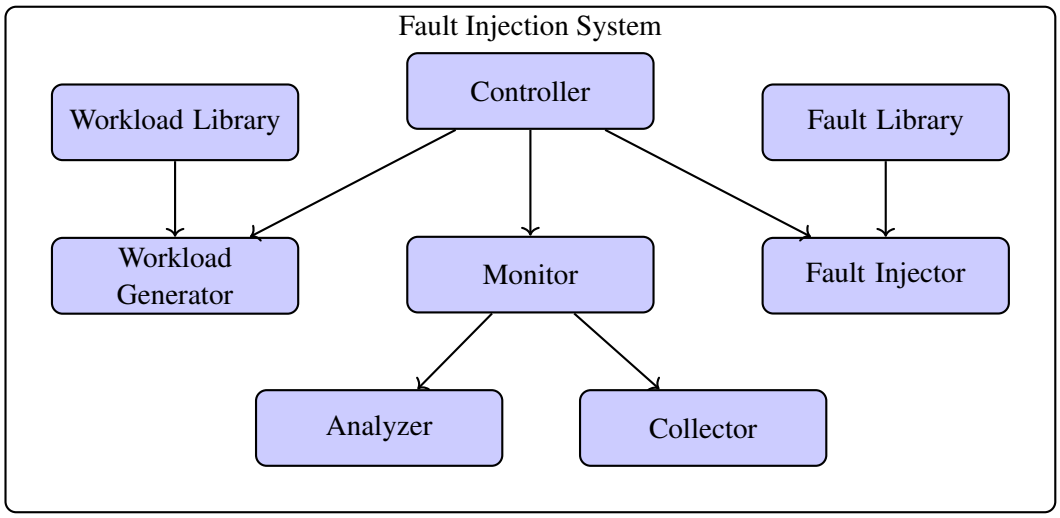


Figure 2.3: The hierarchy of the components of a typical fault injection system.

plan. The monitor makes use of the *collector* component for online data collection [75]. To process and analyze this data an *analyzer* component is used.

2.3.2 The Relation between Theory and Practice

We now relate the features and key ideas of fault injection systems to these concepts. Our comparison is limited to the most important components, including only the fault injector, fault library, and the analyzer.

Fault injectors have a large responsibility in the conceptual model of Hsueh et al. They sample faults from a fault library, which is a set of stored fault types and options. ThorFI [26] has such a fault library[2] as a method of presenting fault injection capability. These fault libraries can also be integrated in code, as implementing a certain fault injection method automatically adds them to the fault library. As soon as these faults are supported, they can be used as part of a fault model. Alternative methods for executing fault models have been emerging, especially in works that explore novel methods of systematic search [19, 55, 8]. As an example of these alternatives, Phoebe [74] uses a "synthesizer" as a scenario generator. Like others, they introduce a separate component, responsible for generating fault models and represent it as a failure plan. NFTAPE [67] decouple the fault model from their LWFIs. Many systems, including NFTAPE, support the generation of faults as a manual procedure and provide a GUI interface to improve ease of use. Other examples are ChaosMesh [4], Fallout [43] and GOOFI [5].

As part of analysis, different approaches are employed. Some frameworks rely on presenting visual output to users for evaluation purposes [9, 26], whereas others employ checkers and validation functions to assess the validity of the output [25, 68, 19]. A combination of these techniques is common [24, 59, 44, 43], because simple checks can often detect

[2] https://github.com/dessertlab/thorfi/blob/main/thorfi/ThorFI_fault_library.json

critical failures [72, 71]. Checkers consist of rules to analyze datasets and vary in implementation, Jepsen [44] for example, requires a boolean output[3]. All these techniques fit the definition of the analyzer component.

2.4 Conclusion

Fault injection systems have a wide range of faults that they can create in a target. In this chapter we have seen properties and type categories that give us answers to RQ1.1. We find that faults are defined by three properties: WWW, that make it possible to accurately represent the simulated faults. The persistence of faults we have seen also affect the effect on a system. With faults sometimes activating, but remaining latent and permanent faults and temporal faults.

The WWW properties are often used in failure plans to manage injection of fault during execution. However, this alone is not enough for effective fault injection. As part of RQ1.2, we have seen techniques concerning network faults and exception faults. Simple testing has been shown to expose many types of failures. The network failures can expose faults by creating faults in Table 2.1, as been done by NEAT [7] and others. They show there are only few nodes necessary to find critical failures. ChaosMachine [73], creates exceptions and observes the behavior of the SUT, showing the resilience of the tested system. Furthermore, exploration of SUTs is done using different search techniques: unguided random testing, guided random testing and systematic search test generation.

[3] `https://github.com/jepsen-io/jepsen/blob/main/doc/tutorial/04-checker.md`

Chapter 3

Methodology

The purpose of this chapter is to provide a detailed description of the methodology used to create a framework for the fault injection system. This consists of how we generate faults, create injection plans (experiments), and analyze the results of these experiments.

First, we look at research design, why we chose techniques, and what the consequences are. Then, we look at how our framework design is established. We also look at the details of the case study we performed and which choices were made. Finally, we look at our evaluation method and the scope of the research.

3.1 Research Design

This section highlights the design choices of the research. Throughout the design and evaluation of the framework, we follow the process in Figure 3.1. The system is built incrementally starting with a basis in research. After execution of experiments, we analyze the data, to see if the results are representative of reality and reproducible.

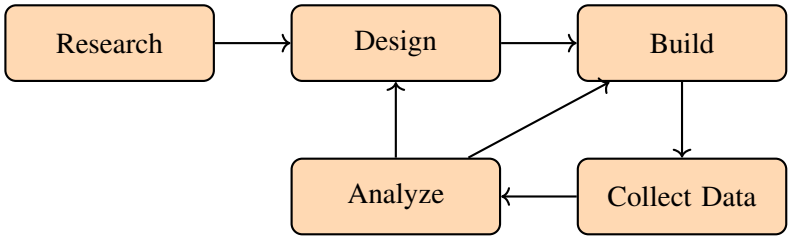


Figure 3.1: Phases of the methodology.

Our research follows (implicit) positivism [64]. We observe the reality as objectively as possible, extracting information from the tested module, and deciding the validity of the outcome. This is possible, as the desired outcomes are known.

The strategy of this research is experimental; we take an existing module and subject it to new testing methods. This is a strategy performed by many previous works, although the preference goes to testing the implementation on multiple target applications. We collect data from a single version of a single module due to time constraints and a lack of clearly

defined relevant bugs in the issue tracker. The module was chosen upon through discussion with engineers at Adyen. The Nexo Router module was chosen because it is a distributed system dependent on messages over the network to synchronize its state. Additionally, we selected an open-source application to run the framework on, showing the generalizability and portability of the system.

In this study, we examine two parts of the engineering process. First, the design of a framework and second, the evaluation of this framework as an implementation on a single system. In the case study at Adyen, we perform quantitative studies, where the output of the experimental data is used to determine if a bug is present and detectable. When executing a fault injection campaign, multiple experiments are run. The percentage of true positives and false negatives can be used to determine the detection rate of the fault injection strategy. As the case study at Adyen is an evaluation of the framework design, our goal of finding the effectiveness is a mixed method. The qualitative aspect comes from the previously stated bug detection rate. The quantitative aspect is dependent on judgment, we have few comparative baselines. There currently is no possibility to run multiple fault injection systems on the Nexo Router to compare the framework. The existing test suite is usable to compare against because there is overlap in the coverage of the framework and the system. The Nexo Router test suite consists of unit tests, integration tests, and end-to-end tests.

3.2 Planning of Design and Implementation

As the creation of an effective fault injection system relies on a good architecture, this is an important aspect of the first stages of creating the framework. Although components, gathered from literature, are a large part of its design, other specifications are required. Most of the architectural concepts are copied from previous works and combined with some of the requirements that are needed to fulfill the framework goals. One of the most important parts is its ability to be language agnostic and usable for multiple systems. After establishing an architecture, we can build an implementation to test the feasibility. Building a minimum viable product (MVP) prevents us from creating a design that is not practically feasible or unsuitable for the real-world conditions in which it will need to operate. There were several stages requiring an MVP, which could be considered deliverables in terms of project planning, shown in Table 3.1.

Table 3.1: System Parts and Deliverables

System Part	Deliverable
Failure Plan Definition	Failure Plan Generation Engine
Failure Plan Execution	Plan Runner Engine
Injector	Fault Injection Module
Failure Plan Analysis	Logging and Metrics Collector

In addition to providing these deliverables, we aim for platform-independence. This is only possible when each component is flexible, thus, we carefully construct interaction with

other components. This interaction is specified as protocols for each component, allowing for implementations by third parties and a portable design.

3.3 Design of Experiments in the Case Study

During the case study, we run experiments using our framework implementation to test the validity of outcomes under various circumstances. These outcomes provide information about the effectiveness of the implementation and indirectly about the effectiveness of the framework. There are several choices in deciding which experiments to run. In fault injection, an experiment is a single failure plan execution, executed during a fault injection campaign. Our methodology tests the Nexo Router module with two types of campaigns. In each type, our failure plan generation engine is responsible for creating a set of application-specific failure plans. Each of these failure plans defines a set of faults that are injected into the SUT. Determining which faults are injected at which time is a challenging task, due to the amount of configurations that are possible. For this reason, we limit our options to a set of faults, determined through discussions with Nexo Router developers and field experts at Adyen.

3.4 Evaluation of Results

Our evaluation method is threefold, combining the results of each part to find the effectiveness of our framework. First, we use experiment data from single campaigns to see how well our implementations perform in different contexts, albeit on a single system, this informs RQ2. Second, a form of comparison between our system and the existing test suite is possible. Third, we can demonstrate the effectiveness of the framework indirectly, by showing that it can successfully integrate new components, adopt techniques from other works, and facilitate running many new experiments.

Combining all of these results will answer RQ1.

3.5 Scope and Limitations

In this thesis, we explore an area of fault injection, limited to software faults. Beside injecting faults, a fault injection system can explore an SUT and determine the validity of experiment outcomes. Both the fields of exploring a SUT and determining the validity of outcomes are large fields of research. We limit our research to relatively small sets of these fields. As mentioned above, faults were selected to inject on the SUT. The failure plan generation part of the framework is given low priority. The campaigns we run build a set of plans according to a simple rule-set, each specified for a particular fault. This research does not compare the effect of using different search strategies. Another part of a fault injection system that we will only crudely implement is monitoring and analysis. Both parts will be implemented using simple methods present in literature.

Chapter 4

Framework Design and Implementation

In this chapter, we present Yet Another Fault Injector: YAFI, a framework based on protocols and a predefined architecture, designed to be platform independent. Our framework can be implemented to enable fault injection across different target systems. Through exploration of the literature, we built a solution that is modular and maintains support for several major features that are present in other works. While designing the framework we provide information about the choices and answer RQ1.3:
What are the potential challenges in designing a platform-independent fault injection framework? The results of RQ1.1 and RQ1.2 from Chapter 2 are taken into account while identifying challenges and designing the components and their interaction.

The framework is built with generalizability in mind. Being able to apply the concept in different environments and on different systems is facilitated by two core elements. The two elements are:

Framework Core Commonly, frameworks offer a base for developers to work from, removing the need for them to (re)implement a reusable architecture from the ground up. This is the idea that YAFI offers; the framework does not have to be implemented separately for different environments.

Extension Implementation An extension is a pluggable component of the framework. The framework core does not rely on any implementation of these extensions. For effective fault injection, implementation of these extensions is expected.

This chapter goes into the details of the challenges and choices of the framework. We start with clarifications of some definitions in our context. Then we explore the framework design, its architecture, protocols, and extensions. After introducing all elements of the framework, we look at the role of the orchestrator in the system. Next, we show our implementation of the framework and show how it can be applied. The chapter wraps up with a discussion and conclusion of the challenges that we have identified.

4.1 Nomenclature and Basic Definitions

Architectures found in existing applications frequently share common components, although they tend to use inconsistent naming conventions, see Section 2.3.

Often, the first stage of fault injection is the definition of configurations that will be used to inject faults. Such configurations describe the faults with which we expect to affect the system during run-time, we call this description the *fault model* or *failure plan*. It contains important information about three properties of faults (WWW), described in Section 2.1.4. Here, we introduce some additional terminology for each property, to indicate the context more clearly:

What The description of what we inject will be used by the system in the form of a *fault type*. These fault types serve as representations of faults and closely resemble them. There are multiple faults that may result in the same fault type. Injecting a fault on language or OS level may result in the same fault type, for example, network calls (fault) may be blocked from either language or OS level and manifest as the same effect (fault type: message delay) for the application.

When The timing of the fault injection. This happens as a result of a condition being triggered. Therefore, we use *trigger* to refer to these conditions. In YAFI, a trigger is always an event, as defined in Section 4.3.3.

Where The location of the fault in the target system. This has two levels, first an *attack surface*, at which the fault is aimed. The attack surface is a component in the system architecture. It may be an operating system, a programming language, or an application. The second level is the *attack point*, a more specific target inside the attack surface. The attack point may consist of a resource, an object, or classes and methods.

We will now define the roles of the nodes and components in the system. The framework is deployed on the node from which it is run. We call this part of the framework the *orchestrator*. The machine on which it runs is the *test execution server*. The injection target is the *system under test (SUT)*, which runs on the the *target machine*. The injection target can also be the target machine itself.

4.2 Framework Design

This section presents the design of YAFI and the challenges that accompany designing a platform-independent fault injection system. There are many ways a fault injection framework can be constructed. The architecture of our framework consists of six components, as illustrated in Figure 4.1. These components are: failure plan generator, workload generator, fault injector, analyzer, collector, and controller.

We explain why we choose these components in Section 4.2.1. After selecting the components, we define the interaction with the target system. Then, protocols are established, allowing for reproduction of each component. Finally, the extensions are explained in detail in Section 4.2.4.

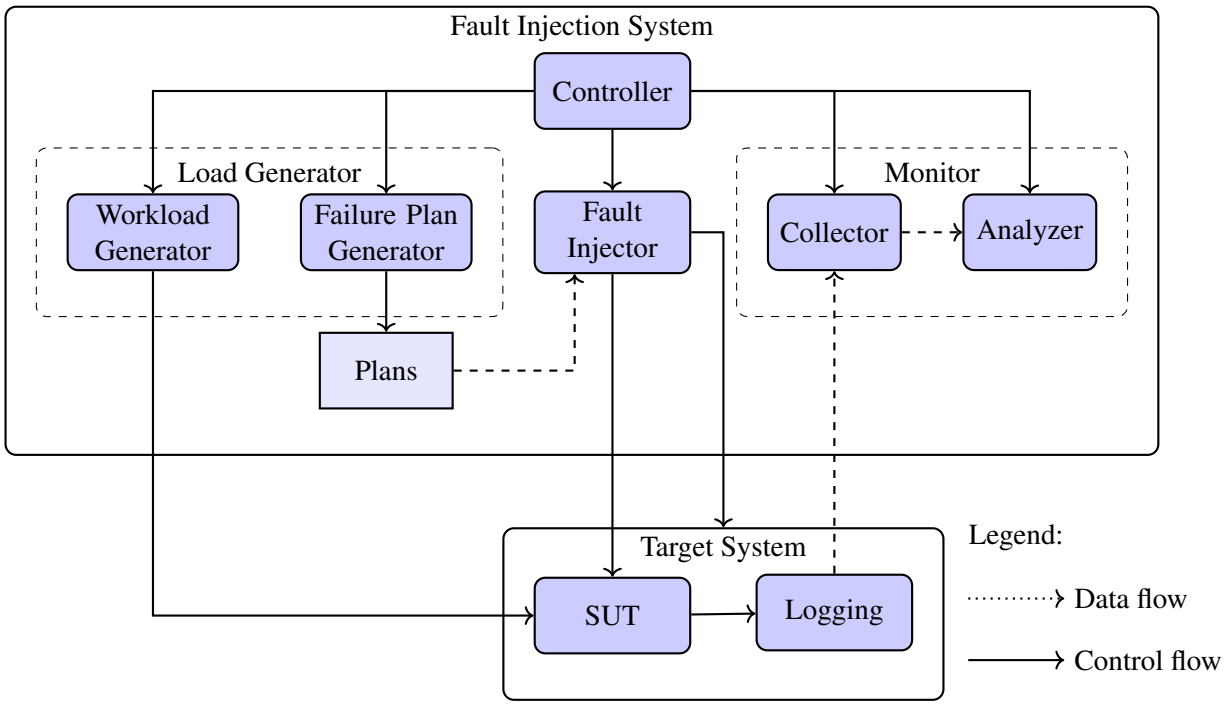


Figure 4.1: Fault injection environment, showing interaction between YAFI and the target system, and internal interactions.

4.2.1 Core Architecture Design

The framework design follows a custom architecture, created using ideas from literature. The first step to define this architecture is to select components to include and exclude. As explained by Ziade et al., a fault injection environment can consist of eight components [75], as shown in Figure 2.3. They identified components that have their own responsibility; these correspond to the components present in the work by Hsueh et al. [40]. Using the responsibilities given to each component in these works we establish the most important components. We define some challenges that clarify when choices are made. Our first challenge concerns which components are necessary for fault injection:

Challenge 1: What components are part of the core architecture?

The most important component is the *fault injector*; it is responsible for injecting faults and it requires faults to be defined in some capacity. Typically, these are stored in a fault library, one of the components defined by Hsueh et al. Our design does not include such a library. Instead, we include a *failure plan generator*, capable of creating a fault model. As explained in Section 2.3.2, faults can be sampled from a fault library by the fault injector to determine the fault model or generated using custom techniques. Our framework 'demotes' the library from a component to an idea implemented by the failure plan generator. An important aspect of injection is the execution management of the SUT. For a failure plan to be effective, it needs to be applied alongside a repeatable workload, together forming

the faultload. To allow for repeatable workloads to be consistently executed, we include this feature as a component: the *workload generator*. At this point, we have the means to build a failure plan and execute it on the SUT. This process generates data that we want to collect. We do this directly with a *collector* component. To handle the collected data, we also introduce an *analyzer* component to make a decision about execution correctness under the faultload. All these components need to be "glued" together, which is the responsibility of the *controller*. Our implementation calls this an orchestrator.

Figure 4.1 shows the interactions between the framework components and the target system in the fault injection environment. We examine the key differences between the conceptual model as described by Section 2.3.1.

Decoupling The fault library is decoupled from the fault injector, making the fault injector responsible for a single task: fault injection.

Plan-first approach Instead of having a controller that manages and initiates all states, actions, and workloads, the controller follows the failure plan.

Monitor responsibility shift The monitor has been removed from the system, as both the collector and analyzer perform "monitoring" tasks. There is no need for an explicit monitoring component since all responsibilities are fulfilled by specific components.

Some frameworks use instrumentation [24, 31] in the injection workflow, which may be seen as a preamble to the fault injector. Since our framework targets arbitrary systems, we do not directly support or look at instrumentation. All prerequisite instrumentation should be present in the SUT without interaction from YAFI.

4.2.2 Component Interaction

One of the decisions concerns the scope of injection, as we have seen frameworks that do not inject directly into the target applications. Our framework does not rely on any platform as the environment of our SUTs. Therefore, the attack surface need to be chosen and defined:

Challenge 2: How do we handle the injection scope while remaining platform-independent?

We have to define the targets of a single fault injection event. If we limit this to a single host per fault, we are unable to reproduce some faults that require a set of nodes to be affected, and not all bugs can be uncovered. To do this, we look at some communication rules between components, with a focus on interaction between the framework core and the extension implementation.

This layer defines the topological connections between components. The restrictions for these connections are defined in the next layer.

The core layer defines the three core components that are responsible for executing the main workflow. The fault injector is able to interact with the target application in several ways. To maintain the detachment between the framework and the target application, a proxy is introduced as an intermediary component. This proxy can be implemented in

different forms: as a direct link to the system under test, as a connection to the OS, or as an agent residing on another system that handles additional injection tasks.

The proxy serves as a bridge between the framework and the target, allowing communication and interaction while preserving the separation between them. By using the proxy, the framework can control and manage the injection process without directly interfacing with the target application itself. This approach enhances the modularity and flexibility of the framework. This proxy is defined as a virtual component with protocol specifications. We say it is a virtual component because the proxy itself has no direct implementation, rather, it consists of a valid set of components as defined by the injector extension behavior in Section 4.2.4.

From the failure plan, generated by the failure plan generator, a target is defined to be injected into during the experiments by reducing the possible targets to only the proxies. This is important because we have a one-to-many relation between the test execution server and the target machines. End-users (developers) of the framework are now able to extend the fault injector functionality in any way they see fit, as long as they conform with the proxy requirements defined in the next section.

4.2.3 Component Protocols and Stubs

In our framework, components may receive data, process them using their respective extension type, and pass the data to the next component. Data are passed along to the point of injection. Injection generates more data, which are passed along again until it yields a usable outcome for the end-user. Usable outcomes include a boolean value, denoting whether an execution of the application was correct, or a time-series, often visually analyzable. We formalize the connections between these components to ensure compatibility of each step with the next. The implementation of each component can vary through the use of extensions. When we do not enforce a standard to follow for the implementation, the components can be made to work together as a fixed set, but do not guarantee compatibility with other extensions. Hence, we pose the challenge:

Challenge 3: How are components made to be reusable and transferable to other systems?

Creating protocols for interaction with components is helpful, as it makes the extension implementations uniform. We call this the extension protocol and define them per component. We also define the extension protocol from the framework core to the extension implementation.

By enforcing communication and protocols in the framework, we limit choices for developers when implementing the framework. This helps the framework to have fewer variable factors and thus, helps with achieving determinism when performing fault injection. First we define some protocols and afterward, we look at how determinism is affected:

Challenge 4: Does the framework enforce deterministic execution?

The connections between the components are defined in Table 4.1. Their protocols are defined in Appendix B. The fault model protocol links several components together. The

fault generator outputs the fault model, which is then used by the fault injector to create an injectable fault configuration. Due to the proxy that was introduced to reduce the complexity of the fault injector and the framework, all elements that are implementations of the proxy also need to have input according to the fault model protocol specification.

Connection	**Protocol in**	**Protocol out**
source → **collector** → generator	collector.input	collector.output
collector → **generator** → injector	collector.output	fault_model
generator → **injector** → proxy	fault_model	proxy.input
injector → **proxy** → SUT	proxy.input	proxy.output
collector → **analyzer** → output	collector.output	any

Table 4.1: Connections between components and extensions

The input of a component is always the output of the preceding component. In this way, the orchestrator can simply run each component in succession, regardless of its implementation. This is where stubs come in, which is a very practical requirement and very easy to enforce when implementing. The orchestrator is responsible for calling the components, and in an implementation, we will do this using method calls or stubs. A stub is an unimplemented method that will serve as a point of interaction between the orchestrator and the components. Currently, we do not enforce the stubs to follow any particular format, as long as they follow the component protocols. As a next step of the architecture, this may be improved by enforcing a uniformly callable endpoint for each component, which is out of our scope at this time. The framework may evolve when modifications are made, which are not expected to permanently break component functionality. Instead, components and their stubs can be updated for compatibility with the new version.

4.2.4 Extensions and Implementations

Each extension type has its own set of requirements and behavior. This constitutes the final layer and not part of the core framework. Instead, all components that are defined here are add-ons as an extendable part of the framework. They conform to the protocols and stubs defined before to be compatible with the framework core. All components that have protocols are listed here.

Collectors and Analyzers

There are two related components that perform monitoring functionality. The goal of the collector is to collect data, present them to the user in an easily understandable way, and allow for implementation of comparators. To achieve this, we first need to gather data from the SUT, some of which is standard and usable across multiple target systems (CPU-metrics, response times) and some is unique per application. Examples of comparators range from simple validations of operations having been executed such as user creation, generation of files, and more, to more complex validations and data collection, such as whether the

amount of failed requests is congruent with the fault load's expectations. This type of evaluation is used in [73].

Data collection may be done actively, during execution of the application workload, akin to monitoring sidecars [73]. Passive collection is also possible, done after execution. Active collection is used when real-time data are required and is time sensitive, whereas passive collection can be done at any time after execution, until deletion of the data takes place. The collector shapes arbitrary user data into a generic format that the analyzer can use to determine the validity of the output of application executions. In addition to the analyzer, the generator can also follow the collector as the next component (see Section 4.2.4).

Analyzers can range from simple equality checks to complex models that compare multiple data sets. In this work, we do not focus on an automatic analysis of the results of the experiments. Manual analysis is performed as a step towards a full implementation of an automatic, platform-independent fault injection system. There are many challenges and possibilities for analysis. We look more into these challenges in Section 4.5.1 The framework supports the extension of evaluation capabilities through compatibility with a user-implemented analyzer that conforms with the specifications in Table B.3.

Failure Plan Generator

The purpose of a failure plan generator is to build a failure plan. A failure plan is built from received input originating from a collector or a direct instruction from the orchestrator. This input can differ based on collected data, therefore they need to be synchronized. To remain agnostic in the observability metrics that can be used to generate failure plans, we leave the input for generators unspecified. As a result, collectors and generators are built as pairs to collect a specific set of information and build a failure plan. Collectors and generators may be exchanged as long as they are compatible, meaning the collector output matches the generator input protocol.

Inside of the failure plan we define the targets, along with the faults and when they should occur (trigger), see Table B.1. The fault and the trigger make up a *conditional event* together, with the fault as the event and the trigger as the condition for it to take place. Conditional events are used to exclude the same fault to be used in a single failure plan (deduplication). This means that a fault may be part of a failure plan twice, as long as the trigger is not equal.

Connector

The connector is a small subcomponent, as part of the injector, that serves as a helper and intermediary between two systems. Users may choose to implement a connector that handles the communication between two nodes using any means. The only requirement is to provide information for the other side to execute. An example is SSH, this type of interaction is used in other applications [44, 7], to facilitate injection. When using SSH, it is not expected to handle the message with an agent, instead running commands that directly affect the system to which it connects.

Connectors are always required and can be a proxy responsible for the actual injection of a fault. It follows the proxy protocol, as defined in Table B.10 and Table B.11.

Injector and Proxy

In most implementations, the fault injector will require access to the file system, command line tools, and sometimes elevated access to the target system when injecting faults, similar to other fault injectors [61, 7]. When using this framework, it is not necessary to have any special access, though it stands to reason if we want to manipulate a system we need to be able to instruct the target machine. We assume that the user has the appropriate access to inject the faults that they intend to inject.

The proxy influences the implementation requirements for injectors. The orchestrator will always call the fault injector for injecting a fault. On the side of the framework, the proxy abstraction causes the injector to be very simple. Introduction of the proxy divides the fault injector into two parts that work closely together. One part is responsible for providing the faults and fault information from the failure plan and runs on the test execution node. Injectors follow the fault injector specifications from Section B.4. The counterpart is responsible for the actual injection of fault behavior, handled by the proxy. Proxies are required to handle the conversion from the specification to an injectable fault on the attack surface or the attack point being targeted. A connector can be a valid proxy as long as it is able to inject a fault with the correct configuration. Another option is to deploy an agent on the target system, capable of injecting the fault, sent by the injector orchestrator side. Therefore, the injectors must implement the same specification used by the fault generator. In the second option, the proxy is fulfilled by the connector sending a message and the fault injector on the side of the target machine. We do not limit injectors to any platform or environment.

4.3 Orchestration and Flow of Components

The orchestrator is a basic failure plan executor, it instructs the fault injector to start or stop perturbation on a node, based on the generated failure plan. However, it still has some responsibilities as a controller of the individual components. This section will define the responsibilites and workflow of the orchestrator.

4.3.1 Controller Responsibilities

A node can only be part of a single orchestrator, with the orchestrator managing a technically unlimited number of nodes. There is a one-to-many relationship between the test execution server and the target systems. The orchestrator is responsible for determining the node(s) that will execute a fault. It will also maintain the state of the executions that have occurred.

Failure plans define the targets for a fault, however, often the targets are not known at time of plan definition, and the set of targets is one of the supported options. The targeting options are defined in Table B.14.

Having multiple SUTs on a single machine can cause one system's actions to influence the other. This is also true for the faults we want to apply to the targets. Some of our faults are system-wide and cannot be applied only on a single application in this case. We leave the responsibility of configuring the infrastructure to the developer using the framework. Developers can choose to enable system-wide faults and have to be aware of the ramifications.

The orchestrator is responsible for coordinating the entire injection process by calling each component. Part of this process is running external workloads as part of the load generator. The workload needs to be coordinated with the injection of faults (constituting the faultload), which is not straightforward. The goal is to have representative faults that are repeatable and non-intrusive [27]. In benchmarking, the faultloads are also expected to be portable to other systems. We identify another challenge:

Challenge 5: To what degree does the system support the execution of external workloads?

The current version of YAFI has no component that is designed for workload generation. YAFI is not a benchmarking tool, and as such, it does not have generic targets that can be instructed with a common set of operations. Fully supporting the execution of an external workload brings significant challenges and is generally not possible in a platform-independent system. Even with options to run all workloads, this would require manual configuration from the user in the framework. With limitless options of user applications as the subject we want to test, this configuration would also have to support an unlimited set of features. Still, we can provide a set of common instructions for the user to implement that has practical value. Other systems have provided support inside the faultload configuration, combining the fault model with the workload [9]. Our solution to this challenge takes a similar approach to NEAT [7], who create an abstraction layer in the form of an API. The orchestrator obtains responsibility for using the abstraction layer and we leave implementation of the workload logic to the developer. The fact that the user is not able to specify operations in the faultload places more work on the developer. Nevertheless, a basic functional test is very easy to implement, posing a low barrier for getting started with our framework.

4.3.2 Phases and Execution Flow

The framework has been designed to support multiple nodes, which is necessary to test a distributed system. Without a scenario generator, an experiment can be manually defined to be fed to the executor. Afterward, manual analysis of the system can be done in various ways. In figure Figure 4.2 each phase that the orchestrator executes is shown, along with the output generated by a phase.

In both the "Execute Reference Run" and "Execute Perturbed Run" phases, a set of data is created, as defined by Table B.13 in Appendix B. The data sets or these phases are called D_{ref} and D_{faulty} respectively. These are the data sets generated by the collector component. Comparison of these data sets can help to decide the validity of the result of a perturbed run, which is done in the analyzer component. After the perturbed run is executed,

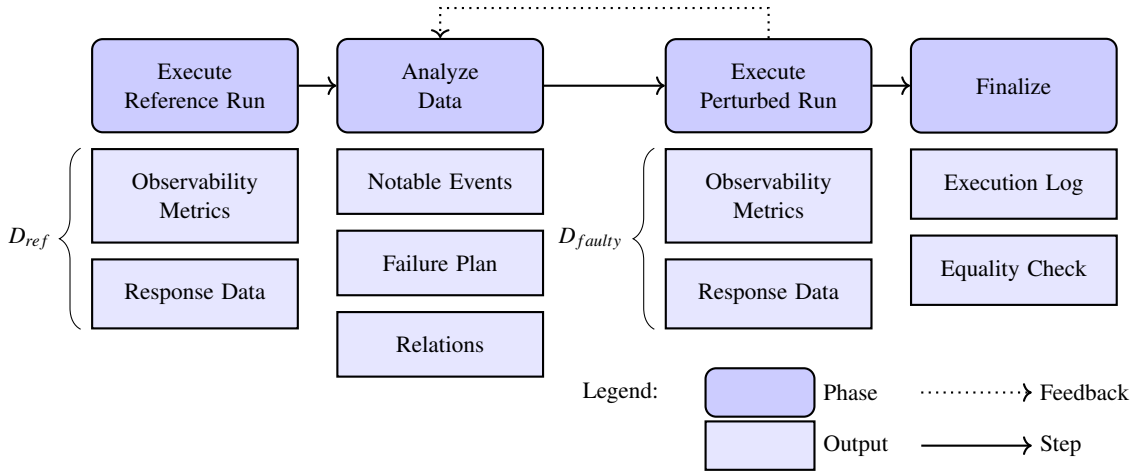


Figure 4.2: Framework phases and their output

the application may finalize or continue testing the system. In case of continuation, the feedback step is performed and the system goes back to the "Analyze Data" phase. Every time the "Analyze Data" phase is completed, new failure plans are generated. Generations are deduplicated before they are executed by the fault injection engine. If there are no generations to run after deduplication, the "Finalize" phase is entered, where final reports are built and, as the name implies, execution is finalized.

In a single execution, which happen in "Execute Reference Run" and "Execute Perturbed Run", the system goes through several steps, starting a workload, injecting faults, and performing cleanup. This execution flow can be seen in Figure 4.3.

4.3.3 Events

During the "Execute Perturbed Run" phase, the workload is executed in parallel with the fault injection engine. Challenge 5 asks about the degree to which we support the execution of a user workload. This is important for two reasons: automatic execution of a workload and synchronization of faults with the workload. The workload is always required since an SUT cannot be expected to execute without input. Then it would have to be run manually, which is a lot of effort and is unreliable. The SUT must be subjected to traffic, representing the usual system interaction, or test execution paths and output. Supporting variable workloads is a way to test application behavior in multiple cases, much like other testing methods. One feature that the architecture supports is the handling of events. This is seen in the conditional events, which consist of a trigger and a fault. A fault is only activated when a trigger condition is satisfied. Triggers always arrive in the orchestrator as events and may come from any source (typically collectors and injectors, as they contain a lot of information about the SUT). When sent as a message to the orchestrator, the expected format is as follows: `event:[event_name]:[event_value]`. An example of an event is a clock event, informing the orchestrator about the run-time of a system. The message would be `event:clock:5000`, for a trigger message containing information of 5000 ms run-time

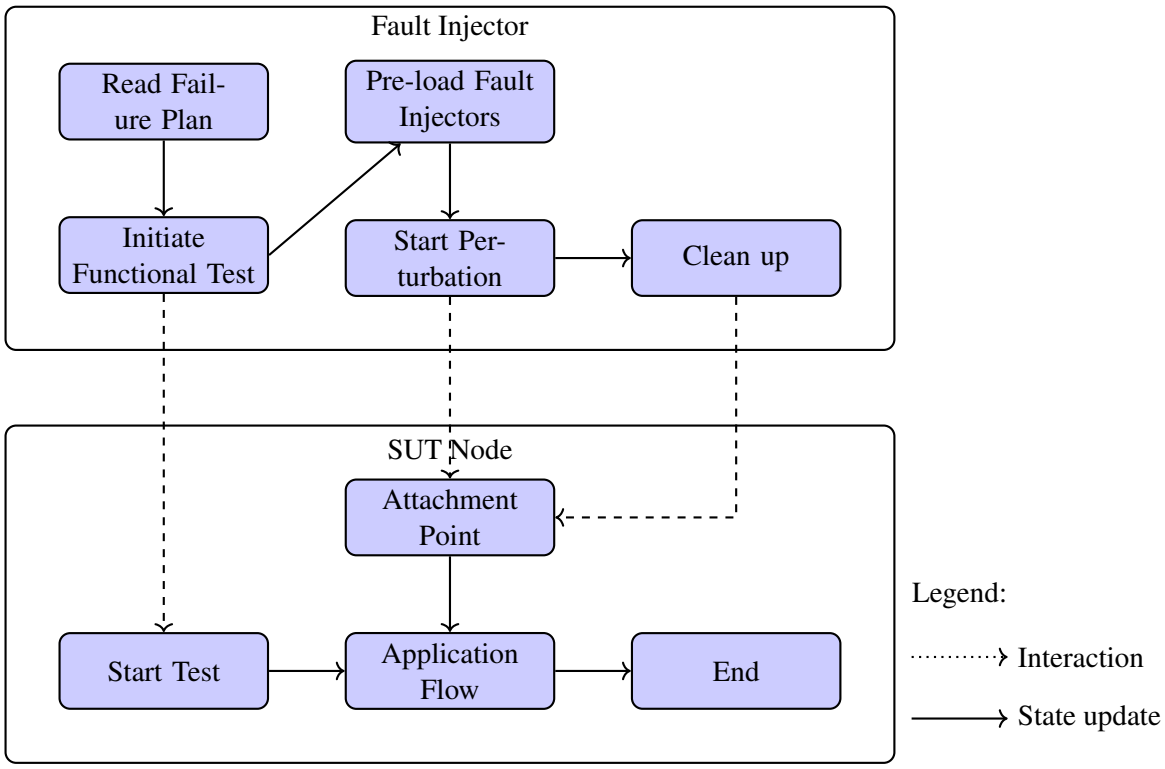


Figure 4.3: Interaction between YAFI and the SUT Node in a single execution flow.

duration. Since we enforce a connection of one-to-many agents, with one-to-one agents to SUT, the agent does not have to send the source, as the orchestrator has this knowledge itself. Other types of events are detectable state changes of the target application, sent from active collectors. Sending these events allows the orchestrator to synchronize with the workload.

4.4 Implementation and Integration of YAFI

An implementation has been made for evaluation, available on GitHub[1]. The main evaluation is presented in Chapter 5, where the fault injector implementation is used in a case study. As verification of the portability, we also apply the system on a different target system. The next section gives some information about the requirements to use the framework. Then we look at integrability of other systems into YAFI.

4.4.1 Portability: Open Source Project Implementation

To test the portability across multiple target applications, we apply the framework implementation on a new, open-source, system. Since the created implementation is built for a

[1]https://github.com/freshcoders/YAFI

Java application, we look for a Java system to test. A Zookeeper Demo application[2] was chosen, for being a distributed application and having straightforward setup. ZooKeeper provides coordination for managing and coordinating distributed applications.

System Requirements for Implementation

The current YAFI implementation relies on certain prerequisites to ensure integration with target systems. The specific requirements include:

Java Both the orchestrator node and the target nodes must have Java 11+ installed.

Byteman Byteman is a bytecode manipulation tool that enables injection of faults into the target application. It must be available on the target nodes to facilitate fault injection. Byteman attaches to the Java Virtual Machine, thereby providing fault injection options.

By fulfilling these requirements, the framework can run on the target system. The implementation is capable of generating new Byteman rules for the new application without requiring user input. Using tracing, we can gather communication information and inject delays into the system. We are not able to run meaningful experiment and lack an evaluation on the ZooKeeper application. Nevertheless, we gain meaningful information by porting the application. The created fault injectors, failure plan generators and collectors are all reusable.

4.4.2 Integration and Transferability

Besides portability of our system to be applied on other contexts such as applications and environments, we briefly look at integrability. One of the main ideas of YAFI is to be extendable. During the study of literature in the context of fault injection, it was clear that many fault injectors comprise components, with ideas shared between them. Nevertheless, this is not beneficial to any of the injection tools as they are all built into their own ecosystem.

It is our belief that not all new research on fault injection techniques requires a system built from scratch. This section will show some high-level steps that can be followed to integrate (existing) systems into YAFI. Benefits of this approach are:

- Less implementation work, saving time and resources on development.
- Research reproduction is made easier. Following standard protocols is easier than using a newly created system and prevents researchers from having to write and provide documentation.
- Lower thresholds for companies to test out fault injection on their system with minimal setup.

[2] https://github.com/bkatwal/zookeeper-demo

- Combinations of techniques are made possible, sometimes without any additional work, plugging in an extension from other injection techniques into another could yield a broader set of results.

To integrate an existing fault injection system into YAFI, all that is required is copying the functionality of their components. We can divide this up in steps some of which are optional:

1. **Identify component functionality**: Inside the architecture of fault injection systems, we find many components that are intertwined. Here we identify the essential features, relating to YAFI components.

   a) (optional) Failure plan generator component.

   b) Fault Injector component.

   c) (optional) Analyzer component.

   d) (optional) Collector component.

   e) (optional) Workload component.

   f) Controller component.

2. **Decide on the proxy**: The proxy that will be used for the fault injector needs to be identified. This can be a *connector*, capable of injecting directly into the attack surface, through an agent that is deployed on the target node, or a custom implementation.

3. **Build identified components**: Build all separated functionalities into components, according to YAFI's protocols. The controller is omitted as its features are absorbed by other components and the YAFI.

4. **Specify failure plans**: A set of failure plans need to be specified. This can be done using automatic generation through generators (if step 1a yielded features) or by creating custom configurations by hand.

5. **Workload specification** Create the workload through an abstraction layer between YAFI and the target application.

Practical Example: NEAT

We can see how the functionality of NEAT [7] can be adopted following these steps. NEAT's implementation is not very modular and several component implementations are defined per test case. In their paper, they define a "test engine", which runs the workloads, injects faults, and tests correctness. In our implementation, all of these functionalities are separate components, linked together by the failure plan and executed by the orchestrator. We now explain each of the steps, relating implementation details of NEAT to our implementation and framework components.

Identify component functionality First, we identify and separate the functionalities of the implementation of NEAT[3]. It has a *partitioner*, which can be classified as the *fault injector* component. The partitioner is capable of creating two types of fault: full partition and partial partition. These will become fault types, supported by the *fault_model* protocol.

Decide on the proxy Before building the injector that builds this fault, we inspect their injection method as part of step 2. The iptables partitioner injects iptable flow rules through SSH[4]. This is directly compatible with an implementation of a proxy that we have built, connecting over SSH[5]. Because this SSH connector is a proxy, it is compatible with all fault injectors, and we can proceed to the third step.

Build identified components In step 3, we only look at the *injector* component. As both the YAFI and NEAT implementations are built in Java, it is straightforward to integrate the partitioner as an injector. If the implementation needs to be realized, a `PartitionInjector` can be built, which would look very similar to our existing `NetEmInjector`[6]. All that is required for implementation is parsing of configuration that goes into the *fault_model* and we can call partitioning methods that are already present in NEAT's implementation from there, with appropriate configuration.

Specify failure plans Step 4 is then followed to define the faults that will be run during our campaign. In the NEAT example, we can see that this is done using a simple test in a programmable class[7]. Our framework offers this through the realization of the *fault_model* protocol as a failure plan, defined in YAML format. In the example that NEAT provides, a manual separation of groups is made, which is also possible to do with a custom *generator* component.

Workload specification Finally, the workload has to be defined and run, which is dependent on the target application.

As a conclusion for this example, we can see that the functionalities of NEAT are very easy to adopt now that we already have a framework core implementation and some extensions. To implement the faults that NEAT offers, very little effort is required compared to building the system from scratch.

[3] https://github.com/UWASL/NEAT

[4] https://github.com/UWASL/NEAT/blob/5d31956a0f9fff03de837aff7ded825e089f52b9/src/main/java/netpart/partitioner/impl/iptables/IpTablesPartitioner.java#LL118C4-L118C14

[5] https://github.com/freshcoders/YAFI/blob/tmp/src/main/java/nl/freshcoders/fit/connection/SshConnection.java

[6] https://github.com/freshcoders/YAFI/blob/tmp/src/main/java/nl/freshcoders/fit/injector/NetEmInjector.java

[7] https://github.com/UWASL/NEAT/blob/master/src/main/java/example/basic/BasicTest.java

4.5 Discussion and Conclusions

With our presented architecture and accompanying protocol specifications, we have a basis for platform-independent injection, capable of building an implementation. We have seen a set of challenges, identified during the design process to answer RQ1.3. In response to RQ1, we propose YAFI, a platform-independent fault injection framework. It can be used as a solution for testing the resilience of distributed applications. Before we make a concluding statement, we will first discuss our choices and options for each challenge. Then we evaluate YAFI using a case study in Chapter 5 and give our final conclusions on RQ1.

4.5.1 Component Challenges and Potential

Revisiting Challenge 1, main components are part of the core architecture because of the functionalities they provide. Each has its own responsibility and can heavily impact the performance of the fault injection process. We examine the components failure plan generator, fault injector, and analyzer, looking at their potential improvements and features.

Failure Plan Generator

The generator is the source of all failure plans for exploring fault scenarios in a system. A generator can initialize a large set of failure plans, even without knowledge of the target system. Using a random testing approach, we can generate fault and trigger combinations that are executed at a random point in an application. This is related to the exponential number of states in a system, known as the state explosion problem [20], as there are almost limitless possibilities for faults to be injected into a system. Executing all configurations is not possible, and we need to decide on a search strategy to explore states of the SUT. A big part of the search is to limit the number of tests that the injection system executes during a campaign. This process is known as pruning or (dynamic) test reduction [55].

Fault Injector

YAFI is designed to be used on arbitrary target systems. Therefore, not all injector implementations will be compatible with all target systems. However, there are many types of injectors that may be generic or apply to a common environment, shared between many software applications. An example is network manipulation through the network drivers of an operating system. An injector built for this use case is not dependent on the target application.

Analyzer

The analyzer component is responsible for deciding the correctness of the system output. Often the correctness of the output depends on the input. Known as the oracle problem, this subject has been studied for a long time [14]. Analyzers can be written with specific applications and test cases in mind and do not have to be oracles. Simpler implementations

of analyzers are capable of detecting failures [71]. Checkers and assertions are similar methods (see Section 2.3.2) and can be implemented in an analyzer.

4.5.2 Platform-independence and Portability

Our goal is to be platform-independent, which can be accomplished on several levels. We look at the different types of attack surfaces in Challenge 2. In software development, we commonly talk about language and OS agnostic systems. Our definition of platform-independence concerns these types but importantly includes orchestration systems. Many fault injection systems target a single type of platform, such as container orchestration system like Kubernetes, Docker, and others. In doing so, they gain a consistent API to interact with and inject faults onto, and at the same time, they lose a large set of target systems. For example, bare-metal systems are not compatible with the framework, since they lack the API that is targeted by these types of systems. YAFI supports all these platforms. It is also worth mentioning, even though out of our scope, YAFI can be extended with fault injectors that are drivers for hardware fault injection, along with compatible collectors and perform HWIFI [67].

Our system is language-agnostic as well as OS-independent, while still being capable of injecting on platforms such as Kubernetes. We achieve this by building a set of protocols that can be implemented in multiple languages, even at the level of the framework itself. Additionally, a proxy is introduced between the injector and the attack surface, to simplify the injection process and decouple the orchestrator from the injection target. The orchestrator can run on the test execution server implemented in one language and communicate to an agent on a test node implemented in another. As long as the injectors implement fault types that are compatible with the target system, failure plans can be executed. Failure plans may also be generated from any source, as long as the output is in the shape of the fault model, defined in Table B.1. This is a result of the separation between the framework core and the extension implementation.

We also highlight a key observation on portability. When implementing the framework, we need to do so in a programming language. As soon as we do this, we may limit the degree of portability; even when building in a cross-platform language. This is a limitation that can be addressed by using the idea of our protocol-compliant extensions. It is often straightforward to modify an extension so that it will work for the new system. With the communication protocols enforced by YAFI, the compatibility of this new extension is ensured. Similarly, there are language-specific injectors and OS-specific injectors that obviously are not portable to all other systems. For Challenge 3, these are also key ideas, as we can interchange components as long as the new SUT is compatible.

4.5.3 Limitations and Consequences of Events

With events responsible for all the faults to be triggered (see Section 4.3.3), we essentially have an event-driven architecture [63]. Some events may be complex and require custom implementations inside the framework. Since these conditions are triggers, as long as they can be evaluated, they can be implemented with relative ease. However, the current ar-

chitecture requires manual implementation for each new trigger event. Although this is an undesirable trait of a generic architecture, it enables usable workload execution from the orchestrator.

4.5.4 Determinism in Experiments

Challenge 4 addresses determinism of repeated experiments of the fault injection campaign. To be deterministic, faultloads (the failure plans, combined with the workload) need to have consistent execution. Any randomness in the system should be reduced and removed when possible. We are not able to have perfect control of all parts of the system, thus, deviations between failure campaigns is not impossible as they may be dependent on factors out of our control. To minimize the risk of these deviations, we work with triggers in the failure plans that should fire at similar times or configurations in each execution. These states can be characterized by a combination of events.

In addition, if failure plans have a non-fixed target, implementation should support pseudorandom number generation. This will help to support repeatability and prevents determinism to be affected by one fewer variable. However, YAFI cannot enforce determinism completely.

4.5.5 Intrusiveness on the SUT

We treat the SUT as a gray-box [49], because we allow users to define workloads adapted to the functionality of their system. YAFI does not rely on white-box methods with static analysis or instrumentation. Due to the objective of creating a platform-independent injection framework, the intrusiveness on the SUT should remain minimal. This causes the injection engine to be used in a similar fashion as LWFI from NFtape, implementable as an extension. This is beneficial since no implementations can fit all target applications, environments, and platforms. Keeping all components responsible for their own features and independent of the target where possible encourages reuse and creates a low barrier for adoption.

4.5.6 External Workload Execution

The design of YAFI offers limited support for the execution of workloads on the SUT. We define an abstraction layer in Section 4.3.1, to execute common instructions and require users to implement their methods and workload support. This abstraction layer provides a partial answer to Challenge 5, allowing us to execute workloads on the SUT through a simple interface. However, the implementation has its limits in terms of synchronization with the failure plan. Ideally, we want to have more control over the states of the system. We can achieve this by creating a more dedicated layer, that is capable of validating states of target-specific operations in a more uniform manner and sending synchronization events to the orchestrator at appropriate timings.

Moreover, the workload component has yet to be separated from the orchestrator and requires protocol definitions. There is a choice between allowing arbitrary operations and making an execution flow, and defining a specific set of operations that will be executed as the faultload is executed. Both require an additional plan, either in addition to the failure

plan (as a workload plan), or in combination with the failure plan. The latter is used in FaultSee [59] and Frisbee [9], where they implement a domain-specific language to execute faultloads.

Chapter 5

A Case Study at Adyen: Empirical Evaluation

In this chapter, we evaluate the framework designed in Chapter 4. First, we implement the framework components for Adyen, by building injectors for the faults we want to inject. Then, we conducted a case study at Adyen where we integrated the framework components and extensions, such as injectors, generators and a simple analyzer into a functional proof-of-concept to test one of their distributed applications.

We present the findings of our experimental results of the case study addressing RQ2: *How effective is the fault injection framework in detecting bugs?*
RQ2 is answered by two subquestions, each answered by obtaining information through its own type of fault injection campaign. The first is RQ2.1, which has the goal of detecting unknown bugs, this is reflected in the first campaign. We execute a set of experiments without prior knowledge of the SUT, exploring with a breadth-first strategy, which we refer to as the *untargeted campaign*. The second is RQ2.2, which evaluates the system's ability to detect a known bug, a common benchmarking approach for fault injection systems [50, 55]. To achieve this, we direct our injections to an area of the system where we have located a bug, by manual inspection of the SUT. We refer to this as the *targeted campaign*. By examining the results of these campaigns, we make observations that give information about the effectiveness in bug detection.

For the evaluation of the framework, we make use the distortion evaluation technique by Van der Kouwe et al. [70] measuring the difference in input and output faultload. Van der Kouwe et al. also introduce the terminology fidelity as the degree to which input faultload and the executed faults align with each other. We also use the rate of detection as an evaluation metric.

We first examine the individual campaigns and then address limitations during the process. Finally, we conclude by discussing the results and providing our evaluation of the campaign results.

5.1 Fault Injection Campaigns

This section presents specific details about the fault injection campaigns that were run as part of the case study. The target application or the system under test (SUT) is called the Nexo Router, explained briefly in Section 5.1.1. Then we look at the implementation details used for the target application. Implementation details include the extensions presented in Section 4.2. In this stage, we set out to identify potential fault types that could significantly impact a distributed system, causing outages or inconsistent states, by leveraging the expertise of developers at Adyen. Afterward, we proceed to apply these potential faults in a practical experimental execution, transforming the theoretical understanding of system vulnerabilities into concrete tests that effectively simulate their occurrence and potential impact within the system. Injection of the faults is subject to a phenomenon similar to fidelity called representativeness. Injected faults should be representative of the actual faults [53, 57]. Finally, using the case study, insight is obtained to answer research questions 2.1 and 2.2. In this case study, we perform two types of campaigns. The first is an untargeted unguided search campaign where we use faults, chosen based on developer insights, and only use automatically generated failure plans. The plans are executed and generate data that can be analyzed and yield observations. Because this unguided search explores the system automatically, it will provide information about the ability to detect bugs in the system. The second campaign is done with a vulnerability (bug) in mind and focuses on exposing it using hand-crafted faults as part of automatically generated failure plans. This will show us the capability of the system to follow the fault model during injection (fidelity) and provide information about the ability to detect bugs. Information gathered during the campaign will be used to evaluate the effectiveness of the framework.

5.1.1 Experimental Setup

For both experiments, a common setup is used. We explain the setup here, before diving into the executions of the failure plan campaigns. We start with an explanation of the module under test: the Nexo Router. For the Nexo Router, we wanted to test a limited set of faults, for a higher probability of finding interesting system resilience. The options and choices for fault types are presented in this section. YAFI defines component specifications that need implementation. The implementations are also presented, to be used within both our campaigns.

As our case study, a distributed application developed at Adyen called Nexo Router is evaluated. We refer to this module as SUT, target application, case study application, or Nexo Router. The Nexo Router is part of one of their Adyen's business models, in-store payments, internally and in the API reference[1] better known as in-person payments. With this form of payment, merchants (stores) often have a setup with a point-of-sale (POS), connecting to a Pin Entry Device (PED) to allow customers to pay. The core of the Nexo Router application is about 24 KLOC, without any of its internal dependencies.

We determined interesting scenarios, their faults, and possible bug manifestation through discussions with Adyen's Nexo Router engineers and other experts in distributed systems

[1] https://docs.adyen.com/point-of-sale

and observability. We discussed scenario setups that could trigger undesired behavior as well as the faults that could provoke them. For example, intermittent network failures could provoke timeouts or broadcast failures, potentially overloading network capacity due to numerous retries, thus causing potential system failures. Data center outages could lead to failures in load balancers, disrupting overall system functionality due to an excess of connections. A partial network partition could cause inconsistent states and invalid routes. Simultaneous registration might create an invalid route, as existing information could be overwritten. Lastly, clock skew might lead to message denial if the clock lags behind previous messages from other nodes.

Having identified these potential faults and causal relationships, we designed campaigns that would simulate several of these conditions, enabling us to observe how the system responded and assess its resilience. The simultaneous registration concern is a critical path to test and we design a functional test that represents this behavior. The workload of the SUT will be executed through the functional test. For our case, we do not test large scale outages, rather looking at simple network faults that were discussed with engineers to be interesting. With additional knowledge of the system, such as timeout configuration, network delay is chosen a primary fault type to run our campaigns with. Moreover, network delays (high-latency links), when exaggerated, can cause stale-reads and system crashes, as we saw in Section 2.1.2, which we can look for when evaluating. Timeout configurations help us limit the number of tests that we have to run, as our system now does not have to explore states and timeout settings without knowledge about them.

The test setup consists of a single machine, emulating multiple nodes. By running duplicate instances of the SUT on different ports (all as standalone applications), the experimental setup is similar to a real-world setup. The "test execution server" role was fulfilled in this case by a local MacBook Pro with a 3.2 GHz M1 processor. Our campaigns are run with at most 3 instances of the target application.

The campaigns target simultaneous registration faults, by injecting network delay faults with the goal of creating an inconsistent state in the SUT.

5.1.2 Implementation Details of the Fault Injector at Adyen

To perform the evaluation of the framework, we utilize the implementation as presented in Section 4.2. On top of the framework, we build the generators, monitors and injectors. They serve to extract observability metrics, generate failure plans and inject the desired faults. We perform the case study with the Nexo Router as a target.

A crucial problem is the execution of the failure plan in the target system, during the execution of the workload. We have to execute faults at the right time in the execution, the *when* in WWW (see Section 2.1.4). Some systems integrate the management of SUT workloads into the fault model directly. Allowing the system to execute the faults at specific timings of the workload. Applications that are testable through standard benchmarks can effectively make use of this [9]. The throughput is stable and the injection of a fault will perturb the state, affecting the throughput while the failure lasts. Deciding when to inject can be difficult with an arbitrary target system, which YAFI is designed for. We do not want to manually start and stop the target application and its workload each time and we

want to avoid writing custom workload runners. Without having designated parts of the application where the injector knows it can start injecting and testing for results from an external point, we require active communication from the SUT at run-time. As discussed in Section 4.3.1, YAFI does not have a well-defined protocol specification to follow for the execution of workload. In our implementation, this is facilitated by a test script, responsible for controlling the SUT. The script has four steps, which are: startup, workload execution, verification, and finally shutdown. In this way, we can freely inject faults between startup and workload execution. This is similar to the approach of NEAT [7], for the abstraction of the target system. NEAT uses an API through an interface that has to be implemented, containing start and stop methods, as well as a method to obtain the system status. The API (in our case, a script) prevents us from forcing constraints on the target system. Since the framework works with triggers, there are many possibilities to trigger faults. Some faults can be triggered at the first opportunity and remain dormant in the application. Throughout this section, the Java Virtual Machine (JVM) chaos injector is highlighted.

Using a running example, we give more details on the implementation of the targeted campaign's experiments. The example we use injects a network delay on the Nexo Router. For executing the test cases, we follow the logic of the framework's orchestrator phases. There are four phases in an execution of the framework:

Execute Reference Run In this stage, we execute a single functional test. During the test, we only observe the behavior of the application and collect data using collectors as D_{ref}. These data are used in the next stage and in the final stage.

Analyze Data After the test has finished, we enter the analysis step. Data are analyzed and converted into usable sets of output for a coupled failure plan generator component. These generators then create sets of failure plans.

Execute Perturbed Run Here, we execute the generated failure plans. Similarly to the first stage, we observe the behavior of the application during a test, collecting the data. However, during this phase, the application is perturbed while running the functional test. Collectors create a second set of data, D_{faulty}. In our implementation, automatic feedback is disabled to limit the size of campaigns it would cause.

Finalize All output is compared to the reference run and stored for manual analysis. Again, we are limited in terms of the realization of components. Our analyzers are implemented as simple checks that validate the equality of D_{ref} and D_{faulty}.

For the implementation a socket connection is used to connect to the agents. We will now briefly explain how we integrate each component of the framework as they are required in each of the four phases, for reference, see Table B.13. This includes how we extract and obtain observability metrics, the fault generation mechanism, and the injector that was used.

Execute Reference Run

The YAFI implementation initially performs a reference run. The reference run is an execution of the SUT without interference of the fault injector. Under these circumstances, the

output is assumed to be correct. This phase has an optional dependency on the collector component. If the SUT provides its own active collection mechanism (see Section 4.2.4), the collector is not required in this step and we can use passive collection in the analyze phase. The Nexo Router provides information about its execution through OpenTelemetry[2], which provides information about (distributed) execution traces and metrics, thus we do not define an active collector. Without built-in tracing, we would have had to rely on instrumentation, use logging output, or build an active collector, capable of extracting online information.

With the collection ready, we can execute the workload on the SUT. The workload component is a required dependency (if we do not want to manually execute the SUT). As mentioned above, this is currently loosely defined by an abstraction layer, using a script.

For our running example, we look at the execution of the reference run as present in the WaitForReferenceExecutionState[3] class, as seen in Listing 5.1. The `sut-control` script is invoked to execute our functional test.

```
public void execute(Orchestrator orchestrator) {
    String controlScript = System
        .getProperty("functional.dir", "/usr/bin/") + "sut-control.sh";

    orchestrator.getFailurePlanRunner()
        .logUserAction("starting functional test");

    LocalConnection.executeCommand("bash " + controlScript + " start");
    LocalConnection.executeCommand("bash " + controlScript + " exec");
    String res = LocalConnection
        .executeCommand("bash " + controlScript + " test");
    LocalConnection.executeCommand("bash " + controlScript + " stop");

    orchestrator.getFailurePlanRunner().setInvariant("response", res);
}
```

Listing 5.1: Code snippet executing the workload in the reference run.

This concludes the first step, now that the reference run has been executed and data has been collected, it needs to be analyzed and turned into failure plans.

Analyze Data

The next step is to use trace information generated from the reference run. The traces provide information about the execution and are used by YAFI to build failure plans.

To do this, YAFI now requires a passive collector, that retrieves the data of OpenTelemetry. This implementation has to follow the protocols: collector output (see Table B.3) and generation input (see Table B.6). We implemented a collector[4] and a compatible generator.

[2] `https://opentelemetry.io/`

[3] `https://github.com/freshcoders/YAFI/blob/tmp/src/main/java/nl/freshcoders/fit/plan/workload/states/WaitForReferenceExecutionState.java`

[4] `https://github.com/freshcoders/YAFI/blob/tmp/src/main/java/nl/freshcoders/fit/tracing/TraceReader.java`

The code of Listing 5.2 shows how the collector fetches data from a source. The collector is coupled with a compatible generator[5], as per the requirements of YAFI.

```
// wrapper around the elasticsearch client
private final ElasticSource elasticSource;
public List<RelatedSpans> getCommunicationTraces() {
    List<RelatedSpans> relatedSpans = new ArrayList<>();
    Set<String> traceList = elasticSource.getTraceList();
    for (String traceId : traceList) {
        Trace trace = elasticSource.buildTrace(traceId);
        List<RelatedSpans> communicationRelations = RelatedSpans
            .findCommunicationRelation(trace);
        // the found spans can now be analyzed and turned into faults
        relatedSpans.addAll(communicationRelations.stream()
            .filter(c -> c.origin != null)
            .collect(Collectors.toList()));
    }
    return relatedSpans;
}
```

Listing 5.2: Code snippet collecting elasticsearch data.

The use of OpenTelemetry is convenient, information that may be of use can be added easily to the output metrics for making decisions about the validity of a response. We also gain information about the system architecture, due to the nature of distributed tracing. Calls between traced servers will generate traces showing their interactions. We can use this information to introduce network faults in subsequent experiments. A failure plan specifies the target nodes, which will be decided at run-time in various combinations and capacities based on the target mode (see Table B.14). If a trace from the reference run shows interesting properties, such as communication with another host, this is taken as a notable event. From this event, we can generate several faults that can be included in the failure plan. This procedure causes a single functional test to generate many failure plans, and in combination with knowledge of the system, such as the timeout duration, we can build tests for the system that could reveal critical information.

Execute Perturbed Run

With the failure plans generated, we move on to the next phase where we perform a perturbed run. During this run, make use of the previously defined collector and workload components.

The difference between the reference run and the perturbed run is the injection of faults (the perturbation of the system).

In the case study that was performed, a single injector was used to manipulate the SUT. This injector is capable of injecting faults into the JVM (see class: JvmChaosInjector[6]).

[5] https://github.com/freshcoders/YAFI/blob/tmp/src/main/java/nl/freshcoders/fit/generator/CommunicatorPerturbationGeneration.java

[6] https://github.com/freshcoders/YAFI/blob/tmp/src/main/java/nl/freshcoders/fit/injector/JvmChaosInjector.java

Currently, this implementation uses Byteman, a bytecode manipulation tool, to perturb the standard behavior of the SUT.

Three types of faults were injected as part of this campaigns: setting a fixed clock, introducing delay, and throwing exceptions. The faults were injected with the following Byteman rules, which are specifications that follow the WWW principle of fault injection. In a rule, we can specify a target class, method, and location (where). We also specify the behavior (what) and the condition (when). A fixed clock rule is defined as an example in Listing 5.3. It causes the Java call `System.currentTimeMillis()` to return a fixed value as long as the rule is active, since the condition (see line 5 of the rule) is always **TRUE**. Byteman uses the JVM class loader to make sure that the correct class is modified with injection. In this case, the target class is a built-in Java class.

Listing 5.3: Byteman rule for fixed clock

```
1 RULE eventId 1071372933
2 CLASS java.lang.System
3 METHOD currentTimeMillis
4 AT RETURN
5 IF TRUE
6 DO traceln("injection-event:1071372933");
7   return 1680075138445L;
8 ENDRULE
```

The fixed clock behavior was used to force the system into situations that can happen organically but which are unlikely to be replicated. Using a fixed clock at a certain time in the experiment forces the system to enter a path that we want to test. Without this, we would harm the determinism of experiments.

As for our running example, the rule that we activate is targets a single attack point: a class and its method. The rule can be seen in Listing 5.4. Specifically, it simulates packet delay, since it does not affect the whole network, but only a specific operation in the system.

Listing 5.4: Byteman rule for network delay

```
1 RULE eventId 1956301459
2 INTERFACE com.adyen.nexorouter.Broadcast
3 METHOD listDevices
4 AT RETURN
5 IF TRUE
6 DO traceln("injection-event:1956301459");
7   Thread.sleep(9000);
8 ENDRULE
```

Finalize

In the final phase of a campaign, we require an analyzer. Like the previous components, it follows its respective protocol, defined in Table B.4 and Table B.5. This component receives its input data from the collector output (like the generator).

The analyzer outputs data in two ways: a boolean result and a custom result. The custom result is not automatically used by the framework, but can be used to present data to

the user visually or in other formats. The boolean result is used as the test result for the experiment, the framework will notify the user when this result is `false`.

This concludes the four phases of the orchestrator.

5.1.3 Untargeted Fault Injection

We now examine the results and influence of the first campaign experiments on the performance of the system and its execution correctness. The campaigns presented here provide information for answering RQ2.1:
How can system metrics be used to validate system output for detecting unknown bugs?
Our campaign is executed using a simple functional test, covering part of the main logic of the SUT. Through evaluation of the results of the experiments, we will determine whether the implementation and its methods are effective in detecting bugs.

In Figure 5.1, we see the back-end latency of the application during the functional test. In this functional test, only a single call to the system is made that logs this metric. On the x-axis the run-time of the campaign is shown; each test case is separated into a section denoted by the vertical dashed lines. Each of the sections are named at the top of these lines, starting with the reference run and its analysis, followed by all variations of test cases. Variants of test cases are now denoted by their fault type, followed by the generation number. Some faults are combinations of multiple faults and are listed as "comb" in the graph. The campaign concludes with the overall analysis, as the final section in the graph. To see the actual results of this overall analysis, we have to take a look at the source data. However, in this case no analyzers raised an alert. The y-axis shows the back-end latency in milliseconds, which is essentially the response time of the Nexo Router when a request is made. Latency is represented on a log-scale to be able to show the fluctuations of the fast responses, while having outliers present.

Before we look into the results, we need to look at the failure plan of some of the experiments. If we take the delay configuration and look at the corresponding execution metrics, we see an expectation mismatch. Listing 5.5 shows the YAML configuration for experiment *delay-2*. Looking at line 16, we see a delay configuration of 9000 milliseconds. However, the section of *delay-2* in Figure 5.1 does not reflect this delay. This can be related to the concept of distortion in fault injection. We see a misalignment in the expected results of the faults and the actual results of the fault experiments. Even though the fault activates, the resilience of the code handles the injected fault and uses alternative methods to provide the user with a response.

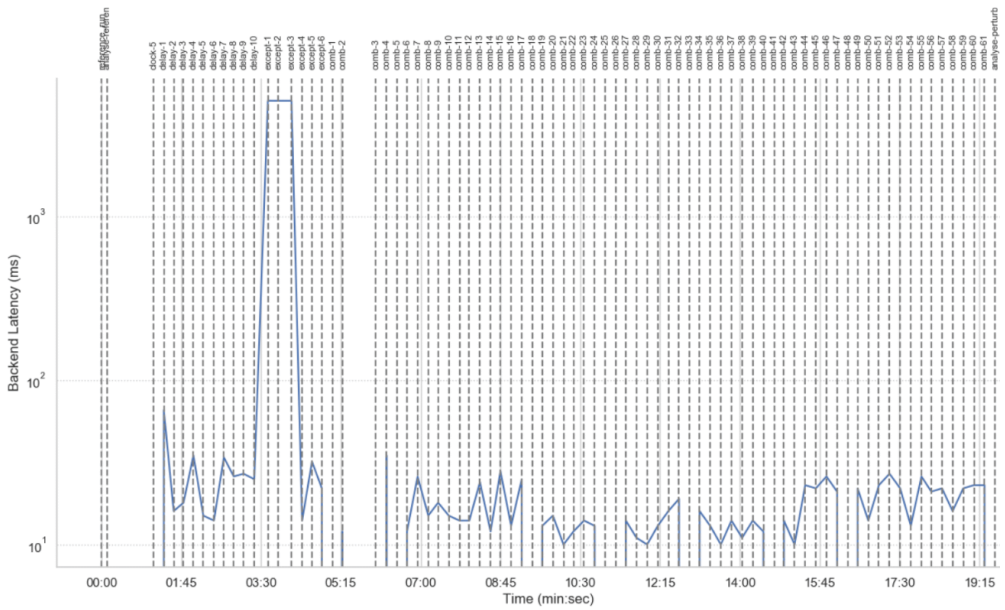


Figure 5.1: Experiment results of a single campaign, back-end latency on a logarithmic scale on the y-axis. The dashed lines separate sections, corresponding to the start of each experiment.

Listing 5.5: Failure plan YAML for delay-2

```
hosts:
- uid: 'host-dev-Y '
  port: 13014
  ip: 127.0.0.1
- uid: 'host-dev-X '
  port: 13013
  ip: 127.0.0.1
events:
- fault:
    occurrence:
      target: host-dev-Y
      timing: once
      location:
        method: listDevices
        class: interface com.adyen.nexorouter.Broadcast
    config: '9000'
    type: delay
  trigger:
    arguments:
      time: '0'
      duration: '0' # 0 means infinite
    type: clock
```

The trace data shows an immediate response from the node part of the package delay fault. A new method is observed in comparison with the reference run. This code has not been

seen before, and as such, no failure plans have been created that do manipulations in such an execution. Interestingly, this factor of the fault loads has not been deemed as intuitively important in the paper about distortion. Ideally, these newly found trace data are fed to the generator, which would then build more failure plans for these operations. In this case study, this step was performed manually to avoid having too many experiments to run. The reason for this is the lack of efficient generation and pruning of unnecessary experiments. With this single new generation configuration, our fault injection campaign size more than has doubled in amount of failure plans. Our campaign now consists of 172 plans. Rerunning the YAFI tool now gives us the data visible in Figure 5.2. If we look at the experiments past the 24 minute mark, we see the injected faults reflected in the latency.

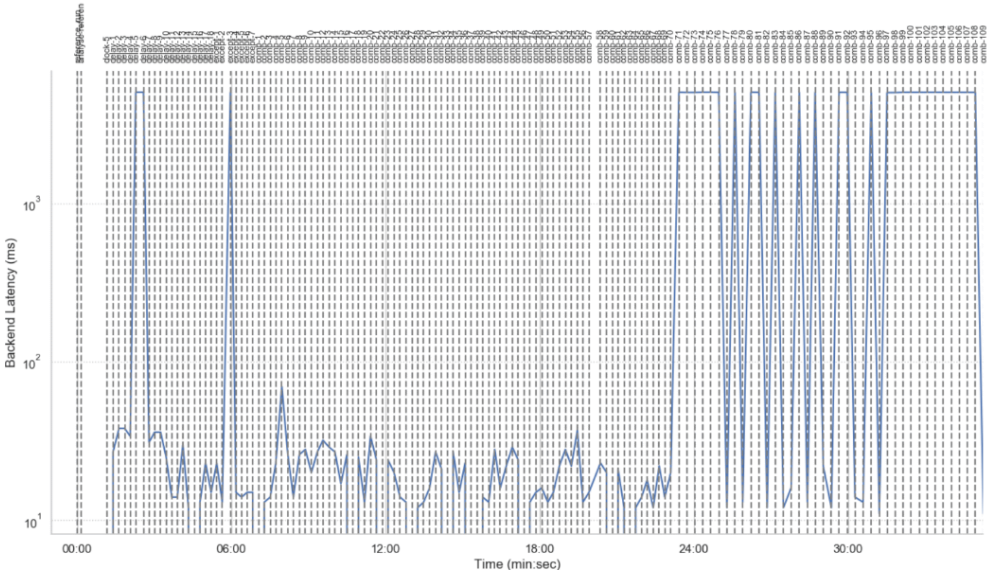


Figure 5.2: Experiment results of a single campaign updated with feedback, back-end latency on a logarithmic scale on the y-axis. The dashed lines separate sections, corresponding to the start of each experiment.

These results tell us that feedback on the generator from generated executions should be used. An execution of the system where a single fault is injected activates other paths in the system. The system can analyze this execution and create a new fault. This is needed, since we do not get enough information about the system in a single execution. This can lead to many more system states and consequently generate a high number of failure plans. Some of these failure plans are redundant and unproductive to execute, causing the system to be less effective. Because of this, we need to weigh the value of exploring the superficial application flow against exploring deeper paths. Note that seeing a quick response from the application also advocates for the system being resilient in certain fault cases.

Although we have now achieved a visible impact on the SUT through manipulation, this does not mean that the tool is effective. Before we look deeper into this, it is useful to look at more response data, which come from the experiments. So far, we have only seen figures with back-end latency as a response metric. While this is a useful metric when evaluating

overall performance and detecting faults that impact response time, it may not provide sufficient information to determine whether a particular operation is valid or not. Although we gain information in these cases, we require additional metrics to determine the validity of arbitrary operations. We would like to determine the validity of the functional, done through comparison with the reference run, which may include the outcome of operations. Monitors may probe the system during execution to obtain comparable response data.

The automatically generated failure plans and their experiments did not result in finding any bugs in the application.

> **Observation 1**: A broad untargeted search, as a result of automated failure plan generation, is unlikely to find bugs.

Even though the system was unable to locate actual bugs, a side effect of running the tool resulted in finding unexpected behavior. The type of unexpected behavior that is found can be hard to test with functional testing alone. One feature of the fault injector is that it enables us to do many iterations of executions of the applications with slight deviations. During testing, one of these deviations led to the execution of two simultaneous operations, which caused unnecessary events to occur. This was noticed when we looked at the data from the tests. The information extracted from the tests of these executions was different from the expected behavior.

> **Observation 2**: Output metrics can be utilized to assess the validity of an application's execution, thereby enabling the identification of potential issues that may not be detected through standard testing methods.

The experiments were repeated three times to show the fault injection system is capable of reproducing the same state across multiple runs. Most of the experiments across different runs ended up in the same result, showing latencies corresponding to the faults that were injected. However, some of the experiments suffered from external influence, causing the result of perturbed runs to be invalid. This was noticed by the simple analyzer, which expected response data equal to the reference run, but found an error instead. The cause of this was the inability of the system to restart the target application in rare cases. This caused multiple instances to run on the same port and resulting in errors in one case. Another case saw the effects of the previous experiment persist in the next, also having unwanted side effects. Observation 2 is supported by the fact that the effects of faults are observed and can be used to determine if the execution failed or not.

Another outcome of this untargeted experiment is related to various unexpected behavior that was found by analyzing the collected data. Output showed multiple connections to other hosts being made, where one was expected. We observed this behavior in 6 of the 172 experiments in the first run. Repeated runs saw different results as an effect of missing data. This was caused by an infinite loop in the code, which generated too many traces for an external trace collector. This collector then dropped the traces, due to their size. In this particular case, we observed non- deterministic behavior, where the output or result was not consistently reproducible under the same conditions.

Though exposing this is not a unique feature to fault injectors and end-to-end tests is also capable of showing this, no additional effort was required, other than manual inspection of the output. It shows the system can assist in executing non-trivial configurations without much effort:

Observation 3: The ability of the fault injector to precisely time and execute operations enables the collection of accurate and informative data, facilitating the identification of potential issues.

We can do a few things to increase the probability that bugs will be found. By specifically targeting areas of the system that are crucial and sensitive, we have a higher probability of finding faults. The parts we target are sensitive due to their crucial and complicated logic since they have high code churn, which has been linked to higher probability of bugs [32]. In the next section, we look at how targeted fault injection can be used to explore the system.

5.1.4 Targeted Fault Injection

As a second type of fault injection campaign, we look at a bug that we found during inspection of the SUT and perform a targeted fault injection. The goal of this case study is to gather more information for RQ2.2:
Is the system capable of detecting known bugs using targeted fault injection?
We run a campaign that injects specific faults, leading to a potential bug in the system. If the bug occurs and the system is able to detect it, we gain valuable information about the injection capabilities of the system. It will also become clear whether or not the system metrics used to assess the validity of the execution are enough for automated detection.

To determine the system's ability to detect bugs when present, we look at a case that was found during the framework implementation. A bug that was located is present in the Conflict-free Replicated Data Type (CRDT) logic. The CRDT is responsible for determining if a route to a registered device should be updated. Since the goal of a CRDT is to achieve state convergence across nodes after processing all messages have arrived, it should not have any inconsistencies. However, the result of the bug we found would cause different states to occur in certain nodes after all messages have converged. For this bug to manifest, a precondition must be met. In its simplest form, we assume that there are two versions of the application running as part of a single cluster. The difference between the versions is the host configuration, which causes them to have a mismatch in the host ids in relation to each other. The logic of the CRDT is shown in Algorithm 1, after receiving a registration broadcast message, a route information update will take place.

To create a session id, a logical clock is used. The id will become the highest of a previously received message or the system's clock plus one:

$$id = max(latest\_event, clock) + 1$$

If these are equal, the host id will be compared, to break the tie and see whether a route will be replaced. In case the id of the receiving host is higher than the incoming host id, the host id comparison will always result in no update.

Algorithm 1 NexoRouter CRDT Logic

Input broadcast message *message*
Output updated list *routes*

procedure NEXOROUTERCRDT
    Initiate Remote Registration
    $SessionId_{remote} \leftarrow$ timestamp of remote
    $SessionId_{local} \leftarrow$ timestamp of local session
    $HostId_{remote} \leftarrow$ ID of remote host
    $HostId_{local} \leftarrow$ ID of local host
    if $SessionId_{local} = SessionId_{remote}$ **then**
        if $HostId_{local} < HostId_{remote}$ **then**
            $routes \leftarrow routes \cup \{(message.id, message.ip)\}$
        end if
    else if $SessionId_{local} < SessionId_{remote}$ **then**
        $routes \leftarrow routes \cup \{(message.id, message.ip)\}$
    end if
end procedure

For the bug to manifest, the registration of a device has to occur at the same time on both hosts, meaning the clock times of these hosts are the same, which may not mean they receive the registration at the same time, due to potential clock drift. In the Nexo Router application this was possible by running a script, triggering simultaneous registration. This resulted in the same session id in most cases, however, some executions did not have such a timestamp due to the nature of the test environment. To better control this variable and get more deterministic results, we can modify the system through our fault injector, injecting the fixed clock fault from before, making sure the session id is always equal on both registrations. We run 106 experiments, with the bug manifesting in which 4 experiments resulting in an observable failure: a bug. The execution time of this campaign was 48 minutes, excluding analysis. Due to implementation inefficiency, post-execution analysis (using ElasticSearch data) took over an hour. Due to time limitations, this campaign was only executed with analysis once. Another execution showed similar behavior at run-time, but was not verified to be consistent between runs using post-analysis. The bug is unlikely to occur in practice because the first comparator is the session id, which is based on a millisecond timestamp and a single connecting device will typically not send two registration requests at the same time. This was presented to the developers and was confirmed as a bug. Some time after sharing the bug, a change was made to the application to prevent this bug from producing failure cases.

As an example and a proof of effectiveness, we can use our implementation to expose this flaw. This has been done by setting up a functional test that runs on nodes, and manipulating one of the application's code such that it can comply with the precondition.

Other undesirable behavior that was found during usage of the YAFI tool was a concurrency issue, where too many broadcast messages were sent and another mismatch in the expectations of the results.

Observation 4: Known bugs may be found by setting the preconditions and executing the fault injector.

5.1.5 Validation of Authentication

As an additional fault experiment in the targeted case, we inject a fault to validate an authentication module of the Nexo Router. We manually identify the method that should be called on which class and construct a failure plan. We construct a single failure plan, that we can execute as a standalone test case with the YAFI implementation. When executing this failure plan on the functional test that tests communication with a device, the experiment failed to trigger the fault. The hypothesis of running this plan was a clear inequality between the reference run data and the perturbed run data, we are testing an unhappy path. A difference in these data is expected because validation needs to take place successfully. Instead, when injecting an exception, this validation results in an exception being thrown and no successful validation. However, what we saw is equivalent output of the reference run and the perturbed run.

We look at why this could happen and the actual reason for this behavior. A cause that can be responsible for this behavior is proper error handling even in case of an injected exception, also known as a *resilient try-catch block*, as defined by ChaosMachine [73]. Another cause would be a bug in the system, where the authorization is not required for this part of the system. It turns out neither of these was the case. The actual cause was that the authorization was not used, but the logic was duplicated and integrated directly into the communication behavior. While this may be done on purpose and the fault that we injected should have been aimed at this part of the code, instead of the logic present in a dedicated validation class, this exposed another type of system flaw: duplicate code. This was confirmed to be unintentional and undesired duplication by developers.

5.2 Limitations

In this project, we were subject to limitations. While we had a proper case study project that was updated to work on an environment conducive to testing, it was hard to get the setup running as a multi-cluster setup. Taking into account the limited amount of time, the project was run on a single machine. As the application had to be run and tested on company equipment, no elevated privileges (root access) were available. This limited the available injection methods, for example system-wide network manipulation was excluded.

Another limitation with respect to the limited time is the setup of fault injection campaigns and how they have to be executed. Running this type of campaign takes a lot of time, due to the nature of running failure plans. When an experiment is run, the target application must start up correctly; this must be validated by the fault injection tool. Only then can the workload (functional test) be executed, faults injected, and cleanup is performed afterward. Finally, analysis needs to be performed on the collected data. This caused collection of the data set to be limited.

Design/Method	Tests Run	Exposed Flaw	Symptom	Expected Outcome
Untargeted, Automatic	172	Code Smell: Incorrect Parallel Execution	High Communication Count	OK
Targeted, Automatic	106	Bug	Incorrect Response	OK
Targeted, Manual	1	Code Smell: Duplicate Code	No Failure	500 Error
Total Tests Run: 279				

Table 5.1: Summary of the tests run and their outcomes.

5.3 Evaluation and Conclusion

In this section, we evaluate the framework and the results of the implementation. The case study provided valuable insight into the behavior of the application under various fault conditions. The experiments showed that the system was resilient in certain scenarios, and the injected faults were reflected in response metrics when feedback on the generation was used. This information is helpful in understanding the system's performance and identifying potential areas of improvement.

The case study also demonstrated the limitations of using back-end latency as a single response metric. While it proved to be useful for detecting network issues, this metric alone did not sufficiently determine the validity of specific operations. This emphasizes the need for additional response metrics to better evaluate the system's behavior under different fault conditions.

We have seen three cases of unexpected behavior in the system. In Table 5.1 we show the campaign information, with its exposed flaws and diagnostics. Of the three cases, one is a bug that we detected through careful tuning of the system setup. The experiment size of this targeted attack was 106 tests and the bug manifested in 4 of these. The execution time of this campaign was over two hours.

In the two other cases we examined, the exposed flaws were less severe. We found an incorrect execution in one case where we used a single test. This execution consists of incorrect parallelism, causing slight inefficiency in CPU usage. Regarding the test execution count, initially we ran 172 tests, of which 6 showed the symptoms of the flaw. Repeated runs showed different data, due to tracing framework configurations.

For the final case, where we identify duplicate code by distortion in the experiment, an unhappy path is tested. When an exception occurs (which we injected in this case), we *expect* the application to fail, as we can see in the "Expected Output" column. However, due to the wrongful duplication of code, the fault remained latent in the system. Repeated runs of this test always yielded the same result. This specific experiment was conducted as part of other campaigns and has been replicated 20 times.

Our observations that were made as a result of running fault injection campaign are now discussed. Observation 1 has been seen as part of running a small evaluation benchmark. Observation 2 shows the fidelity of the fault model we use and the outcome we observe. This

provides us with information towards answering RQ2.1. Another effect of the fault injector is reproducible results as Observation 3 mentions, system calls can be coordinated by the system across multiple nodes to assist the identification of potential issues and bugs. With reproducibility, many different faults can be injected on the same system state, generating useful information. In our case, a bug was detected, with the help of user-defined metrics, with Observation 4, the possibility of exposing or reproducing a bug is shown.

The case study highlights the importance of targeting critical areas of the system and meeting certain preconditions to increase the likelihood of finding bugs. This targeted approach could prove to be more effective in identifying problems within the system than untargeted exploration, however, to show this, more evaluation needs to be done on a set of known bugs. Moreover, the targeted approach requires advanced knowledge about the part where system failures have occurred.

For Adyen, we recommend focusing on exploring similar techniques to the targeted approach and to write analyzers that better detect incorrectness of experiment output. The Nexo Router has a critical role in the Adyen ecosystem and operates 24/7. In case of outages, either at data center level, or system level availability is important and needs to be tested. There are many faults that can occur in the system, with complicated preconditions, many of which are hard or impossible to replicate in the current test suite. It is also a good idea to execute several functional tests at a high level. Many of the superficial tests are useful to match expectations under different fault models with very low effort required compared to unit tests.

We answer RQ2 based on the campaign results. Table 5.1 shows the effectiveness of YAFI in numbers. 279 unique tests were executed as part of two campaigns, resulting in the identification of three system flaws in Adyen's Nexo Router module. An important result is the difference in detected flaws during the targeted and untargeted campaigns. During the execution of the untargeted campaign we have seen a form of distortion, where the actual effect of the injected faults does not represent expectations in all cases. This demonstrates an indirect capability of YAFI: the ability to assess expected system behavior. This is also used to find a flaw: duplicate code in a targeted test. Answers to RQ2 are also obtained from looking at the targeted exploration execution. With this type of campaign, we see valuable information for RQ2.2, specifically. We also see that there are many possibilities in determining output validity during execution. When testing unhappy paths, our analyzers need to be coordinated properly, and take into account the appropriate system metrics. This insight is important for RQ2.2.

Overall, we find YAFI to be an effective fault injection framework. It managed to help detect several flaws in the system with a minimal implementation of the components. With improvements made on oracles and the use of search strategies adopted from literature, the framework can reduce the need for writing traditional tests and save engineers a lot of testing time.

These results also answer our question about resilience in RQ1. Perturbation of the system was proven to be possible in this case study with real-world results showing that the system can be tolerant of faults and prevent failure, or end up in detectable, incorrect states, providing wrong or no responses during otherwise correctly functioning operations.

Chapter 6

Conclusions and Future Work

In this chapter, we discuss the implications of our work, identify potential future areas of improvement, and examine threats to validity. Finally, we present our conclusions based on the contributions and evaluation of the YAFI framework.

6.1 Discussion

6.1.1 Future work and Implications

In this section we discuss open questions and see how future work can make use of YAFI to answer these questions.

Our YAFI implementation is available to everyone in its public GitHub repository[1]. This repository will be updated in the future, with the goal of implementing the framework more faithfully. As the implementation was made during an experimental process, compromises were made, making it less user-friendly than desired. Future work can include custom implementations that are currently possible to be built by anyone. However, YAFI requires improvement in the load generator. Mainly, the workload has to be implemented in a formal way. This will require updating the complete faultload model, as the fault model and workload are closely related.

The presented architecture has a lot to offer, but is implemented with focus on the fault injection module. There is still a lot of evaluation that can be done on many search strategies and automated analysis of experiment data. These require new implementations of components like the practical example we have shown for NEAT integration in Section 4.4.

In terms of search strategy, there are many algorithms that can be explored. Many different strategies are present literature, including novel works like CoFI's consistency guided search as part of guided random testing. Similar methods can be evaluated and implemented as generators in YAFI. On top of these techniques, there is much room for improvement in run-time efficiency and effectiveness. One way this can be done is by pruning the output set of failure plans, when a plan is determined to be redundant. Currently, the only pruning done in YAFI is deduplication of the same faults.

[1] https://github.com/freshcoders/YAFI

For analysis, significant improvements can be made as well. Our methods only include simple assertions and comparisons between the reference run output and the perturbed run output. Using more advanced comparators in the analyzers not only allow for better detection of faults, but also improve automatic validation and bug detection. As a result, more observability metrics can be collected and compared as well as a larger set of failure plans can be tested without manual intervention. The use of oracles and checkers in YAFI and fault injection systems in general, can be explored and change the capability of these systems in automatic detection.

Future work that can yield results with little effort is the implementation of various techniques from existing work and combining these techniques to explore the effects. One example of search techniques that are not mutually exclusive are found in CoFI and Filibuster.

Practical value can be added to YAFI if a potential community builds reusable, generic components. Earlier works have relied on exporting metrics to Prometheus, visualizing through Grafana (and other dashboards). If one such generic component is built, it provides benefit to a whole community.

Finally, we have observed much literature presenting novel fault injection. They all try to evaluate their system, doing this using custom evaluation criteria and targets. There exist bug datasets on cloud systems [34, 48] that can be used to build a benchmark for fault injection systems. The ability to evaluate newly presented fault injection systems or techniques on a common data-set and environment is not well-defined at this time.

6.1.2 Threats to Validity

Internal Validity

1. The SUT was tested using a single functional test. Although it covers critical business logic and this research has resulted in positive feedback for Adyen, its effectiveness may be different when applied on a greater scale.

2. The experiments were carried out within a strict timeline and some campaigns were executed without replication. Without repeating experiments we cannot be certain of the detection rate, as non-determinism plays a role in manifestation of these bugs.

External Validity

1. Our framework is only evaluated on a single application type. In other fault injection research, arbitrary sets of target applications are part of evaluations, however, they have common applications. To properly evaluate YAFI, it should be executed on these common applications and with similar configurations, where possible.

2. Limited fault types are tested. In other fault injection systems, different faults are injected and target applications may benefit more from testing with faults other than network faults.

6.2 Conclusions

We presented YAFI, a platform-independent framework complete with an architecture and protocols to follow for implementation and extension. Over the past years, fault injection tools have emerged that implement a new framework from scratch. The results of these tools have been loosely based on seminal work in the context of fault injection. We have observed that recent frameworks use their own architecture to implement and evaluate their novel techniques. Our goal has been to create a framework that is usable on multiple platforms and is independent of environmental changes. This is achieved by building a modular framework with extensions as replaceable components. This allows portability across platforms and target applications. In addition, components that are built are easily transferred from the method they derive from, to a new method. As a result, our framework is capable of three major achievements. First, the first goal of this research is achieved, a platform-independent framework that can be applied to arbitrary targets, running on arbitrary operating systems, and showing resilience information answering RQ1:
What are the potential challenges in designing a platform-independent fault injection framework?

Second, YAFI can integrate earlier works, techniques, and support extensibility for new systems to be built on top of it. Third, any technique can be taken and combined with compatible components from other fault injection strategies. This offers unseen combinations that were previously unfeasible to evaluate.

Our evaluation of YAFI with a case study has shown promise. The implementation of fault injectors and simple analyzers as evaluation on the Nexo Router resulted in the discovery of multiple system flaws were not clear beforehand. This advocated for effectiveness and we have provided an answer for RQ2:
How well can the fault injection framework identify bugs in the system by using user-defined metrics?

Injecting faults with YAFI is made easy, however, gathering system information and assessing the correctness is still a challenging task. There are few limitations when using our framework, as each component is highly configurable, only enforcing input and output specifications according to YAFI protocols.

Bibliography

[1] Apache Cassandra — Apache Cassandra Documentation — cassandra.apache.org. https://cassandra.apache.org/_/index.html, . [Accessed 08-Jun-2023].

[2] Apache ZooKeeper — zookeeper.apache.org. https://zookeeper.apache.org/, . [Accessed 08-Jun-2023].

[3] Today's outage for several Google services — blog.google. https://blog.google/inside-google/company-announcements/todays-outage-for-several-google/. [Accessed 11-Jun-2023].

[4] A powerful chaos engineering platform for kubernetes: Chaos mesh®, 2023. URL https://chaos-mesh.org/.

[5] Joakim Aidemark, Jonny Vinter, Peter Folkesson, and Johan Karlsson. Goofi: Generic object-oriented fault injection tool. In *2001 International Conference on Dependable Systems and Networks*, pages 83–88. IEEE, 2001.

[6] Sundos Abdulameer Alazawi and Mohammed Najim Al-Salam. Review of dependability assessment of computing system with software fault-injection tools. *Journal of Southwest Jiaotong University*, 54(4), 2019.

[7] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. Toward a generic fault tolerance technique for partial network partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 351–368, 2020.

[8] Peter Alvaro, Joshua Rosen, and Joseph M Hellerstein. Lineage-driven fault injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 331–346, 2015.

[9] Miguel Amaral, Miguel L Pardal, Hugues Mercier, and Miguel Matos. Faultsee: Reproducible fault injection in distributed systems. In *2020 16th European Dependable Computing Conference (EDCC)*, pages 25–32. IEEE, 2020.

[10] Lörinc Antoni, Régis Leveugle, and M Feher. Using run-time reconfiguration for fault injection in hardware prototypes. In *17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002. Proceedings.*, pages 245–253. IEEE, 2002.

[11] Jean Arlat, Yves Crouzet, Johan Karlsson, Peter Folkesson, Emmerich Fuchs, and Günther H Leber. Comparison of physical and software-implemented fault injection techniques. *IEEE Transactions on computers*, 52(9):1115–1133, 2003.

[12] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of dependability. *Department of Computing Science Technical Report Series*, 2001.

[13] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[14] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[15] Ali Basiri, Niosha Behnam, Ruud De Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.

[16] Jossekin Beilharz, Philipp Wiesner, Arne Boockmeyer, Lukas Pirl, Dirk Friedenberger, Florian Brokhausen, Ilja Behnke, Andreas Polze, and Lauritz Thamsen. Continuously testing distributed iot systems: An overview of the state of the art. *arXiv preprint arXiv:2112.09580*, 2021.

[17] Luis Berrojo, Isabel González, Fulvio Corno, Matteo Sonza Reorda, Giovanni Squillero, Luis Entrena, and Celia Lopez. New techniques for speeding-up fault-injection campaigns. In *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 847–852. IEEE, 2002.

[18] Antonia Bertolino, Guglielmo De Angelis, Micael Gallego, Boni García, Francisco Gortázar, Francesca Lonetti, and Eda Marchetti. A systematic review on cloud testing. *ACM Computing Surveys (CSUR)*, 52(5):1–42, 2019.

[19] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. Cofi: consistency-guided fault injection for cloud systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 536–547, 2020.

[20] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics*, volume 10, pages 176–194, 2001.

[21] Mike Cohn. *Succeeding with agile: software development using Scrum*. Pearson Education, 2010.

[22] Carlos Colman-Meixner, Chris Develder, Massimo Tornatore, and Biswanath Mukherjee. A survey on resiliency techniques in cloud computing infrastructures and applications. *IEEE Communications Surveys & Tutorials*, 18(3):2244–2281, 2016.

[23] International Electrotechnical Commission et al. Functional safety of electrical/electronic/programmable electronic safety related systems. *IEC 61508*, 2000.

[24] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella, and Nematollah Bidokhti. Failviz: A tool for visualizing fault injection experiments in distributed systems. In *2019 15th European Dependable Computing Conference (EDCC)*, pages 145–148. IEEE, 2019.

[25] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, and Roberto Natella. Profipy: Programmable software fault injection as-a-service. In *2020 50th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 364–372. IEEE, 2020.

[26] Domenico Cotroneo, Luigi De Simone, and Roberto Natella. Thorfi: a novel approach for network fault injection as a service. *Journal of Network and Computer Applications*, 201:103334, 2022.

[27] Joao Duraes and Henrique Madeira. Generic faultloads based on software faults for dependability benchmarking. In *DSN*, volume 4, page 285, 2004.

[28] Luis Alfonso Entrena Arrontes, Celia López Ongil, Mario García Valderas, Marta Portela García, and Michael Nicolaidis. Hardware fault injection. 2011.

[29] International Federation for Information Processing. IFIP Working Group 10.4 — — dependability.org. `https://www.dependability.org/?page_id=265`. [Accessed 01-Jun-2023].

[30] Aakash Gangolli, Qusay H Mahmoud, and Akramul Azim. A systematic review of fault injection attacks on iot systems. *Electronics*, 11(13):2023, 2022.

[31] Giorgis Georgakoudis, Ignacio Laguna, Dimitrios S Nikolopoulos, and Martin Schulz. Refine: Realistic fault injection via compiler-based instrumentation for accuracy, portability and speed. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2017.

[32] Emanuel Giger, Martin Pinzger, and Harald C Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th working conference on mining software repositories*, pages 83–92, 2011.

[33] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. 2005.

[34] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM symposium on cloud computing*, pages 1–14, 2014.

[35] Ulf Gunneflo, Johan Karlsson, and Jan Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *1989 The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 340–341. IEEE Computer Society, 1989.

[36] Seungjae Han, Kang G Shin, and Harold A Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems. In *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*, pages 204–213. IEEE, 1995.

[37] John P Hayes. Fault modeling. *IEEE DESIGN TEST COMP.*, 2(2):88–95, 1985.

[38] Mike Hicks. Internet report: Pulse update – when automated ops goes bad (and other tales of cloud and data center woes). `https://www.thousandeyes.com/blog/internet-report-pulse-update-september-26-2022`, 2022. [Accessed 08-Jun-2023].

[39] Martin Hiller, Arshad Jhumka, and Neeraj Suri. An approach for analysing the propagation of data errors in software. In *2001 International Conference on Dependable Systems and Networks*, pages 161–170. IEEE, 2001.

[40] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[41] Nisrine Jafri. *Formal fault injection vulnerability detection in binaries: a software process and hardware validation*. PhD thesis, Université Rennes 1, 2019.

[42] Chong Hee Kim and Jean-Jacques Quisquater. Faults, injection methods, and fault attacks. *IEEE Design & Test of Computers*, 24(6):544–545, 2007.

[43] Guy Bolton King, Sean McCarthy, Pushkala Pattabhiraman, Jake Luciani, and Matt Fleming. Fallout: Distributed systems testing as a service. *arXiv preprint arXiv:2110.05543*, 2021.

[44] Kyle Kingsbury. Jepsen. URL `https://jepsen.io/`. Accessed on 12.11.2022.

[45] Maha Kooli and Giorgio Di Natale. A survey on simulation-based fault injection tools for complex systems. In *2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6. IEEE, 2014.

[46] Mariam Lahami and Moez Krichen. A survey on runtime testing of dynamically adaptable and distributed systems. *Software Quality Journal*, 29(2):555–593, 2021.

[47] Jean-Claude Laprie. Dependable computing and fault-tolerance. *Digest of Papers FTCS-15*, 10(2):124, 1985.

[48] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 517–530, 2016.

[49] Rakesh Kumar Lenka, Sarthak Padhi, and Kabita Manjari Nayak. Fault injection techniques-a brief review. In *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, pages 832–837. IEEE, 2018.

[50] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. *ACM SIGARCH Computer Architecture News*, 45(1):677–691, 2017.

[51] Ingrid Lunden. Gmail, YouTube, Google Docs and other services go down in multiple countries (Update: slowly coming back online) — techcrunch.com. https://techcrunch.com/2020/12/14/gmail-youtube-google-docs-and-other-services-go-down-simultaneously-in-multiple-countries/?guccounter=1&guce_referrer=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnLw&guce_referrer_sig=AQAAALoGEBxmn0w9XYUyqt7HSzmkyHpoyMNw7cxytn2jgodYRjIW2B25eHWE_g-2jVi4eMalobipxT4McOXyNuHRkcXavFAuhlwayOKnw0PTaQfY94v7XIqcj7pX-XX7KSeEPI4UQVele0Vs2cpZZs6SgTZjUODE4xYrCBvoBZBq90af. [Accessed 12-Jun-2023].

[52] Michael R Lyu. Software reliability engineering: A roadmap. In *Future of Software Engineering (FOSE'07)*, pages 153–170. IEEE, 2007.

[53] Henrique Madeira, Diamantino Costa, and Marco Vieira. On the emulation of software faults by software fault injection. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pages 417–426. IEEE, 2000.

[54] Timothy C May and Murray H Woods. A new physical mechanism for soft errors in dynamic memories. In *16th International Reliability Physics Symposium*, pages 33–40. IEEE, 1978.

[55] Christopher S Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. Service-level fault injection testing. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 388–402, 2021.

[56] Hamid Mushtaq, Zaid Al-Ars, and Koen Bertels. Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems. In *2011 IEEE 6th International Design and Test Workshop (IDT)*, pages 12–17. IEEE, 2011.

[57] Roberto Natella, Domenico Cotroneo, Joao A Duraes, and Henrique S Madeira. On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering*, 39(1):80–96, 2012.

[58] Roberto Natella, Domenico Cotroneo, and Henrique S Madeira. Assessing dependability with software fault injection: A survey. *ACM Computing Surveys (CSUR)*, 48 (3):1–55, 2016.

[59] Fotis Nikolaidis, Antony Chazapis, Manolis Marazakis, and Angelos Bilas. Frisbee: A suite for benchmarking systems recovery. In *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems*, pages 18–24, 2021.

[60] Strategic Planning. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, 1, 2002.

[61] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, Ruidong Tian, Chunlu Wang, and Gang Qu. Voltjockey: A new dynamic voltage scaling-based fault injection attack on intel sgx. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(6):1130–1143, 2020.

[62] Casey Rosenthal, Nora Jones, Lorin Hochstein, and Ali Basiri. Chaos engineering. *O'Reilly*, 2017.

[63] Amazon Web Services. Event-Driven Architecture — aws.amazon.com. `https://aws.amazon.com/event-driven-architecture/`. [Accessed 09-Jun-2023].

[64] Forrest Shull, Janice Singer, and Dag IK Sjøberg. *Guide to advanced empirical software engineering*. Springer, 2007.

[65] Jacopo Sini, Massimo Violante, and Fabrizio Tronci. A novel iso 26262-compliant test bench to assess the diagnostic coverage of software hardening techniques against digital components random hardware failures. *Electronics*, 11(6):901, 2022.

[66] Ningfang Song, Jiaomei Qin, Xiong Pan, and Yan Deng. Fault injection methodology and tools. In *Proceedings of 2011 International Conference on Electronics and Optoelectronics*, volume 1, pages V1–47. IEEE, 2011.

[67] David T Stott, Benjamin Floering, Daniel Burke, Zbigniew Kalbarczpk, and Ravishankar K Iyer. Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000*, pages 91–100. IEEE, 2000.

[68] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. Automatic reliability testing for cluster management controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 143–159, 2022.

[69] Haley Tucker, Lorin Hochstein, Nora Jones, Ali Basiri, and Casey Rosenthal. The business case for chaos engineering. *IEEE Cloud Computing*, 5(3):45–54, 2018.

[70] Erik van der Kouwe, Cristiano Giuffrida, and Andrew S Tanenbaum. Evaluating distortion in fault injection experiments. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 25–32. IEEE, 2014.

[71] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *OSDI*, volume 10, pages 2685048–2685068, 2014.

[72] Wei Yuan, Shan Lu, Hailong Sun, and Xudong Liu. How are distributed bugs diagnosed and fixed through system logs? *Information and Software Technology*, 119: 106234, 2020.

[73] Long Zhang, Brice Morin, Philipp Haller, Benoit Baudry, and Martin Monperrus. A chaos engineering system for live analysis and falsification of exception-handling in the jvm. *IEEE Transactions on Software Engineering*, 47(11):2534–2548, 2019.

[74] Long Zhang, Brice Morin, Benoit Baudry, and Martin Monperrus. Maximizing error injection realism for chaos engineering with system calls. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2695–2708, 2021.

[75] Haissam Ziade, Rafic A Ayoubi, Raoul Velazco, et al. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.*, 1(2):171–186, 2004.

Appendix A

Glossary

Table A.1 provides an overview of frequently used terms and abbreviations. If a definition includes a term defined in the table, it appears bold.

Table A.1: Glossary.

Term	Definition
System Under Test (SUT)	Refers to the software system being tested.
Experiment	A controlled test case conducted on the **SUT** to obtain information about its behavior.
Fault	A deviation from normal operation of any component of a system.
Failure	The observable incorrect behavior of a system due to a **fault**.
Bug	A manifestation of incorrect behavior in application code.
Fault Injector	Refers to the component of a **fault injection system** responsible for injecting faults into an **SUT**.
Fault Injection System	A system that can execute a full **fault injection campaign**, starting from the **fault model**.
Fault Injection Campaign	The execution of all **fault models** in the form of **experiments**.
Fault Model	A specification for a single fault injection experiment. It specifies all properties of the faults, allowing the **fault injector** to inject them at the appropriate time and location.
What-When-Where (WWW)	Refers to what fault should be injected (fault type), where it should be injected (location on the system), and when it should be injected.
Node	A computer system, as part of a distributed system.

Appendix B

Framework Specifications and Protocol Definitions

B.1 Fault Model

Protocol: fault_model

Property Name	Type	Description
hosts[].port	integer	The port number for the host
hosts[].ip	string	The IP address of the host
hosts[].uid	string	The unique identifier for the host
hosts[].tags	set of strings	A set of tags to filter sets of application types with
events[].fault.type	string	The type of the fault
events[].fault.config	string	The configuration for the fault
events[].fault.occurrence.timing	string	The timing of the fault occurrence
events[].fault.occurrence.location.info	map(string,string)	A set of key-value pairs, containing additional information for the target. Map contents may vary based on fault type.
events[].fault.occurrence.target	string	The target of the fault occurrence
events[].trigger.type	string	The type of the trigger
events[].trigger.arguments.time	string	The time argument for the trigger
events[].trigger.arguments.duration	string	The duration argument for the trigger
events[].trigger.target	string	The target of the trigger

Table B.1: Fault model protocol specification

B.2 Monitor Component

B.2.1 Collector

Protocol: collector input

Property Name	Type	Description	Required
application log	any	Collectors receive data from an external source, which may be of any type as they are built for collection from specific sources.	Yes

Table B.2: Collector input protocol specification

Protocol: collector output

The output data from both properties is combined into a single output type, at least one is required, but either can be omitted.

Property Name	Type	Description	Required
observability metrics	collector_out	Data compatible with the generator to build a set of data from data collected from experiments.	No
response data	collector_out	Data compatible with the generator to build a set of data from data collected from experiments.	No

Table B.3: Collector output protocol specification

B.2.2 Analyzer

Protocol: analyzer input

Property Name	Type	Description	Required
input data	collector_out	Analyzers process collector data, taking input compatible with a collectors output.	Yes

Table B.4: Analyzer input protocol specification

Protocol: analyzer output

Property Name	Type	Description	Required
result	bool	When using the analyzer as an assertion, the orchestrator can output the correctness of sets of analyzers and aggregate the boolean results.	Yes
custom_result	any	Analyzers may be used to provide additional warnings and feedback to the user, this may be done in any format.	No

Table B.5: Analyzer output protocol specification

B.3 Failure Plan Generator

Protocol: failure plan generator input

Property Name	Type	Description	Required
observability metrics	custom	A set of time series observed during an experiment	No
response data	string		No
fault type	string		No
collection results	collector_out	Data compatible with the generator to build a set of data from data collected from experiments.	No

Table B.6: Failure plan generator input protocol specification

Protocol: fault generator output

Property Name	Type	Description	Required
failure plan	fault_model	A representation of a fault model data structure. Any call to a fault generator may produce multiple failure plans.	Yes

Table B.7: Fault generator output protocol specification

B.4 Fault Injector

Protocol: fault injector input

Property Name	Type	Description	Required
fault	fault_model	Single faults, according to the fault_model protocol are given to the injector.	Yes

Table B.8: Fault injector input protocol specification

Protocol: fault injector output

Property Name	Type	Description	Required
message	fault_model	Fault properties are encoded in a message.	Yes

Table B.9: Fault injector output protocol specification

Protocol: proxy input

Property Name	Type	Description	Required
message	fault_model	Fault properties are decoded from a message.	Yes

Table B.10: Proxy input protocol specification

Protocol: proxy output

Property Name	Type	Description	Required
fault	fault_model	Proxies inject the fault eventually, regardless of their implementation.	Yes

Table B.11: Proxy output protocol specification

B.5 Orchestrator

Protocol: orchestrator

Property Name	Type	Description	Required
generations	list of fault_model	A list of representations of the fault model, preferably a sortable data type, for prioritizing plans	yes
generators	list of generators	All the enabled generators which will be used to generate failure plans (will be stored in generations)	yes
injectors	list of injectors	All the enabled injectors which will be used to execute failure plans. If an injector is not enabled, the fault in the failure plan will be ignored.	yes
collector	list of collectors		yes
analyzer	list of analyzers		no

Table B.12: Orchestrator protocol specification

Specification: orchestrator

The orchestrator runs in four phases. Each phase is subject to performing a set of operations, resulting in output. The phases and outputs are defined in table Table B.13

Phase	Dependencies	Output
execute reference run	load generator, collector	collector.observability_metrics monitor.custom_output response data
analyze data	failure plan generator, analyzer	failure plan
execute perturbed run	load generator, fault injector, collector	collector.observability_metrics collector.custom_output response data
finalize	analyzer	analyzer.result

Table B.13: Orchestrator phase specification

Table B.14: Targets of the System

Target	Description
all	All proxies matching the trigger filter are selected as the injection target.
any	A random proxy is selected at run time, based on a user-definable seed.
set	Only a pre-defined set is filtered, resulting in the injection target set. This includes a single proxy.
match_trigger	Only the proxy on which the trigger was activated is used as the injection target.
not_trigger	All but the triggering proxy are selected as the injection target.