Exploring Beyond the Happy Path

Practical Automated Network-Level Fault Injection Testing of Service-Oriented Distributed Systems

MSc Thesis

D.E. Flipse

TU Delft

ASML

Exploring Beyond the Happy Path

Practical Automated Network-Level Fault Injection Testing of Service-Oriented Distributed Systems

Msc Thesis

by

D.E. Flipse

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Tuesday July 22, 2025 at 13:00.

Student number:	4606116	
Project duration:	November 11, 2024 – July 22, 2025	
Thesis committee:	Dr. B. Kulahcioglu Ozkan,	TU Delft, Supervisor
	Dr. J. E. A. P. Decouchant,	TU Delft, Supervisor
	Prof. Dr. G. Smaragdakis,	TU Delft, Committee Member
	H. Simsek,	ASML, Company supervisor

Style: TU Delft Report Style, with modifications by Daan Zwaneveld

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

Organisations are increasingly adopting Microservice and Service-Oriented Architectures, moving from monolithic applications to (service-oriented) distributed systems. By their nature, distributed systems are prone to partial failures, where a subset of processes fail while others continue to operate. To assess the behaviour of the system when subjected to partial failures, we can inject faults that mimic partial failures. By automatically injecting all relevant combinations of faults, we can verify the system's resilience or uncover resilience bugs. However, the search space consisting of all combinations of faults grows exponentially. Moreover, it requires significant engineering effort to adjust systems to work with existing state-of-the-art tools. This work investigates a practical approach to automated fault injection for service-oriented distributed systems while remaining efficient. We design a novel automated fault injection tool that uses network-level instrumentation to inject faults. By using an abstraction to model the system's behaviour, we can dynamically infer information available to service-level instrumentation. Furthermore, we efficiently represent the search space as a search tree and prune it using multiple pruning policies. To evaluate our design, we developed an implementation of this design called Reynard. Compared to the current state-of-the-art, we show that Reynard is efficient, requiring equal or fewer states in the search space, has minimal overhead, and can be integrated into various existing benchmark systems with a feasible engineering effort. Furthermore, we demonstrate the applicability of Reynard by integrating it into an industry system, highlighting that it can run practical experiments and can identify real bugs. In conclusion, Reynard can provide a starting point for lowering the engineering effort required to perform automated fault injection testing, while increasing the confidence in the resiliency of systems.

Preface

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable." - Leslie Lamport

Dear reader,

Before you lies my master's thesis, the culmination of 8 months of hard work and the conclusion of my master's degree in Computer Science. It is also the end of 9 years at the TU Delft, a journey that started with a year of Electrical Engineering and led me to discover that my interest in programming and computer science was not merely a hobby, but a passion. I like to think that during these years, I have had the great pleasure and privilege of learning, experiencing and being part of just about everything one can do in a student career. But it wasn't until my Master's that I rediscovered my motivation to really *learn* a subject, instead of just doing what I needed so I could spend my time outside of lecture halls.

This project has been challenging, but I have had the fortune of finding a thesis topic I am extremely interested in, and which has allowed me to apply what I learned from various sources. It made the long days and evenings tinkering away feel more like a hobby project than work. It also provided me with the opportunity to delve deeply into the topic and read the fantastic work by Leslie Lamport, Peter Alvero, Christopher Meiklejohn, and Kyle Kingsbury, of whom the latter can be attributed to sparking my interest in testing distributed systems.

First and foremost, I would like to thank my supervisors for all their time and attention. To Burcu, for our lively but ever-hectic discussions, for helping me make sense of it all, and for all the constructive feedback. To Hakan, for connecting me to the right people, for helping me navigate the corporate maze, and for always making time in his ever-busy schedule. And to Jérémie, for always pointing out something that I had not thought about, but should.

I would also like to thank the people who have made this thesis project manageable. To Hans, Ole, Benji, and Peter, for being great study buddies, for always finding time for another coffee break, and for being my go-to "rubber duck" when I needed one. Never in my life have I drawn this many circles and arrows. To Davis, for the team effort, as our parallel projects often ran into similar challenges at different times. It was a great experience working with you. To my friends and roommates, for allowing me to vent my frustrations when I needed to, and for celebrating the small victories. To my family, for supporting me unconditionally, and for taking the time to let me explain an abstract computer science problem in *Jip-and-Janneke taal*. I fondly remember explaining my thesis subject to my grandmother, who has never owned a computer or smartphone.

Lastly, I would like to hint as to why I named the tool "Reynard". The name is based on the medieval Dutch stories of "Reinaert de vos", a mischievous anthropomorphic fox and his allegorical adventures deceiving other anthropomorphic creatures such as "Koning Nobel" (a lion) and "Cuwaert" (a hare, which Reinaert ends up eating). Our Reynard deceives the system into believing that faults are occurring. But as to the exact reasons, I'll leave that an exercise to the reader to figure out.

And finally, thank *you* for taking the time to read this work. I hope you learn something new, as I did when I wrote it.

D.E. Flipse
Delft, July 2025

Contents

List of Figures

List of Tables

Acronyms

OTel OpenTelemetry. 60, 62

RPC Remote Procedure Call. 3, 4, 25, 26

SDLC Software Development Life Cycle. 64
SUT (Distributed) System under Test. v, 8, 12, 13, 16–20, 23–26, 28, 36, 37, 40, 42, 45, 58, 65, 66

Terms

Fault Space The search space that is composed of all possible combinations of faults. 2, 28–30, 32, 39, 40, 43, 44, 46–50, 56–62, 64, 67, 68, 70

Faultload A combination of faults injected in the system. 19, 20, 23, 29–38, 40, 41, 44, 49, 55, 57, 60, 61, 64, 65, 67, 68, 71

Happy Path The functioning of the system in the absence of (partial) failures. 18–20, 28, 34, 37, 38, 40, 47, 58

Idempotent An operation that can be applied multiple times such that the outcome is the same as a single operation. 11, 40, 52, 64

Perturb Alter the normal or regular state or path. 12, 18–22, 67

Temporal Relating to time. 10, 24–26, 31, 32, 34

Transient Lasting for a short duration; temporary. 9–11, 35, 38, 48, 51, 64, 67, 68

Transitive Indirectly related. For example, if A relates to B, and B relates to C, then A is transitively related to C. 23, 24, 34, 37, 47

1 Introduction

Motivation

As the world becomes increasingly digital, modern software systems must keep up with growing demands in scale and complexity. The internet has made systems available to billions of users worldwide. To handle this unprecedented scale, systems must process vast volumes of data and perform intensive computations. At the same time, we have become reliant on our software systems. Services such as instant messaging, social media and web search have become so integral in our daily life that outages can significantly disrupt them [1]. To meet the scale and quality required by modern systems, developers must be able to build systems that can handle significant workloads while remaining available to their users at all times.

To meet these demands, modern systems are increasingly designed as distributed systems: software that runs on physically separated machines and cooperatively provides functionality [2]. A distributed system has access to resources that exceed the capacity of a single machine, enabling it to meet its computational and data scaling requirements. Additionally, it allows the system to provide redundancy by preventing a single point of failure. And for some systems, such as messaging, banking, and the internet, a physical distribution is a requirement [3]–[6].

However, distributed systems come with inherent challenges, of which a defining challenge is partial failures [5], [7]. A partial failure occurs when a part of the system fails, but the remainder is functional. In distributed systems, partial failures are a common but unpredictable reality [8]–[11]. They are both a risk and a benefit of distributed systems. Redundancy allows the system to remain functional in the case of partial failure. But, unexpected partial failures can cause the system to behave incorrectly in subtle and complex ways [8]–[11]. Furthermore, partial failures can cascade if incorrectly handled, potentially compromising the entire system [12].

Because partial failures are an inherent risk in distributed systems, developers must make deliberate choices about handling them. By simulating partial failures, it is possible to deterministically evaluate the system's behaviour when exposed to partial failures. Fault injection is a common test strategy used in both software and hardware testing.

By systematically exposing the system to all feasible combinations of partial failures, it is possible to verify its resiliency. However, due to combinatorics, the search space grows exponentially. In large-scale systems, this can make exhaustive testing intractable. In the literature, various techniques are proposed to reduce the search space to make it tractable to exhaustively test all distinct behaviours of the system [13]–[16].

These solutions are effective but impractical for real-world systems, as they require significant instrumentation effort. They require the whole system to be rewritten in a specific programming language [15], or provide only a proof-of-concept for a particular language and framework [16], [17]. As such, this work addresses how automated fault injection for distributed systems can be made practical while still being tractable.

Objective

This research has the following objective:

Improve the practicality of automated fault injection testing of Service-Oriented Distributed Systems.

By improving the practicality of automated fault injection, we intend to lower the barrier of entry for practitioners who want to apply the technique. Furthermore, we intend to make the technique applicable to a broader range of systems. This is beneficial, as verifying a system's resilience helps increase its reliability.

We specifically focus on Service-Oriented Distributed Systems, which form the basis of numerous industry systems and have properties different from those of other distributed systems. In Chapter 2, we define this class of distributed systems.

Research Questions

To reach this objective, this work answers the following questions and subquestions:

(RQ1) How can we practically assess the resilience of a Service-Oriented Distributed System using automated fault injection testing?

- (RQ1.1) What type of instrumentation can enable fault injection in Service-Oriented Distributed Systems with a reasonable integration effort?
- (RQ1.2) What technique can identify the causal relation between interactions?
- (RQ1.3) Can we uniquely and consistently identify equivalent interactions between services in the system?
- (RQ1.4) Which representation allows us to enumerate the search space efficiently?
- (RQ1.5) Can we model the influence of faults to reason about the expected system behaviour and the equivalence of combinations of faults?

We answer this question by designing and implementing an automated network-level fault injection testing tool called Reynard. This motivates our second research question:

(RQ2) How effective and efficient is it to assess the resilience of a Service-Oriented Distributed System using automated network-level fault injection testing?

- (RQ2.1) How does Reynard compare to the state-of-the-art in terms of runtime efficiency and search space reduction?
- (RQ2.2) How does the structure of a system's interactions influence the reduction of the search space?
- (RQ2.3) How much overhead does the instrumentation of Reynard introduce?

Finally, we are interested in the industry applicability of such a tool. This motivates our third question:

(RQ3) How effective is Reynard when applied to an industry system?

Contributions

This work makes the following contributions:

- An abstract formulation of the fault space, and a formulation of pruning policies in this abstraction.
- A model of the effects of partial failures on the behaviour of a Service-Oriented Distributed System.
- A design and implementation of a novel, practical, automated network-level fault injection testing tool called Reynard. The tool provides low-overhead instrumentation capable of injecting precise and isolated partial failures. Additionally, it includes a testing library that utilises the instrumentation to exhaustively and efficiently cover the fault space and validate correctness properties of the system under test. The tool is publicly available on Github [18].
- An evaluation of the effectiveness of Reynard, compared to a state-of-the-art technique.
- A demonstration and evaluation of the applicability of this testing strategy by applying Reynard and its testing library to benchmark systems and an industrial system.

2

Service-Oriented Distributed Systems

This work addresses a problem inherent in distributed systems. This chapter defines distributed systems and their inherent characteristics. Furthermore, we classify distributed systems to pinpoint our targeted subclass of distributed systems.

2.1. Definition of Distributed Systems

There is no consensus on the exact definition of a distributed system. Different works have different definitions [2]–[4], [6], [19]. Lamport gives a generic definition in his 1978 paper: "A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages." [2]. This definition covers most, but not all, aspects of what is now considered a distributed system. Similar to [19], we consider a distributed system as having the following properties:

- **Physically separate processes -** The system consists of a collection of computational processes called "nodes" (software running on some machine) that are, at least partially, physically separate (i.e., running on different machines). In modern distributed systems, the collection can vary in size throughout the system's runtime. This is often done in response to fluctuations in the number of user interactions with the system. For a system to be considered a distributed system, the number of nodes must be greater than one. It is possible for multiple nodes to run on the same machine, especially in the era of virtualisation and containerization. However, to differentiate distributed systems from concurrent systems, each node should have a non-shared memory space [7], [19]. Note that a distributed system can consist of nodes that run concurrent software.
- **Working cooperatively -** Different from a random collection of software, the nodes should cooperate to achieve a common goal [3]. Combined, the nodes form a coherent system.
- **Exchanging messages -** As the nodes run on (potentially) different machines, they must communicate through a network connection rather than direct hardware connections. Communication is done using some form of message passing or streaming.

So, rewriting Lamport's definition, a distributed system can be defined as *a collection of cooperative but physically separated processes that communicate by exchanging messages*.

2.2. Fundamental Characteristics of Distributed Systems

We can attribute specific characteristics to a distributed system based on its definition. The most influential work defining these characteristics is the technical report by Waldo et al. [7] from 1994 at Sun Microsystems. Waldo et al. identified four key differences between distributed systems and their local counterparts.

Specifically, it addresses the issues that arise when replacing local function calls with calls to functions on another node, known as a Remote Procedure Call (RPC). Each of these characteristics should be

taken into account when developing distributed systems, as they can lead to inefficiencies or unexpected behaviours. In the following sections, we discuss each of these properties.

2.2.1. Increased Latency

As the nodes in a distributed system communicate using network connections, as opposed to hardware connections, there is increased latency in communication. Modern internet connections allow messages to pass in the order of milliseconds. Hardware connections can communicate in the order of nanoseconds or microseconds, depending on the type of communication channel. As such, developers need to account for an increase of several orders of magnitude in the latency for each message. Simply replacing function calls with remote procedure calls leads to significant delays.

Although latency can be challenging, it is a challenge that, in most systems, can be overcome. If the number of messages passed between separate nodes is kept to a minimum, so is its impact. Additionally, one can physically place nodes closer to the end-user to reduce latency. This solution is applied by, for example, a Content Delivery Network (CDN).

2.2.2. No Shared Memory

Compared to local systems, nodes do not have access to the same memory address space. While local functions can be called with memory pointers as arguments, RPCs must pass along (a copy of) the referenced memory space. This can drastically increase the overhead of RPCs.

Although the impact can be minimised, it forces developers to see RPCs as different from their local counterparts. Especially in the case of consistency, if the same object is shared and stored on separate nodes, they can become inconsistent.

2.2.3. Concurrency

As the computations for the system are done on physically separate machines, there is no fixed control over the order in which operations are performed [7]. This implies that any two unrelated operations can happen in any order. As Lamport details, operations in distributed systems only define a partial order [2]. This implies that any execution of a distributed system corresponds to *at least* one totally ordered execution. As such, operations in a distributed system can be truly concurrent and therefore nondeterministic.

Developers must account for this to ensure their system behaves as expected, even in the face of scheduling nondeterminism. Unexpected behaviours in distributed systems often stem from unexpected orderings of events [9], [20]. The challenge is that accounting for all possible orders of events is intractable, due to the combinatorial explosion. Addressing this challenge is its own research field, with many works focusing on model checking and formal methods to verify correct behaviour for all possible interleavings of events [21]–[24].

Although concurrency bugs can cause inconsistencies or crashes in distributed systems, they are not the primary focus of this research. The research is related, and works in this area are tangential to this work.

2.2.4. Partial Failures

Distributed systems are uniquely prone to partial failures. A local software system can either function normally, hang or crash in its entirety. However, in a distributed system, any node can be prone to hanging or crashing, while the other nodes continue to operate normally. As Lamport's famous quote regarding distributed systems goes: "A distributed system is one in which the failure of a computer you did not even know existed can render your own computer unusable.". Partial failures differentiate concurrent from distributed systems [3], [7].

To complicate matters, the communication link between nodes (i.e., the network) can also malfunction. Messages can be delayed, dropped or altered. This can happen for both the incoming and outgoing messages of a node. Distinguishing between an unresponsive node or a failing network link is impossible in the general case [3], [7].

Partial failures are more common than rare [3], [4], [7]. As Waldo et al. state: "Partial failure is a central reality of distributed computing" [7]. There are multiple reasons why this is the case. Hardware faults

will occur over time [25], software can contain bugs [9], and the network is unreliable [26]. Therefore, it is better to assume that partial failures (either for nodes or network links) will occur. To further quote Waldo et al. [7]: "A central problem in distributed computing is ensuring that the state of the whole system is consistent after such a failure; this is a problem that simply does not occur in local computing."

This is the characteristic of distributed systems that this research focuses on: understanding how a system behaves in the presence of partial failures to assess its resilience to partial failures. However, to reduce the scope, we focus on a specific class of distributed systems. We define this class in the next section.

2.3. Classifications of Distributed Systems

The abstract definition of a distributed system, as presented in Section 2.1, covers a wide variety of systems. There are, however, classes of distributed systems which are more similar. The following sections examine various classifications. Based on these classifications, we define Service-Oriented Distributed Systems, the class of distributed systems that this work focuses on.

2.3.1. Inherent Distribution or Distribution as an Artifact

In the book by Cachin et al., the main distinction is made between distributed systems that are inherently distributed and those that use distribution as a solution to a problem ("as an artifact") [3].

Inherently distributed systems require a physical separation between nodes to function properly. For example, notification or broadcast systems use the physical location of different devices to allow communication between users that would otherwise be physically impossible. Similarly, the Internet is a communication system that is inherently distributed. Furthermore, any IoT solutions that rely on the physical location of sensors for the system to function are inherently distributed.

On the other hand, distribution as an artifact tries to solve an engineering requirement. For example, when a system requires more resources than a single physical machine can provide, multiple machines and cooperative software can address this challenge. Furthermore, if a system needs to continue functioning even if a single machine fails, a fallback to another machine can be implemented. Effectively, the distinction is that if the system would work if it were bug-free and could run on a single hypothetical machine with unlimited resources and perfect hardware, then it is distributed as an artifact.

2.3.2. Distributed or Decentralised Systems

The more recent work by Steen and Tanenbaum calls these inherently distributed systems "decentralised systems" [4]. Their distinction primarily focuses on whether systems resources are either *sufficiently* (as an artifact) or *necessary* (inherently) distributed.

Computational, Informational or Pervasive

Steen and Tanenbaum further categorize sufficiently distributed systems as either "computational", "informational" or "pervasive" distributed systems [4]. These are defined as follows:

- *Distributed computational systems* use shared resources to increase computational throughput. Examples are high-performance, grid and cluster computing systems. In this class of distributed systems, a large computational task is subdivided into smaller tasks. If a partial failure is detected, it is often masked by repeating the failed smaller tasks.
- *Distributed informational systems* connect multiple distinct information systems to compose a more complex information system. This class of distributed systems can be found in numerous organisations, where different software (from different departments) must cooperate to fulfil a business need. Examples are client-server (web) applications, database management systems and numerous enterprise systems. In this class of distributed systems, the functional capabilities and data are distributed to different applications.

  In the case of partial failures, various resiliency techniques exist. The most common is the use of fallbacks. To prevent impeding a functional capability, redundancy is used to have different nodes capable of performing the same functionality. To prevent data unavailability, replication is used so that multiple nodes can provide the same data. If no fallback is possible, systems can employ graceful degradation to provide functionality at a lower quality (e.g., slower, less accurate).

- *Pervasive Systems* are composed of nodes that blend into a physical space. Examples are mobile telephones and IoT devices. These systems often have a greater variety in their topology compared to other systems. Furthermore, their network and nodes are often more likely to fail, as they are not part of a controlled environment such as a data center.

This work focusses primarily on distributed (as an artifact) informational systems. This system class is commonly used in the software industry [27], [28]. Specifically, we are interested in service-oriented, distributed information systems, as discussed in the following section.

2.3.3. Service-Oriented Distributed Systems

In the industry, there is a steady trend toward developing systems in a Service-Oriented Architecture (SOA) or Microservice Architecture (MA) as opposed to a monolithic design [27], [29]. In essence, a SOA is a distributed information system that is distributed as an artifact. Specifically, it distributes the system by decomposing the functionality into distinct domains, where each node provides functionality for a specific domain (i.e., offers a service). This contrasts with other distributed systems, which tend to distribute resources or data, but where each node has the same or similar functionality. For example, in most distributed databases, the nodes are often instances of the same software.

The Microservice Architecture (MA) is an especially popular approach to system design within industrial systems [27], [28]. It is similar to the SOA, but decomposes each service into even smaller (hence micro) units of work. The main proposed benefit of the Microservice Architecture is the ability for teams to work independently on the system [30]. It enables organisations to work on large, complex systems while keeping the cognitive load for teams manageable. It is, therefore, mostly an organisational approach to software, rather than an engineering solution. The popularity of the Microservice Architecture has resulted in many organisations moving away from monolithic design, resulting in more distributed systems. We refer to systems built in a SOA or MA as Service-Oriented Distributed Systems.

There is a lack of representative publicly available Service-Oriented Distributed System [31], [32]. Industry examples often involve systems developed by hundreds of full-time employees, feature domain-specific functionality and, therefore, are proprietary and a company secret. Benchmark systems exist, but these are not representative of real industry systems as they are more “idealised” than their less-ideal industry counterparts [32], [33]. To address this, we apply our experiments to benchmark *and* real industry systems. This way, our results more accurately reflect the applicability in real systems.

In addition to the characteristics of distributed information systems, this class of systems has its own unique characteristics that must be taken into account. These are detailed in the following sections.

Polyglot

As each service is a separately functioning program, it can be written in a programming language that best suits the service’s requirements. In theory, one could make every node in a different programming language. In practice, organisations tend to limit the number of languages used, but it is common to see a small number of different programming languages used to build the system. The same applies to libraries used within the same programming languages, as these can differ based on the service’s needs.

As a result, it is challenging to enforce a singular approach to tackling the inherent challenges of distributed systems. For example, one language might have a well-maintained library for common partial failure resilience practices. Yet, this might not be true in other languages.

Therefore, any solution that addresses partial failures in Service-Oriented Distributed Systems should be either independent of the programming language and libraries used in each node or, at most, require a minimal change to the existing source code.

Loosely Coupled and High Cohesion

Services in a Service-Oriented Distributed System are intended to be independent and loosely coupled. This means that nodes should not require the availability of other services to function. As such, it is important to verify whether a node can withstand the failure of its dependencies.

Furthermore, nodes should be highly cohesive. Most of its functionality is encapsulated within the service, thus requiring only a few dependent services. This is beneficial, as each network dependencies introduce latency. The dependency graph should not form a clique; it should be a sparse graph.

Emergent Behaviour

As Service-Oriented Distributed Systems are decomposed by domain, each domain functionality can be tested by testing the service. However, the complete application provides functionality defined by the composition of the different services. This can result in emergent behaviour: behaviour that only occurs by the interaction of different components.

Service-Oriented Distributed Systems can have undesirable emergent behaviour. The complex interactions between services, in combination with partial failures and concurrency, can lead to unexpected results. To test the resiliency of a system, it cannot be tested by validating each service. The composition of fault-tolerant components does not make for a fault-tolerant system [34]. Therefore, we must test the resilience and system behaviour using the composition of all relevant components.

Stateless and Stateful Services

A Service-Oriented Distributed Systems is usually composed of stateless (micro) services and stateful datastores such as databases and message queues. Stateless services enable the system to scale its throughput by allowing multiple instances of the service to run in parallel, dividing the incoming requests among these services.

Because services are stateless, their partial failure is less impactful, as they do not lose data. Furthermore, we can mask their failure by falling back to an identical copy.

However, even if they do not store data, a partial failure can still cause issues. A partial failure can interrupt the normal flow of operations, leading the system to a potentially incorrect state. Therefore, we must verify their functional correctness in the presence of partial failures.

High Frequency of Deployments

A key reason for adopting the Microservice Architecture is to enable teams to deploy their service independently. This results in a high deployment rate. In turn, this limits the time that the current "version" of the system can be tested for.

Therefore, testing the resilience to partial failures must be done within a time window bounded by the (short) time between deployments. This window can be significantly shorter in large organisations, where the system can contain a considerably larger number of services. As such, a goal must be to minimise the time needed to analyse the system's resilience.

3

Functional Testing under Partial Failures in Service-Oriented Distributed Systems

Distributed systems have the inherent challenge of being prone to nondeterministic partial failures, as detailed in Chapter 2. In this chapter, we provide an overview of the relevant types of partial failures that a Service-Oriented Distributed System can be exposed to, their effect, and how they can be simulated. Additionally, we explore different techniques to verify the correct behaviour of a (Distributed) System under Test (SUT) in the presence of partial failures. We highlight why automated fault injection testing is an effective technique for verifying correctness properties of a SUT.

3.1. Partial Failures Types in Service-Oriented Distributed Systems

This section describes types of partial failures and their effect on the behaviour of a Service-Oriented Distributed System. Each type of failure can be caused by various "failure modes", i.e., the specific causes of a failure. Understanding the failure modes provides insight into the likelihood of each type occurring. More importantly, understanding the effect of each failure type is crucial for accurately simulating partial failures.

Several classifications exist for the types of failures in distributed systems. In the book by van Steen and Tanenbaum [4], a classification of failure types is given, as presented in Table 3.1. The book by Cachin et al. [3] proposes different types of process failures, with a focus on distributed algorithms. Their classification of failure types includes eavesdropping and crashes with recovery failures, but omits timing and response failures. We follow the classification by van Steen and Tanenbaum, as we find their classification to best match failures in Service-Oriented Distributed Systems.

Table 3.1: Different types of failures [4].

Type of failure	**Description of server's behaviour**
Crash failure	Halts, but is working correctly until it halts
Omission failure	Fails to respond to incoming requests
Receive omission	Fails to receive incoming messages
Send omission	Fails to send messages
Timing failure	Response lies outside a specified time interval
Response failure	Response is incorrect
Value failure	The value of the response is wrong
State-transition failure	Deviates from the correct flow of control
Arbitrary failure	May produce arbitrary responses at arbitrary times

In Service-Oriented Distributed Systems, not all of these failure types are relevant. The aforementioned classifications cover failure types that can occur in all classes of distributed systems. Most notably, eavesdropping and arbitrary failures are only relevant if untrusted nodes are part of the system. In Service-Oriented Distributed Systems, all nodes are managed by a single organisation. As such, we do not consider them relevant. In the following sections, we describe each relevant failure type and to what extent they are included in our work. The class of crash-recovery failures, as described by Cachin et al. [3], is regarded as a subclass of crash failures.

3.1.1. Crash Failures

In a crash failure, a node operates as expected until it suddenly ceases to function. Formally, it implies that the process ceases to function permanently. Numerous causes can result in a crash failure in a single node. Potential causes include:

- **Hardware Failures** - Most software requires correctly functioning hardware. Hence, a common cause of software crash failures is hardware failure [35]. In data centers, disk and memory failures are everyday occurrences [25]. In an empirical study on bugs in cloud systems, Gunawi et al. found hardware failures to be 13% of causes for system failures [9].
- **Resource Failures** - Hardware resources can fail both physically and functionally. Aside from being imperfect, hardware is often limited in capacity. Limited disk or memory space can cause crashes when exceeded. Additionally, interactions with the resource can fail. For example, incorrect file permissions can cause failures [36].
- **Invalid Input** - Faults can occur when a node is presented with input that it cannot process. This is a common cause of failures in local programs. This type of failure is addressed in the tangential field of fuzz testing, where the input space is searched to find potential program failures [37].

  Within the scope of this work, it is assumed that invalid inputs can cause the system to crash; however, we do not focus on identifying which specific inputs are responsible for this.
- **Incorrect Error Handling** - Within an application, the previously mentioned failures often result in exceptions and errors. In the ideal case, these errors can be resolved. However, in some cases, it can result in crashes. For example, when a hard disk is corrupt, a database cannot function normally. In other cases, these errors can be observed in the interactions with the process.

  As various empirical studies demonstrate, it is often simple error-handling bugs that cause outages [8], [9]. It can be infeasible to know all possible errors that a component or node can return (which includes the operating system) [38]. In most cases, the system is unaware of the errors it can return, which makes the documentation incomplete. Unexpected errors without proper error handling can lead to unpredictable behaviour, including crashes.

In general, a crash partial failure is highly likely to occur in a Service-Oriented Distributed System during its runtime. Hardware and resource failures are nondeterministic and can happen accidentally. Functional errors are unintentional but can be latent, and every new version of software can introduce new bugs. The combination makes crash failures likely to happen, but unpredictable.

When a crash failure occurs, it can only indirectly impact other services. For example, other services cannot communicate with a crashed node. To investigate the effect of a crashed node on other nodes, we can simulate crash failures by either directly crashing nodes or by replicating the effects. The overhead of crashing nodes is significantly higher than simulating their effects.

Crash-Recovery Failures

If a crash failure can be detected, we can attempt to recover the failing node. Instead of a permanent failure, it can become a transient (temporary) failure. This recovery process can range from a simple restart to a more complex recovery routine. Typically, this recovery process is managed by a node that monitors the status of each node, detects failures, and restarts it as needed.

A recovery “masks” (hides) the failure of a node. While the node is failing, but not yet detected, other nodes will still experience the effects of a crash failure. It is therefore beneficial to test for crash failures even if recovery is possible.

Additionally, the following challenges should be considered for crash-recovery failures:

- **Faults can be transient**. At any time, when interacting with another node, the node can crash *or* recover. A simulation of this failure type should be able to simulate the temporal aspect of failures.
- **The recovery process can be faulty**. As Gao et al. [39] noted, the recovery process is often not as well-tested. When a crash-recovery failure is simulated by its effect, then the recovery process is not covered, as this is not triggered.
- **Recovered nodes can have amnesia**. If a crash failure occurs while processing a message, but before a state transition is made persistent, then a node can "forget" about a message when recovered. This, too, is not covered by simulating the effects of a crash failure. However, in Service-Oriented Distributed Systems, a node is expected to be stateless. Hence, for this effect, there should be little difference between an actual crash and a simulated crash.
- **Failure detection can be inaccurate**. This means that functioning nodes can be falsely detected as failing and unnecessarily recovered. This can cause emergent behaviour, where the recovery process renders nodes unavailable. In the next section, we address the challenge of failure detection.

As Service-Oriented Distributed Systems value high-availability, their runtimes are often configured with health checks and automatic recovery. As such, crash-recovery failures are highly relevant and should be considered for Service-Oriented Distributed Systems.

Failure Detection

If it is possible to detect a failing node, we can account for it. But, failure detection in itself is a distributed algorithm, which means that it too is prone to the same maladies that affect distributed nodes. In the work of Cachin et al., different types of failure detection are distinguished for crash failures [3]. These are:

1. **Fail-stop** - All other nodes can perfectly detect the failure of the crashed node.
2. **Fail-noisy** - A subset of the nodes might (eventually) detect that the node crashed.
3. **Fail-silent** - There is no way to detect if the node crashed.

For Service-Oriented Distributed Systems, it is common to assume a fail-noisy effect. The fail-stop paradigm is unrealistic for real applications, as the network is unreliable [26] and no distinction can be made between a crashed node and a failing network [40]. Fail-noisy is hard to guarantee in the case of truly asynchronous communication, as it is impossible to know if a node crashed if it can respond in an unbounded amount of time. Fail-silent systems are impractical to reason about, as systems cannot afford to wait for undefined amounts of time. For most systems, it is most realistic and practical to assume that crashes are fail-noisy and assume asynchronous communication with an upper bound (partially synchronous communication). That is, if a node does not respond in a reasonable amount of time, we assume it to be crashed. As a result, this can lead to false positives.

3.1.2. Omission Failures

Messages are not guaranteed to be sent or received. Distributed systems communicate over a network, rather than through hardware connections. But, communicating over the network is unreliable [26]. As a result, messages may fail to be sent or received.

Send and receive omissions have different effects on the system's behaviour. In the case of a send omission, the sending node can be aware that it failed to send the message and act accordingly. In the case of a receive omission, neither party to the exchange might be aware that a message was not sent. In that case, it is more complex to derive a correct algorithm, especially in asynchronous communication, where the distinction between a receive omission and a crash failure upon receival is indistinguishable.

In most Service-Oriented Distributed Systems, communication is commonly done in a request and response pattern. In this type of communication, both the request and the response can be omitted due to a send or receive omission. A request omission results in a specific behaviour in the upstream (sending) node, but the downstream (receiving) node is unaffected, as it has never processed the message. However, in the case of a response omission, the downstream node processes the message, but the upstream node cannot be informed of this. In this case, the node can have inadvertently changed its state (and potentially that of other nodes). This latter case is more challenging to be resilient to.

Although response omissions can occur, most works do not consider them due to their low likelihood and higher complexity [16], [41]. To reduce the scope of this work, we also do not consider them as part of our testing strategy.

3.1.3. Timing Failures

A network is both unreliable and inconsistent [26]. The same is true for the nodes that need to process the messages. Nodes can hang during the processing of a message for various causes. Any problems arising from the timing of sending, receiving and processing of messages can be categorised as timing failures. Timing failures can result in the following effects:

- **Scheduling nondeterminism** - Timing inconsistencies can cause reordering of (concurrent) messages. For example, if two messages are sent directly after each other, the network does not guarantee that they will be delivered in that order. Although this is a source of bugs in distributed systems [9], we do not consider these types of failures as they are a result of concurrency issues, not partial failures.
- **Timeout bugs** - A common crash failure detection method is the use of timeouts [42]. However, timeouts can also be a source of bugs in distributed systems [43]–[45]. Finding the proper timeout bound can be a challenging task. If it is too lenient, it is not effective. But if it is set too strictly, it can lead to false positives. As we discussed, failure detection is imperfect in Service-Oriented Distributed Systems.

  If a downstream node is responsive, but the timeout bounds are set too strictly, the response might be discarded as the timeout is reached. As a result, the upstream node can incorrectly assume that the downstream node crashed. In reality, it is now a response omission failure, which warrants a different resolution.

  Timeouts in combination with retries can cause issues. Retries are a resilience pattern based on the idea that failures tend to be transient. There is a good chance that the issue will be resolved if it is tried again. However, in the case of a misconfigured timeout, a retry can cause a violation of the system's expected behaviour [38]. For example, consider a system that acts as a distributed queue, and we send a request to dequeue one item. If this message is received, processed, and a response is sent, but the combination of these actions takes longer than the set timeout value, we might consider it a crash failure. If we attempt to resolve it by retrying the request, we can inadvertently dequeue two items. A common solution to this challenge is to ensure that requests are only retried on idempotent interfaces. I.e., if the message is received a second time, the resulting state of the other node is equal.

Timing failures are impractical to simulate if there is no internal control over the clock of the nodes. Without control, we must wait the whole duration of the timeout bound, which is usually in the order of seconds to minutes per event. Without manual definition or static analysis, the actual timeout bound is unknown. In our approach, we do not have control over the node's clock, and as such, we do not include "real" timing failures.

However, due to the prevalence of retries and timeouts as resilience patterns in Service-Oriented Distributed Systems, we cannot completely disregard timing failures. Primarily, we must ensure that any adjustments to the system for testing purposes do not accidentally trigger a timeout bound. Furthermore, in some cases, we can simulate a response that indicates a timing failure occurred (a timeout was reached). In that case, we can still observe the resilience patterns related to timing failures, without controlling the timing of events.

3.1.4. Response Failures

Response failures are closely tied to the system's functional requirements. In particular, a response failure is a deviation from the system's specification.

In theory, we could inject failures that mimic the actual response of the system, but with a variation. However, in a Service-Oriented Distributed System, we assume that each component and their composition is tested through unit, unit integration and end to end tests. As such, we do not consider simulating response failures in this work.

More importantly, response failures are the type of failures that we want to *test for*. If, in the presence of simulated partial failures, we observe unexpected values or state transitions, then we observe incorrect behaviour. This work therefore simulates crash, crash-recovery, response omission and timing failures to investigate whether they cause response failures.

3.1.5. Cascading Failures

A common effect of partial failures is that a single partial failure causes another partial failure in a dependent node [9]. This is referred to as a cascading failure [5], [12]. This cascading effect can make it difficult to pinpoint the exact cause of a partial failure, as there might be multiple.

Regarding this work, we must account for the fact that failures can propagate through the system and are not isolated to a single node. But, nodes in a Service-Oriented Distributed System are encapsulated [16]. That is, a node can only process what it observes from its *direct* dependents. Hence, from an upstream node's perspective, it does not matter whether its direct dependency propagated an error or if it caused it; they are equivalent. Yet, from the perspective of the whole system, there can be a difference.

3.2. Functional Testing for Partial Failure Bugs

Given that a Service-Oriented Distributed System is prone to partial failures, we want to assert that the system functions as expected in the presence of partial failures. The question is: Given a correctness property, how can we verify that this property holds? Or, can we find a violation that indicates a partial failure bug? The following sections discuss techniques presented in the literature.

3.2.1. Fault Injection Testing

Fault injection testing involves simulating faults in a system. As faults can occur at unpredictable times, fault injection provides control over when and where faults occur. This allows for testing the SUT in a deterministic manner.

A test engineer can use their expert knowledge of the system to manually define test cases. Using fault injection tools, the system can be systematically perturbed, and its behaviour analysed. This approach has been shown to uncover bugs in real systems [46].

The downside of this manual approach is that it is incomplete. As famously said by Dijkstra in 1969: "Testing shows the presence, not the absence of bugs." Distributed systems are notoriously complex, and leaving it up to the expert to completely define all possible faults is error-prone. Service-Oriented Distributed Systems can be such complex systems that often not one person understands the entire functionality of the system. As Alvaro and Tymon [47] argue, it is better to systematically and automatically test than to rely on geniuses.

3.2.2. Challenges in Automated Testing

However, automated testing comes with its own set of challenges. In this section, we address two primary concerns that complicate automated testing for partial failures in Service-Oriented Distributed Systems.

Test Oracle Problem

Before we can verify a correctness property, we must first define it. However, defining a correctness property is non-trivial [48]. Distributed systems can have various properties that can be tested for. For example, distributed data structures such as relational databases and queues have well-defined properties (such as consistency and liveness) that should not be violated. For Service-Oriented Distributed Systems, these properties are domain-specific and often not well defined, or not at all defined.

We cannot automatically infer the correctness properties of the SUT. Instead, this requires the expertise of a developer. Consider the case of the fictional webshop, as presented in Figure 4.6. Once the user has created their order, filled in their payment details, and confirmed the purchase, the system must ensure that the order is processed. What should happen if the payment service is unavailable? What should happen if the payment succeeds, but the shipping service fails? All of these properties depend on the type of system and its domain, and involve a *decision* by the developer to define what the expected behaviour is. A tool can then verify if this expectation is upheld for all combinations of faults.

Exponential Explosion of the Search Space
In order to cover all possible combinations of states of a distributed system, we run into a problem: accounting only for partial failure nondeterminism (omitting concurrency nondeterminism), the number of states grow exponentially. Consider a Service-Oriented Distributed System providing different functionalities. To provide a specific functionality, m messages are communicated between different services. Each of these m messages can fail to be delivered. The failure of each message can (in the worst case) result in a potentially undesirable state of the system. For m independent messages, there are up to 2^m possible combinations of failures, each of which can result in a bug. The search space grows exponentially with the number of events. And, this only considers the nondeterminism of partial failures related to message passing for one system functionality. Each functionality can influence a different type of state for the system.

In other words, it is intractable to consider every possible distinct behaviour of the SUT. This makes it impossible to exhaustively test even the smallest Service-Oriented Distributed System wholesale [32]. However, techniques exist to decompose the problem into smaller, tractable problems.

3.2.3. Formal Verification

Formal methods can be applied to verify an *abstraction* of the system. To reduce complexity, only the core functionality is modelled in the abstraction. Potential failure types can be defined within the system's model. A model checker can verify that a defined property holds for *all* possible states. This provides a guarantee of completeness for the property.

A well-known model checker is TLC, with its corresponding specification language TLA$^+$ [49]. The specification language enables the modelling of a distributed system's behaviour, properties, and invariants in logical statements [50]. The model checker then verifies the invariant for each possible model of the specification. If violations are found, these are reported, and the specification can be changed to resolve them.

This method has been used in practice to verify the behaviour of crucial systems, such as distributed databases, queues, or filesystems, which are the backbone of other systems. For example, Amazon has used lightweight formal methods to verify its proprietary distributed algorithms [51], [52]. These systems often have well-defined invariants. As such, formal methods are an effective tool for proving properties of distributed algorithms.

The issue with formal verification for many organisations is its high learning curve and investment cost. Formal verification requires fundamental knowledge, often only available to highly specialised experts [47]. Instead, many opt to use formally verified distributed systems and build upon them. However, the composition of fault-tolerant components does not make a fault-tolerant system [34].

Furthermore, a formal model is only as good as its abstraction. Possible states can be missed if the abstraction does not represent the actual system. Furthermore, the implementation of the abstraction can still contain bugs. Hence, formal verification does not guarantee a bug-free system.

For Service-Oriented Distributed Systems this method poses two challenges in particular:

1. Functional domains are hard to abstract, as it is difficult to formalise business logic (if possible).
2. Most Service-Oriented Distributed System have emergent behaviour, and should be modelled as a composition of different services. But each service can provide numerous functionalities. This makes the search space explode, rendering verification intractable. Formal verification is more effective on smaller models.

As such, applying formal methods to Service-Oriented Distributed Systems is less effective compared to its application in other distributed informational systems.

3.2.4. Automated Fault Injection Testing

To close the gap between formal verification and manual fault injection, techniques exist that automatically cover (a subset of) the possible combinations of faults. In the following sections, we cover some of these techniques.

Random Testing

By automatically random sampling of the search space, it is possible to encounter bugs [53]. Especially with distributed databases, this approach has shown great effectiveness in finding network partition tolerance bugs [54].

A difference with finding network partition bugs compared to finding resilience bugs is that, for network partitioning, the problem's structure (the topology) is known upfront. In fault injection testing, it is possible to uncover new events due to resilience patterns. Using random testing, it is difficult to cover these cases.

Similar to manual experimentation, random testing is ineffective at providing guarantees and can only, by coincidence, drive the system into interesting states. Edge cases are easily missed if the search space is only sampled, instead of being structurally covered. It is challenging to provide guarantees about the system's resilience using random testing.

Chaos Engineering

This form of automated fault injection testing was popularised by the engineers at Netflix, where a tool called Chaos Monkey would randomly crash nodes to test the system's resilience. In principle, chaos engineering involves running resiliency experiments in production-like environments to expose resiliency bugs. It is primarily based on experiments: creating and testing a resilience hypothesis.

The primary objective of chaos engineering at Netflix was to foster a culture of resiliency within the company. If the assumption is that faults rarely occur, they can be easily overlooked. With Chaos Monkey, engineers are motivated to build resilient services.

Although this approach can effectively find bugs, the main issue is that it can inadvertently cause real system outages. It is simply too destructive to use in all Service-Oriented Distributed Systems. Modern chaos engineering approaches are combined with advanced monitoring solutions to ensure a minimised blast radius is not exceeded [55]. This allows the system to detect failures outside the scope of the experiment and quickly roll back the applied faults, ensuring no effect on the end user. Furthermore, chaos engineering suffers from the same issue as manual fault injection testing: it is incomplete and relies on human expertise.

Guided Search and Prioritisation

Instead of purely random testing, it is possible to use feedback from the system to steer the exploration towards potentially failure-prone states. The assumption is that certain states are more likely to reveal bugs, and prioritising partial failures that cause these states increases the likelihood of finding bugs.

This is an improvement over random testing, as it enables an informed decision on which combination of partial failures to investigate next. Studies have demonstrated the effectiveness of this technique [41], [56]–[59].

These techniques are well-suited for solving decision problems in the form of: *Is there a combination of partial failures for which the property does not hold*? By converging directly to potentially troublesome states, this can be answered quicker, while exposing more relevant parts of the search space. This is effective if there is a time limit or a budget to find bugs, as we can more efficiently use this time.

Even if the search space is explored in a more efficient order, in the absence of bugs, guided search tools would still need to cover all cases to answer the decision problem with a "no". In that case, the additional instrumentation required to guide the search might make it less efficient.

Search Space Reduction

Different techniques exist to reduce the search space of all combinations of faults to a tractable size [14]–[16]. Instead of bypassing the exponential explosion problem, they address the issue of whether the search space can be reduced to the point where it is tractable to test.

Given constraints or assumptions on the system's behaviour, we can reason about redundancies. This is the foundation of most reduction techniques. For instance, if in a specific system, specific combinations of faults are redundant, we can omit testing them [14]. Similarly, if two combinations of faults lead to the same observable state, only one representative needs to be tested [16]. Techniques such as Lineage-Driven Fault Injection (LDFI) exploit this idea by analysing causal relationships to eliminate fault combinations that cannot influence the outcome under test [15].

These reductions make exhaustive testing feasible while providing stronger guarantees. If no violations are found in the reduced space, we can argue that no violations exist in the whole space under the given assumptions. This contrasts with guided or random testing, which cannot claim the absence of bugs, but are effective at finding them.

4

Practical Automated Network-Level Fault Injection Testing

The goal of this work is to make it practical to analyse the effects of (combinations of) partial failures on a the behaviour of a Service-Oriented Distributed System. We want to simulate the effects of relevant failure types and verify the behaviour of a (Distributed) System under Test (SUT). To provide guarantees that a correctness property holds for a SUT, we utilise automated fault injection testing with search space reductions. This enables us to exhaustively cover the search space within a reasonable time frame.

In this chapter, we present our design for a practical approach to automated fault injection testing of Service-Oriented Distributed Systems, taking into consideration their characteristics and failure types as defined in Chapter 2 and Chapter 3, respectively. Furthermore, we developed an implementation of this design, called "Reynard", which can be applied to a wide variety of Service-Oriented Distributed Systems. The source code for Reynard can be found on GitHub [18].

We first present our design criteria and high-level design of our approach. We then discuss each component of this design, along with its implementation details. Finally, we discuss the known limitations and assumptions of our approach. We relate our design and implementation choices to existing techniques from the literature and the industry.

4.1. Designing an Automated Fault Injection Testing Strategy

Existing state-of-the-art automated fault injection tools were first considered for use. However, we found that they were either closed source [17], [41], [57], a proof-of-concept limited to only one or a few languages [16], [17] or lacked functionality that newer tools had [46]. Therefore, we could not trivially use them as a starting point.

To this end, we designed and developed Reynard. This automated fault injection tool fits our intents and purposes, and is flexible enough to evaluate different components of a fault injection tool. Our design draws inspiration from, and integrates components of, these earlier works. This is attributed in the relevant sections.

First, we state the specific goals and design criteria of our method. The following sections then describe the high-level components of the design and implementation that fulfil the goals and criteria.

4.1.1. Design Goals and Criteria

The goal of the test method is to be **applicable**, **composable** and **practical**. In the following sections, we define these goals and the criteria necessary to achieve them.

Applicable

It must be possible to use the tool in *existing* benchmark and industry systems. That means it should have a high ease of integration and be compatible with various systems. The tool should match the

system, not the other way around. As such, we define the following criteria:

- **Language Agnostic** - Considering the polyglot nature of Service-Oriented Distributed Systems, a feasible solution should not be specific to a single programming language. A solution that requires manageable and minor changes to the configuration of services is deemed feasible if the changes are independent of the number of features a service has.
- **Runtime Agnostic** - Service-Oriented Distributed Systems can run in many different runtimes. Even for the same system, there might be different runtime configurations based on the runtime environment (e.g., local, pipeline, staging or production). The tool should be able to operate in a wide range of these environments.

Composable
It should be possible to combine existing and novel components. As such, we define the following criteria:

- **Uniform** - To compose different components, they must share a common interface. This interface must be flexible enough to accommodate both existing and novel components, allowing them to be integrated into the tool.
- **Decoupled** - For components to be used together, they must function independently of each other. If components rely on shared information, they must have a uniform interface to access this information.
- **Measurable** - For evaluation purposes, we must be able to measure the impact of each component. Measurements should include the runtime of different components and their effect on the search space.

Practical
The tool should be usable in real-world settings. That means that it should be reliable, realistic and efficient enough to be applied to existing systems. As such, we define the following criteria:

- **Isolated Faults** - Any service within the system can provide numerous interactions. If a fault influences other interactions, simultaneous failures can occur, making it hard to distinguish the root cause. Therefore, faults must have a limited "blast-radius", contained to a specific interaction with the system. Of course, when a resilience bug is encountered, this can violate this blast radius, so this has to be accounted for.
- **Reproducible** - If a property is violated, its cause must be reproducible. This way, the resilience bug can be resolved. This requires faults to be deterministic.
- **Realistic** - The failure types must cover cases that can be realistically expected to occur in production systems. Furthermore, the test solution should run in a realistic time frame. Otherwise, the solution is unusable for real testing purposes.
- **Configurable** - To ensure that we can effectively evaluate the tool and increase flexibility, it should be possible to easily configure the parameters and settings of the tool.

4.1.2. High-Level Design

For our method, we expect that the following are already present:

1. A (Distributed) System under Test (SUT). In some cases, a subdivision of the SUT suffices.
2. A test scenario that covers a distinct functionality of the SUT by interacting with the system through its external interfaces.

In theory, we only need the services required to perform the test scenario. Hence, a subdivision of the SUT can suffice, depending on the system. It is possible to mock services that are only required to set up the test scenario. However, to be exhaustive, no mocked interfaces should be involved in the test scenario.

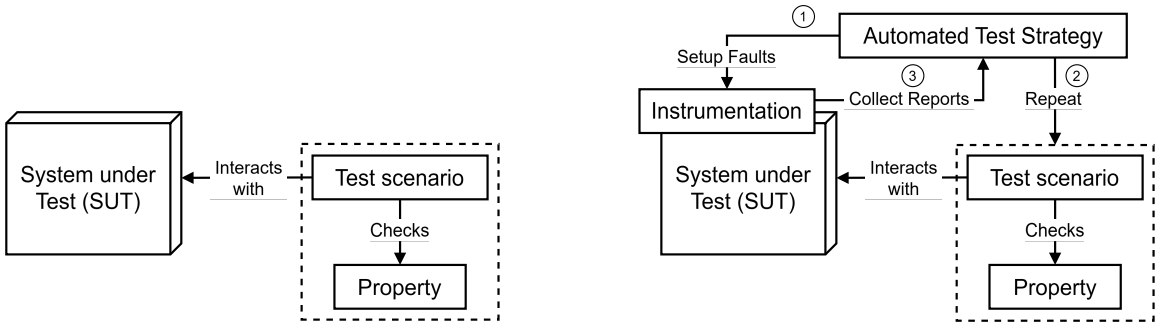
The test scenarios are comparable to integration, end-to-end or acceptance tests. An important condition for the test scenario is that it should be repeatable; its outcome should be independent of earlier executions of test scenarios and the exact number of times it is repeated. It should therefore be

deterministic. The test scenario can be an existing test that can be extended or a new one created for this purpose. Furthermore, we assume that a (preliminary) correctness property should hold for the test scenario. This should capture (at least) the expected behaviour of the "happy path" (the behaviour when no faults are introduced) of the test scenario. This property may need to be adjusted to account for the behaviour when the system is exposed to different combinations of partial failures.

Our high-level approach is to extend the SUT and test scenario:

1. We add instrumentation to the SUT. The instrumentation serves two purposes: to enable the deterministic injection of faults and to probe what happens within the system.
2. We run the test scenario repeatedly with different combinations of faults to verify that the property holds, or to find counterexamples.

Figure 4.1 provides a visual representation of this design. The automated testing strategy is responsible for iteratively planning and executing the test scenario with specific combinations of faults. It instructs the instrumentation on which faults should be injected when, and where, for a particular execution of the test scenario. Once the test scenario is concluded, it can collect reports on the interactions with the system from the instrumentation.



(a) The existing SUT and test scenario which verifies a correctness property.

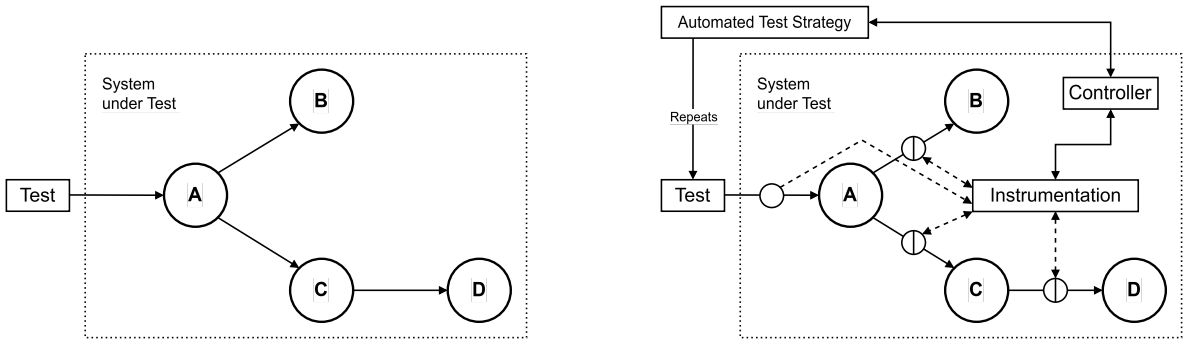
(b) The SUT with added instrumentation and an automated fault injection test strategy.

Figure 4.1: The high-level design of our testing approach. A (Distributed) System under Test (SUT) is instrumented to enable fault injections and analysis. Furthermore, an automated test strategy iteratively follows the phases indicated in the figure: 1) prepare a combination of faults to inject, 2) run the test scenarios, and 3) analyse the system's behaviour to find violations of the correctness property.

4.1.3. High-Level Instrumentation Design

To simulate faults in the SUT, our design requires the addition of instrumentation in the SUT. Specifically, we require instrumentation on the interaction between services.

Figure 4.2 provides a high-level overview of how the SUT is instrumented. The test scenario interacts with four services (*A*-*D*). We adjust the runtime to add instrumentation. The instrumentation should be able to inspect and modify messages on the network link between nodes. This can be accomplished in multiple ways, as discussed in Section 4.2. The root interaction with the system (at node *A*) does not have to be perturbed, as we do not want to assess the test's resilience. A centralised process (the "controller") oversees the instrumentation: collecting reports from the instrumentation, and coordinating when and where faults should be injected. This process provides the interfaces required by the test strategy.



(a) The interactions between services in the SUT for an interaction with an external interface of the system.

(b) The interactions between services in the SUT can be inspected and perturbed when instrumentation is introduced.

Figure 4.2: Our design for the instrumentation required for automated fault injection testing. The instrumentation can inspect and modify message exchanges between services in the SUT. A controller is responsible for overseeing the instrumentation.

Note that the final implementation of this design requires additional light-weight service-level instrumentation. This instrumentation is necessary to increase the precision of where and when faults are injected, as well as to isolate faults. The required instrumentation effort is minimal in most cases and is in some cases already present in the SUT. Section 4.3 provides further details.

4.1.4. High-Level Automated Test Strategy Design

The automated test strategy is responsible for exploring all combinations of faults that can influence the SUT for a given test scenario. Figure 4.3 provides a high-level overview of the stages and components that make up our automated test strategy. There are four stages: The workload, analysis, search, and faultload stage.

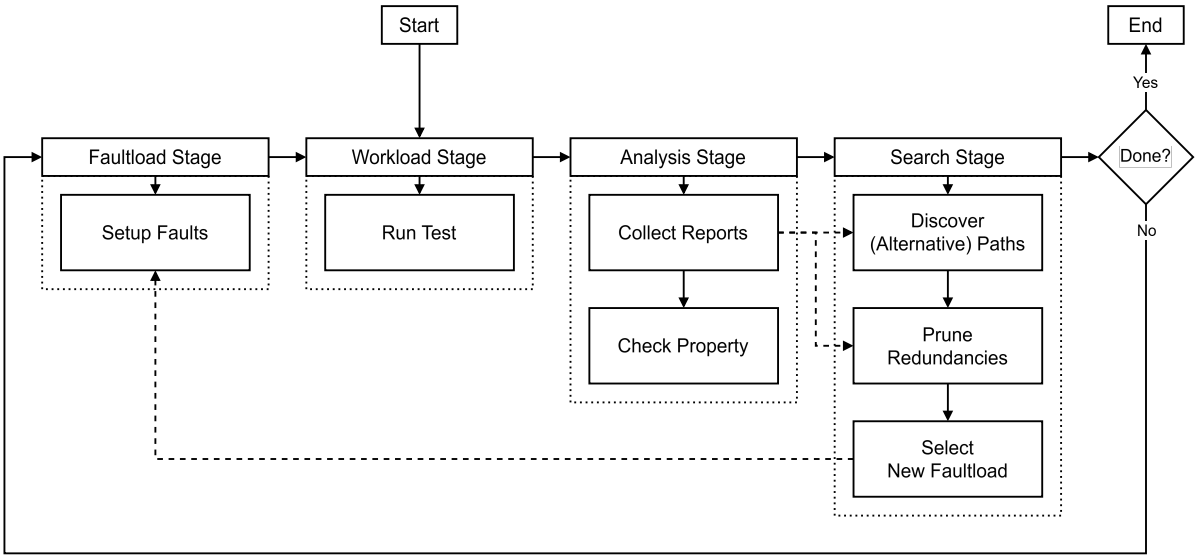


Figure 4.3: A high-level overview of the automated test strategy's components, stages and iterative process.

Workload Stage

The interaction between the test scenario and the SUT defines the "workload": the interaction with the system that can be perturbed. We begin by observing the system in the happy path to understand how it behaves without the presence of partial failures. Therefore, we begin at the workload stage, omitting the injection of faults on the first iteration.

Analysis Stage

During the analysis stage, we first collect reports from the instrumentation. These reports describe the interactions within the SUT that occurred during the test scenario. The collected reports are analysed to infer new information about the system's behaviour. Additionally, we check if the correctness property of the system is upheld when exposed to the injected partial failures. The metadata from the system's interaction can be used as part of the correctness property. The analysis provides feedback to the search stage.

Search Stage

The search stage is responsible for selecting a new combination of faults. First, we check if the injected faults introduced new paths that should be further investigated. In the first iteration, we uncover the paths that the happy path takes through the system. In subsequent iterations, we may uncover the effect of resilience patterns.

Based on the new information gathered during the analysis stage, we may identify redundant combinations of faults. We make sure we do not run the test scenario with these combinations of faults. Finally, a new, non-redundant combination of faults is selected for further investigation.

Faultload Stage

To ensure the instrumentation injects the intended faults, we communicate with the instrumentation controller to instruct it which faults to inject. Once the controller confirms this instruction, we are ready to repeat the test scenario. This process is repeated until all non-redundant faultloads are exhausted.

4.2. Fault Injection Method

Our testing strategy requires instrumentation to function. A key component of the instrumentation is the ability to simulate faults. Fault injection can be performed using different methods. However, not all methods are viable to use. In particular, because they are challenging to implement in an existing SUT.

This section addresses the following research question:

> (RQ1.1) What type of instrumentation can enable fault injection in Service-Oriented Distributed Systems with a reasonable integration effort?

To address this research question, we first examine the approaches presented in the literature and industry. Based on this, we describe our proposed fault injection method. Furthermore, we detail the specific failure types our tool Reynard can inject. Finally, we provide implementation details of Reynard.

4.2.1. Fault Injection Level

Faults can be injected at various levels of the application:

1. **Process-Level Fault Injection** - By perturbing complete processes directly, partial failures can be introduced into the system. This approach is often employed in chaos engineering practices, where processes are intentionally crashed.

   Causing failures, rather than simulating them, enables the evaluation of the system's realistic behaviour under partial failures. However, the overhead of crashing and restoring processes is significantly greater than simulating failures. Additionally, it is beneficial for faults to be isolated. Otherwise, concurrent interactions with the system are subject to the same injected partial failure, which can cause unintended side effects.
2. **Application-Level or Service-Level Fault Injection** - A more fine-grained approach is to inject faults by instrumenting a process' implementation [16], [17], [60]. For example, faults can be injected by adjusting the implementations of libraries, such as the HTTP client. Using this approach, we can simulate faults of external dependencies (such as the network or resources).

   The advantage of this approach is that faults can be injected from within the process. This enables access to process-level information, allowing for grey-box or white-box testing. For example, we can inspect messages before they are encoded into a network message. Additionally, runtime metadata can be accessed.

The downside of this approach is that it is not as practical as it is not language-agnostic. Implementing the same fault injection tooling for all relevant libraries of all relevant programming languages is often infeasible. For example, in [17], faults are injected by replacing the message serialisation library. If one considers only the JSON format, which has multiple implementations for most programming languages, it cannot be regarded as low-effort.

3. **Network-Level Fault Injection** - The effects of partial failures are perceived in the communication links between nodes. As such, the effects of crash failures can be *simulated* at the network level. And, evidently, network failures can be simulated at the network level.

   Messages can be modified in transit to simulate network failures. This can be achieved by routing all messages through a central process or placing agents in the network layer. These agents can be specialised network equipment [61] or (reverse) proxies [41], [46], [57]. These agents inspect network traffic and inject faults where instructed.

   This approach has the benefit of only requiring changes at the network layer or configuration, which is often uniformly defined. Another benefit is that it can be done with minimal or no changes to the original service. This approach has been successfully applied in multiple cases [41], [46], [57], [62].

   The growing popularity of network meshes has made this method more approachable, as network meshes already introduce proxies between each service. For example, the popular Istio service mesh can configure its proxies to inject network faults[1].

   However, using the fault injection capabilities of a service mesh is not sufficient for our testing purposes. For one, because not all Service-Oriented Distributed Systems use service meshes. Additionally, we require more fine-grained control over when and where faults are injected, as discussed in Section 4.3.

Given the goals and criteria of our design, we decide to apply network-level fault injection. This enables non-intrusive fault injection with minimal change to the existing system. In the implementation of our design, the fault injection is enabled by placing reverse proxies before each service of interest.

4.2.2. Supported Failure Types

At the network level, multiple types of failures can be simulated. From the types of failure defined in Section 3.1, we focus on simulating crash, crash-recovery, request omission failures and to a lesser extent, timing failures. As other nodes perceive the effect of a crash failure as a response or omission failure, these are perceptually equivalent. As such, we simulate failures by injecting response faults.

We do not simulate timing failures by controlling the timing of messages. Furthermore, we do not inject response omission failures. Although these failures would be feasible to inject using our instrumentation, and are relevant to Service-Oriented Distributed Systems, we chose to exclude them to reduce the scope of our work.

Table 4.1 provides an overview of the failure types supported by our tool Reynard, along with the corresponding status codes per protocol. We simulate failures by intercepting messages and returning erroneous responses that contain specific status codes, as defined by the HTTP[2] and gRPC[3] protocols. For example, we simulate a request omission by responding with a status code of 503 (“Service Unavailable”). In the gRPC protocol, this corresponds to a status code 14 (“Unavailable”). Crash-recovery failures are simulated by transiently injecting faults. For example, the first request to a service yields an injected message with an erroneous status code, but the second request is not perturbed.

[1] https://istio.io/v1.20/docs/tasks/traffic-management/fault-injection/#injecting-an-http-abort-fault
[2] https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status
[3] https://grpc.io/docs/guides/status-codes/

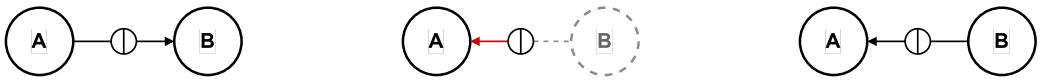
Table 4.1: Failure types simulated in Reynard and their respective status codes per protocol.

	Status per protocol	
Failure type	**HTTP**	**gRPC**
None	2xx	1
Crash	500, 502	2, 13
Request-omission	503	14
Timing	504	4

4.2.3. Implementation Details

To control communication between services, we introduce layer-7 reverse proxies as side-cars to each service of interest. Additionally, we introduce a controller process that provides an API for injecting faults and retrieving reports. To route traffic through the proxies, services are reconfigured to communicate with the corresponding reverse proxy, rather than communicating directly with the actual service. This can be done in the network configuration or the node's configuration.

Faults are injected as illustrated in Figure 4.4. The proxy inspects each message that is sent to the respective service. Based on the request's metadata, it can simulate faults if instructed to do so by the controller. If there are no faults to inject for the message, this setup is transparent to the services, as messages are forwarded directly.



(a) A request from service *A* to service *B*.

(b) The injected fault as an erroneous response.

(c) The unperturbed proxied response.

Figure 4.4: The fault injection technique applied by the reverse proxies. In case of an interaction of interest, the message is intercepted. An erroneous response is returned when simulating a fault. In that case, the targeted service does not receive the request. Otherwise, the proxy proxies the request and response.

There exist reverse proxies that can inject faults, but none could be trivially extended to support our specific needs. Therefore, we implemented a prototype of a reverse proxy to achieve the specific granularity required in our design. Additionally, we created a prototype implementation of an instrumentation controller that instructs the different proxies in the system.

Reconfiguring Existing Systems

The reconfiguration varies depending on the system's runtime. If remote addresses are baked into the system's processes, we can manipulate the network routing to reroute messages to the reverse proxies. If they are made available using a configuration provider, we can modify this configuration. In cases where the addresses are hardcoded and network routing cannot be changed, it may be necessary to modify the application source code to replace the hardcoded addresses with configurable ones. In our experience, we only had to adjust configurations in environment variables and/or load balancers.

Instrumentation API

Faults can be set up by using the controller's API. The controller can be sent a collection of faults to be injected for a specific interaction. In Section 4.3, we explain how the proxies know which interactions are of interest. The controller forwards this instruction to all proxies, which save all relevant fault instructions locally.

The proxy inspects each request to the proxied service. If the request is not of interest, it will be forwarded without alteration. If it is part of a known interaction, the proxy communicates with the controller to report the request and to determine if faults should be injected. A sequence diagram of this interaction is displayed in Figure 4.5. The controller aids the proxy in defining a unique identifier for the request, as detailed in Section 4.3. If the identifier matches an instructed fault, that fault is simulated. If not, the request and corresponding response are proxied.

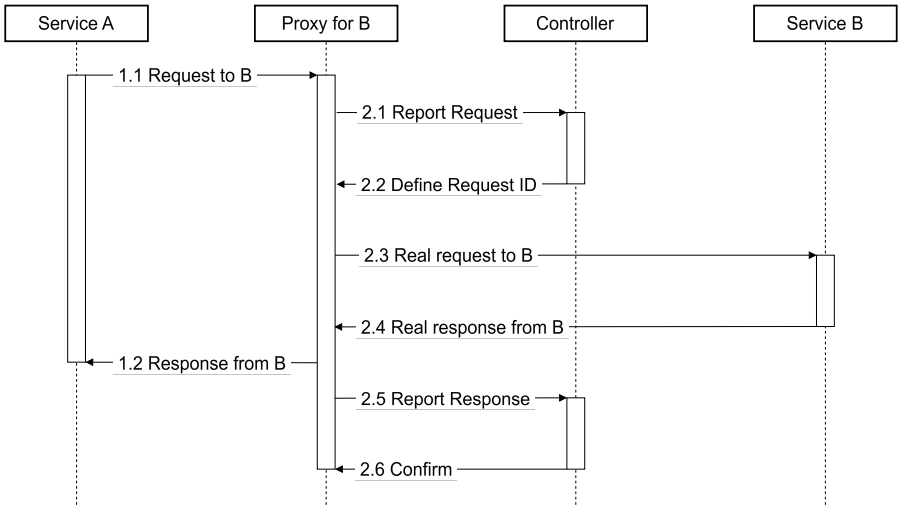


Figure 4.5: A sequence diagram of the interaction between a proxy and the controller for a request of interest, for which no faults should be injected.

After the response is forwarded, the proxy reports the response and relevant metadata (such as duration and which fault was injected, if any) to the controller. This is done *after* the response is forwarded to minimise the overhead of the instrumentation. After the entire workload is completed, the controller can be queried to return all collected reports. If some responses have not been collected, the controller is polled until it returns the complete reports. It is possible to determine if a report is incomplete when the request of an interaction is known but not its response.

Finally, the controller exposes an API to clear a registered faultload, preventing the storage of reports indefinitely. Because the controller is not aware when a test scenario is completed, this must be signalled by the testing strategy. This allows the controller and the proxies to release memory, preventing a memory leak.

Benefits of using Reverse Proxies

By introducing the reverse proxies at the receiving end of each service, there is no ambiguity about where each call is routed. In the SUT, network services (such as service gateways) can exist that route requests through the system. This can be a challenge in service-level fault injection methods [16]. Placing the proxies at the receiving end prevents all interactions that go through a gateway from being grouped at the same service. It is possible to place an additional proxy in front of the gateway to simulate a failing gateway.

Another benefit of this approach is its flexibility in integrating the instrumentation into the SUT. In containerised setups, each proxy can be deployed as a separate container. Alternatively, each proxy can be "baked" into the container, similar to [46]. While this is possible, it can be impractical as it requires different images for the fault injection test environment. It is beneficial if the services remain unaltered.

Finally, because each service has its own lightweight proxy, instead of a single centralised controller, it is difficult to overload the instrumentation. The proxy directly forwards all other traffic with minimal overhead. Only the traffic of interest is forwarded to the controller. This makes the instrumentation as transparent as possible to the system.

4.3. Analysing and Identifying System Interactions

In a Service-Oriented Distributed System, a single interaction with a service can result in numerous (transitively) related interactions between services. These interactions influence the behaviour of the SUT. Hence, influencing an interaction by injecting faults will affect the behaviour of the SUT. We want to be able to understand the effect of injecting different combinations of faults for the same test scenario.

First, we must be able to determine and analyse the causality of interactions. For one, this enables the determination of which internal interactions are related to the external interaction of the test scenario

with the SUT. If not, one would need to exhaustively explore all combinations of all faults for all interfaces of all services. This is an intractable number in even the smallest of systems. Furthermore, it enables us to reason about the interdependency of interactions, which helps to determine redundancies.

The instrumentation must also be able to identify equivalent interactions over different re-runs of the test scenario. This identification enables us to precisely instruct the instrumentation for *which* interaction we should inject a fault. Therefore, this identification must be deterministic and consistent for equivalent interactions.

Additionally, we want to be able to reason about the effect of faults. That is, if an interaction fails, what is its effect on other interactions? Having a consistent identifier enables us to perform this type of reasoning, as we can correlate the effect of faults by observing the system's behaviour. Additionally, in the case of a resilience bug, it provides the means to understand *which* interaction(s) caused the incorrect behaviour.

4.3.1. Correlating System Interactions

Inspecting messages between services on the network allows us to record interactions, but it cannot be directly used to find correlations. One proxy can capture that Service A communicates with Service B, and another that Service B communicates with Service C. Yet, this is not enough to infer that Service B communicates with Service C *because* Service A communicated with Service B.

Therefore, this section addresses the following research question:

(RQ1.2) What technique can identify the causal relation between interactions?

There are several approaches proposed in the literature and the industry for correlating system interactions:

1. **Manual Definition** - In theory, one could manually define all transitive interactions within the system. In practice, this is both an error-prone and infeasible task. A single service can have any number of interactions with other services. And, because services *and* teams are loosely coupled, it might be impossible for one person to know exactly what happens behind the interface of other services.
2. **Log Correlation** - Given enough monitored network requests and/or service logs, it is possible to correlate recurring events. E.g., it can be reasoned that logs regarding event B only occur in direct correlation to logs regarding message A being sent. This is the approach that Facebook took with their project called the "Mystery Machine" [63]. Essentially, it is a big-data solution to the problem.

   Although this approach can infer correlations with low overhead and high ease of integration, it has disadvantages. First, one needs a high number of requests to be able to correlate anything in the system. For local development environments, this is not feasible. Second, these correlations might not be completely accurate and can include many false positives and negatives. And third, it requires that logging is uniform and extensive in all services, which is not always the case. Hence, this solution is impractical for local, pipeline, or test environments.
3. **Distributed Tracing** - A common, modern approach to this problem is the use of distributed tracing. Most modern distributed tracing solutions are inspired by Google's work on their internal tracing tool called Dapper [64]. The idea is to correlate events (messages) to a tree-like structure called a "trace tree". This trace tree contains all related trace spans (events) with a hierarchical (parent-child) relation. The root span is the first event that starts an interaction with the system. Each trace tree has a unique identifier that relates spans to the specific tree it belongs to. The first span of a trace tree gets assigned this unique identifier. Furthermore, each span has a unique span identifier and is related to the span identifier of its parent. Additionally, for each span, the start and end times of their execution are reported, giving the tree a temporal aspect.

   To correlate spans, it uses a concept of "context propagation". When, within the context of a span, another event is initiated (i.e., a request is sent to another service), the message includes the trace identifier and the current span identifier in the request's metadata. When all spans are collected, the complete trace tree can be constructed with this information. Furthermore, in most distributed systems, a small amount of state can be stored in the context.

This approach requires minimal adjustments to existing code to function. That means some engineering effort is required. Primarily, because it is not language-agnostic.

Luckily, the engineering effort has already been performed by existing tools. Furthermore, there is an ongoing effort to establish an industry standard for distributed tracing that can be implemented across various languages and frameworks. The OpenTelemetry project[4], in particular, provides easy-to-integrate instrumentation for 11 programming languages and dozens of frameworks, at the time of writing. For 6 languages, the instrumentation can be added with zero code changes[5].

For our purposes, we use distributed tracing to correlate events in the SUT. This solution requires a reasonable amount of instrumentation effort and can be used in various programming languages and frameworks. In many languages, it can be automatically added without requiring any code changes. Furthermore, this approach has been proven successful by earlier work [16], [17], [41], [57].

4.3.2. Uniquely Identifying Equivalent Interactions

The addition of distributed tracing provides the means to correlate interactions. Distributed tracing can provide detailed metadata about interactions within and between services. However, by default, this information is only available *after* the interaction has completed, as tracing data is only sent once a span is completed. To reduce overhead, the tracing data available during transit is kept to a minimum. Only a centralised collecting service has the complete picture of all spans, once they have all been completed.

For our purposes, we must be able to determine when and where to inject faults *during* the test execution. Tracing information alone is therefore insufficient for this purpose. However, we can use the context propagation functionality of distributed tracing for fault injection. To simulate faults in the system, we define a unique identifier for each fault injection point using the available request metadata. This must be consistent for different (but equivalent) executions of the test scenario.

This section addresses the following research question:

(RQ1.3) Can we uniquely and consistently identify equivalent interactions between services in the system?

Fault Identification

Several solutions have been proposed in the literature. Gunawi et al. [13] identify a fault injection point by a key-value mapping that contains service-level metadata provided by the instrumentation. These metadata include the call signature, service origin and stack trace. To provide an identifier, a hash of the mapping is taken as a more concise representation. A similar approach is taken by Joshi et al. [14]. Its identifier represents the execution context at the moment of an outgoing call, which should be consistent given an unaltered process.

The work by Alvaro et al. [15] introduces a logic language (called "Dedalus") that can precisely reason about the origin and dependencies (lineage) of events in the system. Although provably complete, rewriting a complete codebase to Dedalus is infeasible and undesirable for any industry system [33].

A continuation of this work is the 3MileBeach platform [17], based on the earlier work by Bittman et al. [65]. This line of work argues that accurate fault injection and observability are closely intertwined. It proposes a tool that instruments the serialisation libraries used to perform network requests. In particular, it proposes how the temporal aspect of faults can be encoded into the identifier by partitioning time into segments. It uses identifiers for the signatures and the time segment of each Remote Procedure Call (RPC) to identify fault injection points.

The work by Meiklejohn et al. [60] argues for a more general approach by introducing the concept of a Distributed Execution Index (DEI). It provides examples of scenarios where different RPCs can be hard to distinguish and how to differentiate them. Specifically, it argues that RPCs can be best identified by a unique sextuple defined by the target service, the call signature, the payload, the predecessor events leading up to the request and the invocation count of the other fields within the same trace. To implement this, it extends the Java implementation of the OpenTelemetry instrumentation for the JVM.

[4] https://opentelemetry.io/
[5] https://opentelemetry.io/docs/zero-code/

Fault Identification in Reynard

We agree with the assessment that fault injection should be closely related to observability, hence our use of distributed tracing. However, identifiers in related work generally rely on service-level metadata, which our network-level instrumentation cannot access. Instead, we opt to use a variation of a DEI [60] to identify requests at the *network level*. As the work by Meiklejohn et al. [60] argues, it *requires* a sextuple of information to identify requests uniquely and consistently. A part of that sextuple relies on service-level metadata. As such, we expect there to be indistinguishable interactions at the network level in some specific cases. Section 4.7 outlines the anticipated challenges leading to, and arising from, indistinguishable interactions.

We identify an interaction, which we refer to as an *fault injection point*, by a stack of quintuples $\langle$Destination, Interface, Payload, Predecessors, Count$\rangle$. Each quintuple defines an interaction, and the stack represents the causal dependencies that precede it. Each quintuple is composed of the following fields:

- **Destination** - The name of receiving service. Multiple replicas of the same service can exist in the SUT. As services are stateless, they can be given the same destination identifier.
- **Interface** - A unique identifier for the interface provided by the service. This can be the name of the RPC or endpoint.
- **Payload** - A representation defining the arguments given to the interface of the service. Can be hashed to reduce the space required to store this part of the identifier, taking into account the (low) risk of collisions.
- **Predecessors** - A mapping that represents the (temporally) preceding sibling interactions.
- **Count** - A number indicating how often an interaction with the same destination, interface, payload, predecessors and causality has occurred.

The first two fields of the quintuple define a signature for the functionality being called. The payload then defines the intention (arguments) of the interaction. Similar to local functions, we assume that calling the same function with the same arguments and observing the same side effects should yield the same result.

These three fields alone are not sufficient to distinguish interactions, as different services can invoke the same interface. Therefore, the path of invocations that causally lead to the current invocation is part of the identifier (the stack). This allows the differentiation between calls to the same endpoint from distinct origins.

It is insufficient to distinguish interactions by their causal origin and the invoked interfaces [60]. For instance, the same service may call the same endpoint multiple times. For example, a common resilience pattern is a retry, where the same interaction is repeated upon failure. This results in the exact same request occurring multiple times. To address this issue, we maintain an invocation count.

Not only can the same interaction occur multiple times, but it can also occur for different reasons. In related work, this distinction is made by keeping track of the call stack from *within* the service using service-level instrumentation [13], [14], [60]. However, this is not possible with network-level instrumentation. If the runtime is synchronous, we can recover related information by tracking the completed interactions that precede each interaction. In particular, we track the maximum invocation counts of interactions originating from the same service (i.e., siblings), based on the request's destination and signature. This representation of the preceeding events leading up to the interaction In an asynchronous runtime, this identification is unsound and can render the identifiers non-deterministic, resulting in inconsistencies. This is a limitation of our tool, which we further highlight in Section 4.7.

For the test suite to *query* faultloads, we allow wildcards in the fault injection point identifiers sent to the proxies. These are "*" for the string fields, $null$ for the predecessors field and -1 for the count field. These match any value of the field. In the analysis, we identify interactions in the system by their exact fault injection point identifier, but faults to be injected can be defined using wildcards.

Example Interaction

To understand what we mean by a fault injection point within a test scenario, we give the example of a fictional webshop, as illustrated in Figure 4.6. In this interaction, we test the checkout endpoint of the

Checkout service. To check out the user's order, the Checkout service must first retrieve the current items in the cart from the Cart service. If this call fails, the Checkout service performs a retry as a resilience mechanism.

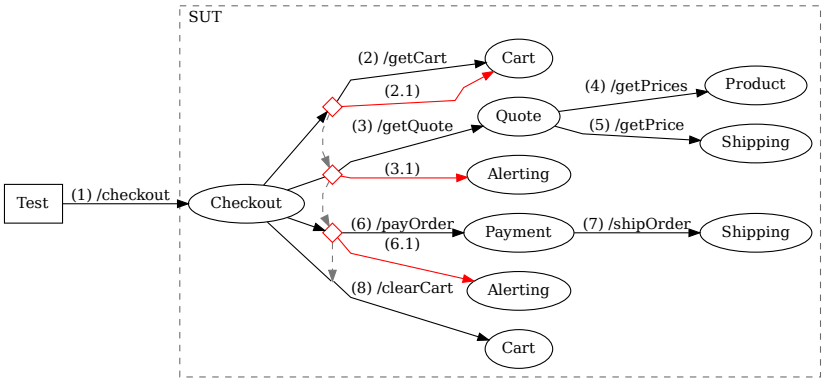


Figure 4.6: The call graph of a fictional scenario of a webshop checkout service, involving 7 services and 10 fault injection points. Dashed grey lines indicate a happens-before relation between events, and a red diamond and a red line indicate a conditional event in the case of a failure.

If either call is successful, the service can proceed with obtaining a quote for the order. To perform this, the Checkout service interacts with the Quote service. In other words, the call to the Cart service *happens before* the call to the Quote service.

To obtain a quote, the total price of all products is summed with the shipping costs. These interactions are done concurrently, and the result is computed on completion of all requests. If any of these requests fail, the Quote service returns an error to the Checkout service. If this happens, then the Checkout service cannot proceed, and a call is made to the Alerting service. In this interaction, the Checkout service is the *upstream* service calling a *downstream* service (the Quote service). Furthermore, the calls from Checkout service to the Quote and Cart service have the same *direct causal dependency*: the call to the Checkout service.

Once the Checkout service determines the order price, it can use the Payment service to charge the user's credit card. The Payment service calls the Shipping service to prepare the shipment if the payment succeeds. If either fails, this is reported to the Checkout service, which sends a request to the Alerting service, so that the payment or shipping issue can be resolved. If the payment is successful, the Checkout service can clear the cart by calling a different endpoint in the Cart service.

This test scenario involves 7 different microservices and 10 fault injection points (represented by the black lines), of which 3 (represented by the red lines) are conditionally dependent on the failure of other interactions. We do not consider the interaction between the test and the Checkout service as a fault injection point.

In this example, the call to the Product service from the Quote service is identified as:

$$\begin{array}{llllll} [\langle \text{Checkout}, & /\text{checkout}, & \#, & \{\}, & & 0\rangle, \\ \langle \text{Quote}, & /\text{getQuote}, & \#, & \{\text{Cart}/\text{getCart} : 0\}, & & 0\rangle, \\ \langle \text{Product}, & /\text{getPriceOfCart}, & \#, & \{\}, & & 0\rangle] \end{array}$$

And the retry to the Cart service as:

$$\begin{array}{llllll} [\langle \text{Checkout}, & /\text{checkout}, & \#, & \{\}, & 0\rangle, \\ \langle \text{Cart}, & /\text{getCart}, & \#, & \{\}, & 1\rangle] \end{array}$$

4.3.3. Implementation Details

When a message is routed through an instrumentation proxy, the proxy can inspect the message to determine a unique identifier. The proxy knows the destination it represents as it is aware of the service

it mimics. Furthermore, the request contains an endpoint in the request headers, and the payload is part of the request body and query string. Hence, these three fields can be trivially obtained.

However, a single proxy is not aware of the interactions preceding it. It only has access to the trace information provided by the distributed tracing. It cannot directly use the context propagation header information, as internal trace spans are unknown to the controller. To allow proxies to determine the previous event, Reynard uses the context propagation mechanism to create a new trace tree span and set a new predecessor in the context propagation state. Using this technique, each subsequent proxy has access to a unique identifier that allows it to look up its predecessor, whose identifier determines the tail of the new identifier.

To determine the identifier, a proxy communicates with the instrumentation controller, providing it with the available information about the trace identifier, its parent span identifier, and the interaction's span identifier. The controller then determines the full identifier, as it is aware of the identifier of the predecessor and tracks the invocation count and preceding events.

The automated test strategy utilises the controller's API to collect all reports generated by the proxies for a specific trace identifier. It further analyses these reports to determine parent-child relations between events, and checks if reports are complete (all request reports have a corresponding response) before further usage.

We additionally use the context propagation state to store metadata related to a proxy's functionality. A flag indicates whether the event should be inspected for fault injection. This flag is not present in "regular" network traffic. On the first request from the test to the SUT, a flag states that it is the initial request and no predecessor exists. Other flags signal whether the response body should be stored in full or as a hash, and whether the predecessors should be included in the identifier.

4.4. Fault Space Search Strategy

With the ability to detect relevant fault injection points and simulate specific faults at those points, we can start searching for *combinations* of faults that result in incorrect behaviour.

Two main challenges make creating a search strategy non-trivial:

- The search space or "fault space", is exponential in the number of fault injection points. Testing all possible combinations of faults is intractable for even small numbers of interactions.
- Injected faults influence the behaviour of the system. It is impossible to know how the system will behave without knowing the full semantics of the SUT. For example, if we know a system's happy path, we do not know the resilience patterns present in the system.

Hence, our representation of the search space must be able to change dynamically, either omitting or adding regions of the search space.

This section addresses the following research question:

(RQ1.4) Which representation allows us to enumerate the search space efficiently?

4.4.1. Defining the Fault Space

First, we define the composition of the search space. This definition is similar to those defined in [13] and [14]. Given the set of fault injection points P and the set of failure modes M, we define the following:

1. A **fault injection point** (or point) $p \in P$ is a stack of quintuples (⟨Destination, Interface, Payload, Predecessors, Count⟩), as defined in Section 4.3.
2. A **fault** is a pair of a fault injection point $p \in P$ (the when and where) and a failure mode $m \in M$ (the how), written (p, m) or f_p^m. The set of possible faults is $P \times M$. We denote a fault at point $p \in P$, regardless of its mode, as f_p.

   We additionally define a **behaviour** as a fault injection point and either a failure mode, or no failure mode ($\perp$), denoted as b_p^m. A behaviour without a failure mode represents the behaviour in the happy path and is denoted as $b_p^\perp$. A fault defines an intended partial failure, whereas a behaviour defines the observed behaviour of an interaction, which can be equivalent to a specific fault.

3. A **faultload** $F = \{f_{p_1}^{m_a}, f_{p_2}^{m_b}, \ldots, f_{p_n}^{m_z}\}$ is a unordered set of faults. A fault injection point can only be represented once in a faultload, but failure modes do not have to be unique. This implies that a fault injection point can be prone to at most one failure mode at a time. We denote the set of fault injection points covered by the faultload as $P(F) = \{p_1, p_2, \ldots, p_n\}$.
4. The **fault space** $\mathcal{F}$ represents all possible faultloads: $\forall F, F \in \mathcal{F}$. We will use the terms fault space and search space interchangeably.

The complete fault space contains all the unique combinations of faults, i.e., all the unique ways that all subsets of P can be combined with all failure modes in M. The size of the fault space is exponential in $|P|$ and polynomial in $|M|$. This is the case because every point can only be represented at most once in a faultload. Otherwise, it would be the powerset of the Cartesian product $P \times M$.

If we consider only one failure mode ($|M| = 1$), then the fault space is the powerset of P, each unique subset of P. The size of $\mathcal{P}(P)$ is $2^{|P|}$. If we consider the powerset as the union of all subsets of size k of P from $k = 0$ to $k = |P|$, we can rewrite this definition as:

$$| \mathcal{P}(P) | = \sum_{k=0}^{|P|} \binom{|P|}{k} = 2^{|P|} \tag{4.1}$$

If we then consider that for a given subset $S \in \mathcal{P}(P)$, where $|S| = k$, each fault injection point $p \in S$ can fail in one of $m = |M|$ ways, then there are m^k possible faultloads (sets of pairs of fault injection points and failure modes) for S. Hence, we can express the size of the complete fault space as:

$$\sum_{k=0}^{|P|} \binom{|P|}{k} |M|^k \tag{4.2}$$

Using the binomial theorem, which is defined as:

$$\sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k = (x + y)^n \tag{4.3}$$

And substituting for $x = 1$ and $y = |M|$, we can define the size of the fault space as:

$$\sum_{k=0}^{|P|} \binom{|P|}{k} |M|^k = (1 + |M|)^{|P|} \tag{4.4}$$

If we define $n = |P|$ and $m = |M|$, then in the worst case, we have to cover $O(n, m) = (1 + m)^n$ faultloads to exhaustively cover the fault space. Hence, it is exponential in $|P|$ and polynomial in $|M|$.

For a realistic case of 12 fault injection points and 4 failure modes, this yields over 244 million faultloads. If each test takes anywhere from milliseconds to seconds, this yields a runtime in the order of days to years. Moreover, enumerating all cases quickly reaches the memory limits of modern machines. Therefore, we must optimise our search strategy by efficiently ignoring redundant cases while representing the search space in a manner that fits into memory.

4.4.2. Efficiently Representing and Enumerating the Fault Space

There are multiple ways to enumerate all possible faultloads in the search space. We know that our search problem has the following properties:

- Enumerating the full space is computationally intractable.
- A significant number of faultloads in the fault space are redundant.
- The known search space can dynamically change, due to discovering new fault injection points during exploration.

It is intuitive to enumerate the search space in order of increasing sizes of faultloads. This allows us to observe the isolated effect of singular faults or minimal sets of faults on the system's behaviour.

Taking this into account, we represent the fault space as a directed rooted tree, starting with the empty set, and each level adding one element to the subset. Figure 4.7 shows a complete search tree for the ordered list of fault injection points $[A, B, C]$ and failure modes $[1, 2]$. Each node represents the subset defined by the union of its value and the values of all its predecessors. In the example, there are 3 fault injection points and 2 failure modes, and therefore there are $(1 + |M|)^{|P|} = (1 + 2)^3 = 27$ nodes.

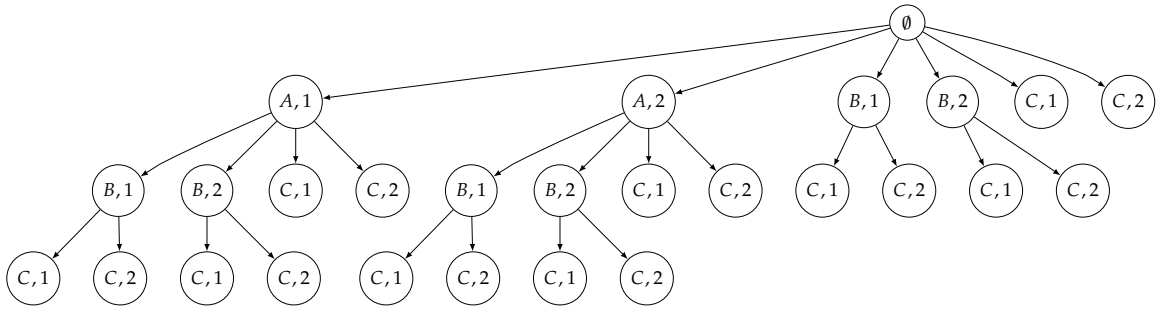


Figure 4.7: The complete search tree for fault injection points $[A, B, C]$ and failure modes $[1, 2]$. Each node represents the set defined by the union of its value and the values of its predecessors.

The complete search tree can be generated by starting with the root node (the empty set) and generating the subsequent child nodes recursively as follows:

1. For each fault injection point that is not part of the subset represented by the node, a new potential node is generated for all failure modes. Hence, we generate at most $|P| \cdot |M|$ children.
2. We check if we have already added a node to the tree representing the same value. If so, we do not add the duplicate node to the tree. As a faultload is a set, the order of faults in the faultload does not matter.

Each node in the search tree represents a faultload (F), and its subtree represents supersets of this faultload. The supersets are defined by the list of points (P_{sup}) that can be added to F to "expand" it. Similar to the search space size, the subspace represented by a node is $(1 + |M|)^{|P_{sup}|}$. For example, for the $(A, 1)$ node in Figure 4.7, there are still 2 fault injection points left that can be added to the faultload. Therefore, there are (at most) $(1 + |M|)^{|P_{sup}|} = (1 + 2)^2 = 9$ nodes in the search tree related to fault f_A^1. If a faultload's supersets are redundant, then the smaller it is, the larger the subspace that can be pruned.

The primary benefit of this tree representation is the ability to build the search space lazily, rather than enumerating it in full. Once a node is visited in the search tree, we can check if it is redundant. If it is, we do not run the test scenario with the representative faultload, and do not generate any child nodes. This way, we can effectively prune all supersets and avoid visiting them. In Section 4.5, discuss which combinations of faults are redundant.

Reduced Search Trees

As the search space can grow exponentially but is repetitive, we can simplify its representation as pictured in Figure 4.8. In this representation, we do not denote the failure modes. Instead, we denote only the fault injection points. As before, each node represents the subset defined by the union of its value and the values of all its predecessors. So, the node A represents nodes $\{\{(A, 1)\}, \{(A, 2)\}\}$. The bottom-most C node represents all 8 combinations of $[A, B, C]$ with $[1, 2]$. These 8 combinations correspond to all nodes in the lowest level of the search tree in Figure 4.7. The labels on the edges indicate the number of nodes in the search tree that are represented by the node in the reduced search tree. The (maximum) number of represented nodes per level (l) is equal to $|M|^l$.

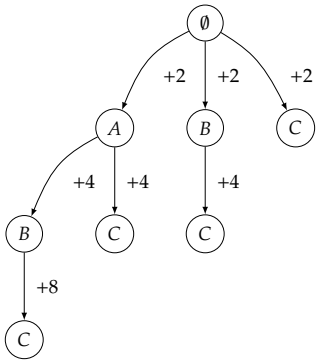


Figure 4.8: A reduced version of the search tree as visualised in Figure 4.7. Each node represents the set of fault injection points defined by the union of its value and the values of its predecessors *and* the combinations of failure modes with that set. Labels on the arrows indicate the number of nodes in the complete search tree that correspond to the node in the reduced search tree. Child nodes are visited left to right.

In the following section, we discuss how the order of exploration affects the efficiency of covering the search space. As such, our representation of the reduced search tree has an explicit order: child nodes are visited from left to right.

4.4.3. Exploration Order of the Fault Space

Because we dynamically prune the search space based on observations, the order in which we visit nodes can influence the efficiency. That is because the order in which we infer information about the system's behaviour can influence which reductions are possible. We can perform a different exploration by choosing a depth-first or breadth-first traversal of the *search tree*, or by visiting child nodes in a different ordering. The child nodes represent the fault injection points that can be added to the current faultload. For the root node of the search tree, this corresponds to *all* singular fault injection points.

The call graph provided by the instrumentation (as shown in Figure 4.6) can be used to determine the order in which child nodes are visited. There are several options for ordering the fault injection points, each leading to a different search tree. We can order the fault injection points based on a depth-first or breadth-first exploration of the *call graph*. For a depth-first exploration, we can choose to perform either a post-order or pre-order traversal. And, at each level of the call graph, we can visit nested fault injection points in a reverse or non-reverse temporal order.

Exploration Order in Related Work

In the related work by Meiklejohn et al. [16], a concolic-style exploration of the search space is taken. This involves a depth-first, reverse post-order traversal of the call graph, with a depth-first exploration of the search tree. The ordering of the call graph corresponds to the reverse sequence of events as they occurred. It is essential to note that service-level instrumentation is aware of the dependencies between events. As such, it is possible to know which events can be combined and which combinations are infeasible. Furthermore, a depth-first exploration of a tree is generally more memory-efficient when using a queue or stack.

Exploration Order in Reynard

This exploration order is suboptimal for our approach. We do not have access to the same information regarding the dependencies of events. Hence, we do not know upfront which events can not be combined; this must be observed. Instead, we explore the call graph in a depth-first non-reverse post-order, and explore the search tree in a breadth-first manner.

As mentioned in the previous section, we find it to be beneficial to explore the search tree in a breadth-first manner, meaning we explore faultloads in increasing order of size. This allows us to infer the root causes of behaviours and dependencies, which makes it simpler to analyse the system's behaviour. Additionally, this prevents us from visiting an infeasible combination, as we first visit individual fault injection points before considering their combination. The downside is a larger memory footprint of our implementation.

Exploring the call graph in a depth-first post-order enables us to observe how erroneous responses are propagated up the call graph before visiting those points. This can lead to reductions in the search space, which a breadth-first ordering can not. In the call graph, we cannot directly determine the order of events that have the same causal dependency (i.e., are caused by the same call to a service's endpoint). But we have some clues as to the temporal order in which events occur. There is a high likelihood that events are related to other events that occur *later*, which is what we aim to determine. As such, we order fault injection points in the order they are reported to the controller (non-reversed). Hence, we order fault injection points based on the depth-first non-reversed post-order of the call graph.

4.4.4. Implementation Details

The search tree, pruning policies and exploration algorithm are components of the testing library of Reynard. This library is implemented as a JUnit 5 plugin[6], allowing users to simply add a decorator to a test to automatically explore the fault space. Furthermore, the library enables the test to obtain the call graph and corresponding analysis, allowing it to make assertions about its properties.

A "generator" component is responsible for generating and exploring the search tree efficiently, taking into account observations to prevent reaching infeasible combinations of faults. The generator utilises the pruning policies to identify redundancies in the search tree.

Algorithm 1 represents a simplified version of our implemented search algorithm. This algorithm resembles a model checking algorithm [66], but instead of performing actions in a state, we add faults to the current faultload.

Algorithm 1 The automated fault space search algorithm.

Require: Test scenario T
Require: Correctness property C
Require: Failure modes M
Require: Pruning policies P_s

1: **procedure** FAULT-SPACE-SEARCH
2: $N_{queue} : \mathcal{F}[] \leftarrow [\varnothing]$
3: $N_{visited} : Set[\mathcal{F}] \leftarrow \{\}$
4: **while** $|N_{queue}| > 0$ **do**
5: $F : \mathcal{F} \leftarrow$ DEQUEUE(N_{queue})
6: $N_{visited} \leftarrow N_{visited} \cup \{F\}$
7: **if** SHOULD-PRUNE(P_s, F) **then** ▷ If any pruning policy determines F is redundant
8: continue ▷ Then: prune it
9: $Reports \leftarrow$ RUN-WITH-FAULTS(T, F)
10: ANALYZE($P_s, F, Reports$) ▷ Analyse the effect of the injected faults
11: **if** HOLDS($C, Reports$) **then** ▷ Check if the correctness property still holds
12: $P_{reachable} \leftarrow$ DISCOVER($TraceTree$) $\setminus P(F)$ ▷ Determine possible extensions
13: $N_{children} \leftarrow$ EXPAND($F, P_{reachable}, M$) ▷ Determine the corresponding nodes
14: $N_{add} \leftarrow [N \in N_{children} \mid N \notin N_{queue} \wedge N \notin N_{visited}]$ ▷ Ignore duplicates
15: $N_{queue} \leftarrow N_{queue} \mid\mid N_{add}$ ▷ Add candidate nodes to the queue
16: **else**
17: **return** False
18: **return** True

We maintain a queue of search tree nodes (i.e., faultloads) to visit, starting with the root node (the empty set). While there are still candidate nodes to visit, we take a candidate from the queue. First, we check if it is redundant to visit. Each pruning policy defines a predicate function that returns whether it deems the faultload redundant. If any pruning policy determines a node to be redundant, the node is not explored further, and its subtree is ignored.

[6] https://junit.org/junit5/

Nodes are pruned once they are intended to be visited. Theoretically, they could be pruned when they are inside the queue and new information becomes available. By pruning nodes once they are scheduled to be visited, we can access the most information to determine whether they are redundant. Pruning them inside the queue could reduce memory space, but we empirically found the computational overhead to be more significant than the queue's memory footprint.

If it is not redundant, the test scenario is run with the preconfigured faults. This results in a list of reports, which represent every interaction that the instrumentation recorded. The result of running the test scenario is then analysed. Multiple components in the tool can analyse the results. For example, our test oracles can use the information to detect incorrect behaviour. Additionally, the pruning policies can analyse the system's behaviour to infer information that would make certain nodes redundant. Furthermore, we check if the correctness property holds for the node. If not, we have found a violation, and we can stop the test execution.

If the correctness property holds, we can continue building the search tree. First, we determine which fault injection points can still be combined with the current faultload based on our observations. This is an ordered list of fault injection points, based on the ordering of the call graph, as discussed in Section 4.4.3. Then, we generate the new child nodes that correspond to the faults composed of the reachable fault injection points paired with all possible failure modes. Finally, we add all nodes to the queue that we have not yet explored or that are not already in the queue. This continues until we have considered all reachable, non-redundant combinations of faults.

4.5. Dynamically Pruning the Fault Space with Pruning Policies

There are combinations of faults that are redundant and do not have to be visited. This is because some combinations of faults are infeasible to combine or do not provide new information. To reduce the search space and make it computationally tractable, we incorporate several pruning policies in our tool. To effectively apply these pruning policies, we must understand how different combinations of faults influence the system.

This section addresses the following research question:

(RQ1.5) Can we model the influence of faults to reason about the expected system behaviour and the equivalence of combinations of faults?

To explain our pruning policies, we refer back to our fictional webshop as presented in Figure 4.6. We refer to fault injection points in the figure by their annotated edge numbers. E.g., we refer to the request from the Checkout service to the Alerting service as $p_{3.1}$. Additionally, we define the following terms:

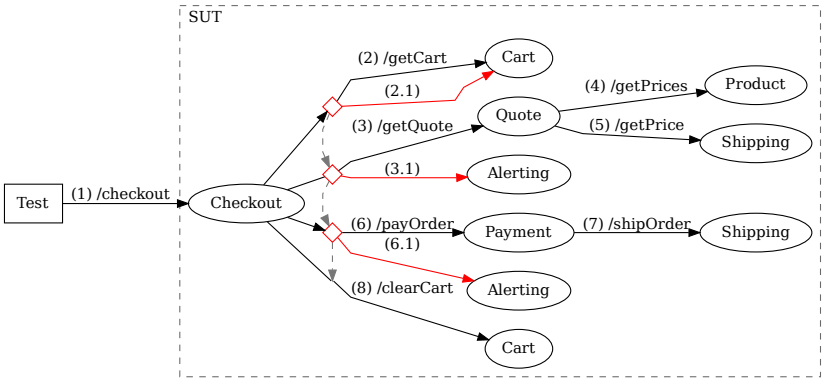


Figure 4.6: The call graph of the fictional scenario of a webshop checkout service with 10 fault injection points.

- **Upstream** and **Downstream** - When two services interact, we refer to the service that initiated the interaction as the *upstream* service. We refer to the receiving service as the *downstream* service. In a request-response pattern, the request originates from the *upstream* service, and the response originates from the *downstream* service. In a call graph, we draw arrows pointing towards the

downstream service. For example, when the Checkout service calls the Cart service at p_2, the Checkout service is the upstream process, and the Cart service is the downstream service.

- **Causal dependency** and **happens-before** - If one event (e_1) *causes* another event (e_2), then we say that e_2 is causally dependent on e_1, or that e_1 happens before e_2. Both are denoted as $e_1 \xrightarrow{hb} e_2$.

  If they are transitively related (e.g., $e_1 \xrightarrow{hb} e_2 \xrightarrow{hb} e_3$), than we denote this as $e_1 \xrightarrow{hb+} e_3$. For example, the request to the Quote service happens before the request to the Product service. And, the *response* of the Quote service happens before the request to the Payment service. Therefore, the request to the Quote service transitively happens before the request to the Payment service (as denoted by the grey arrow).

  We say that two events are **sibling events** if they have identical causal histories. That is, they share the exact same set of causal predecessors. For example, if $e_1 \xrightarrow{hb} e_2 \xrightarrow{hb} e_3$ and $e_1 \xrightarrow{hb} e_2 \xrightarrow{hb} e_4$, than e_3 and e_4 are sibling events. In Figure 4.6, all outgoing requests from the Checkout service are sibling events.

The following sections define the pruning policies used in Reynard to reduce the search space.

4.5.1. Downstream Dependency Pruning Policy

A faultload F is redundant if $\exists f_u, f_d$ such that $f_u \xrightarrow{hb+} f_d \wedge \{f_u, f_d\} \subseteq F$. That is, we cannot combine faults at fault injection points that are causally dependent, as our fault injection method prevents causally dependent events from happening. Combining causally dependent faults is infeasible and therefore redundant.

Based on the call graph, we can trivially infer one form of causal dependency: downstream dependencies. This information is directly contained within the identifier for each fault injection point. For example, a fault in the request to the Product service (p_4) cannot be combined with a fault in the request to the Quote service (p_3). The identifier of p_3 is contained in the stack of the identifier of p_4.

In general, if there is a (transitive) happens-before relation between two fault injection points, e.g. $f_{p1}^{m_x} \xrightarrow{hb+} f_{p2}$ (regardless of the causal dependents' failure mode), then the subset $\{f_{p1}^{m_x}, f_{p2}\}$ is redundant. In the case of downstream requests, we know that regardless of the failure mode of the upstream request (f_u), the (transitive) downstream request (f_d) will not occur as $f_u \xrightarrow{hb+} f_d$. Therefore, a faultloads F for which holds $\{f_u, f_d\} \subseteq F$ is redundant.

4.5.2. Fault Injection Point Exclusion Pruning Policy

We cannot combine faults when one fault excludes another from occurring. This is due to another form of causal dependency, but between fault injection points that have the same direct causal dependency. In other words, between events in the call graph that are each other's siblings.

Opposed to ancestral dependencies, this piece of information cannot be directly inferred. For example, based on the information available for the initial happy path, it is impossible to infer that $p_2 \xrightarrow{hb} p_3$. Based on the instrumentation, we do know that they occur in a temporal order, but we do not know their causal relation. Due to scheduling nondeterminism, we cannot even be sure this order is consistent.

We can observe the *effect* of partial failures to gain related information. Instead of knowing that $p_a \xrightarrow{hb} p_b$, we can observe that if p_a is erroneous, then p_b does *not* appear in the call graph. We assume that if an event is expected to occur (based on earlier observations) but doesn't, the failed sibling events leading up to it cause its disappearance. To ensure we first find the smallest subset that causes an exclusion, it is beneficial to visit the search tree in a breadth-first manner.

In general, if the presence of one or more faults, which we refer to as $F_{exclusion}$ (e.g., all $\{f_{p_2}, f_{p_{2.1}}\}$), makes another fault injection point ($p_{excluded}$) disappear from the call graph (e.g., p_3), then the faultload F is redundant if $F_{exclusion} \cup \{f_{p_{excluded}}\} \subseteq F$. This is regardless of the failure mode of $f_{p_{excluded}}$, as it is infeasible to inject any fault at $p_{excluded}$ when $F_{exclusion}$ is injected.

4.5.3. Fault Injection Point Inclusion Pruning Policy

We cannot inject faults at a fault injection point that only conditionally occurs, but for which no condition is present in the faultload. Faults can trigger redundancy mechanisms, such as retries and fallbacks. This can cause new fault injection points to appear. However, without injecting a cause, we cannot introduce faults at the new fault injection point.

The appearance can be observed in the call graph. For example, fault f_{p_2} causes point $p_{2.1}$ (the retry) to appear. However, we can only inject faults at $p_{2.1}$ if a fault is injected at p_2. Thus, injecting faults at $p_{2.1}$ without a fault at p_2 is redundant.

There can be multiple conditions that cause the appearance of new fault injection points. For example, the fault injection point $p_{2.1}$ appears, regardless of the failure mode at p_2. We keep track of all potential causes for the inclusion of $p_{2.1}$. To ensure we find the smallest subset that causes an inclusion, it is again beneficial to visit the search tree in a breadth-first manner.

Essentially, this is the converse of the exclusion pruning policy. Given a fault injection point p that only occurs when combined with one of the sets of faults in $\mathcal{F}_{preconditions} = \{F_1, F_2, ..., F_n\}$. Then, a faultload F is redundant if $f_p \in F \land \neg\exists F_{pre} \in \mathcal{F}_{preconditions}, F_{pre} \subseteq F$.

4.5.4. Upstream Propagation Pruning Policy

Faults can be propagated upstream, allowing for a reduction as we observe an additional fault that we did not intentionally inject. When a downstream dependency of an interaction responds with an erroneous response, the upstream service must decide how to handle this. In some cases, this erroneous response can be resolved. For example, we can use retries to recover from transient failures, fall back to another service that can provide a similar or suboptimal response, or use a default response to continue.

However, if a downstream failure cannot be resolved, then it is common to respond upstream with an erroneous response indicating the failure. In this case, a fault is propagated up the call graph. This does not have to be the same fault, yet an erroneous response appears where no fault was injected. For example, if an error is injected at p_7, then the response at p_6 will also be erroneous.

After executing a test scenario with faultload F, we can make the following observation. Given a fault injection point p_u that is the causal dependency of fault injection points $P_{downstream} = \{p_d \in P \mid p_u \xrightarrow{hb} p_d\}$. We injected (a subset of) faults ($F_{downstream} \subseteq F$) at (a subset of) the downstream fault injection points $P(F_{downstream}) \subseteq P_{down}$. This caused the equivalent behaviour of failure mode $m \in M$ at p_u ($f^m_{p_u}$). Then we know that any faultload which contains $F_{downstream}$ will additionally observe $f^m_{p_u}$. Note that if we intended to inject $f^m_{p_u}$, then we would not observe the fault injection points in $P(F_{downstream})$ due to causality.

Due to our injected faults, we additionally observed the effect of $f^m_{p_u}$. If $f^m_{p_u}$ occurred in isolation to *its* causal dependency, we can omit visiting the faultload $F = \{f^m_{p_u}\}$, as we have already observed its isolated effect. Additionally, we can use this information to find redundancies by substituting $F_{downstream} \in F$ by $(F \setminus F_{downstream}) \cup \{f^m_{p_u}\}$, as they result in equivalent behaviour. We build upon this idea in the following pruning policy.

4.5.5. Unreachability Pruning Policy

Each of the four preceding policies provides information on the effect of injecting faults at fault injection points. In particular, we can understand how requests cause related requests down the call graph, how responses include or exclude related events, and how responses propagate up the call graph, resulting in a final response. If we combine these pieces of information, we can determine if a specific combination of faults is infeasible. Specifically, we can use this information to predict which fault injection points should be observed when injecting faults ($\tilde{P}(F) = \{p_1, p_2, ...p_n\}$). Then, we can prune any faultload that attempts to inject faults at fault injection points that will not be observed during a test invocation ($P(F) \supset \tilde{P}(F)$).

For example, if we intend to inject the faultload $F = \{f_{p2}, f_{p_{2.1}}, f_{p_7}\}$, then none of the preceding pruning policies can correctly determine the redundancy. The point inclusion policy can reason that $f_{p_{2.1}}$ appears due to f_{p_2} and is thus not redundant. The point exclusion policy can conclude that faults at p_6 would be

redundant, but these are not part of the faultload. Yet, we know that f_{p_7} can only be injected if the event at p_6 is included, due to the downstream dependency.

Modelling System Behaviour

To deduce this, we can model how the system will behave for a given faultload based on our current understanding of the SUT. To express this, we have to reason about the (local) behaviours at fault injection points, not just faults. We predict which fault injection points will be included for a given set of intended faults using the algorithm presented in Algorithm 2.

Algorithm 2 The algorithm to predict the expected behaviour of the system for a given set of faults.

Require: Causally dependent downstream requests $D_{req} : P \mapsto P[]$
Require: Upstream responses related to direct downstream behaviours $U_{res} : \mathcal{B}[] \mapsto \mathcal{B}$
Require: Exclusion conditions per point $C_{excl} : P \mapsto \mathcal{B}[][]$
Require: Inclusion conditions per point $C_{incl} : P \mapsto \mathcal{B}[][]$

Type Expected-Behaviour-For : $(P, \mathcal{F}) \rightarrow (Set[\mathcal{B}])$
procedure Expected-Behaviour-for(p_{root}, F)
    $(b^m_{p_{root}}, B_{downstream}) \leftarrow$ Unfold-Behaviour(p_{root}, F)
    return $\{b^m_{p_{root}}\} \cup B_{downstream}$

Type Unfold-Behaviour : $(P, \mathcal{F}) \rightarrow (\mathcal{B}, Set[\mathcal{B}])$
procedure Unfold-Behaviour(p, F)
    if $\exists f^m_p \in F$ **then**
        return $(b^m_p, \emptyset)$ ▷ Inject a fault
    $P_{downstream} \leftarrow D_{req}[p]$
    if $P_{downstream} = \emptyset$ **then**
        return $(b^{\perp}_p, \emptyset)$ ▷ Leaf node in the call graph
    $B_{current} : \mathcal{B} \mapsto Set[\mathcal{B}] \leftarrow \{\}$
    for $p_d \in P_{downstream}$ **do**
        $(b^m_d, B_d) \leftarrow$ Unfold-Behaviour(p_x, F) ▷ Unfold downstream requests recursively.
        $B_{current}[b^m_{p_d}] \mapsto B_d$
    $B_{current} \leftarrow$ Apply-Conditional-Behaviour($p, B_{current}$) ▷ Apply exclusions and inclusions
    $B_{direct} \leftarrow \bigcup_{u \in B_{current}} \{u\}$
    $B_{all} \leftarrow B_{direct} \cup \bigcup_{u \in B_{current}} B_{current}[u]$
    if $\exists B_{observed} \in U_{res}$, Match($B_{observed}, B_{direct}$) **then**
        $b^m_p \leftarrow U_{res}[B_{observed}]$
        return (b^m_p, B_{all}) ▷ Return upstream behaviour matching downstream behaviours
    return $(b^{\perp}_p, B_{all})$

Type Apply-Conditional-Behaviour : $(\mathcal{B} \mapsto Set[\mathcal{B}]) \rightarrow (\mathcal{B} \mapsto Set[\mathcal{B}])$
procedure Apply-Conditional-Behaviour(p_{parent}, B_{start})
    $B_{current} \leftarrow B_{start}$
    $P_{related} \leftarrow$ Topological-Sort($p_{parent}, C_{incl}, C_{excl}$)) ▷ Sort sibling points
    for $p \in P_{related}$ **do**
        $include \leftarrow$ Should-Include($p, p \in P(B_{start}), C_{excl}[p], C_{incl}[p]$)
        if $include$ **then**
            $(b^{m_x}_p, B_{downstream}) \leftarrow$ Unfold-Behaviour(p, F)
            $B_{current}[b^{m_x}_p] \mapsto B_{downstream}$
        else
            $B_{current} \leftarrow$ Exclude($p, B_{current}$)
    return $B_{current}$

The procedure Expected-Behaviour-For predicts the behaviour of the system for the test scenario, given a

faultload F, which we refer to as $\tilde{B}(F) = \{b_{p_1}^{m_x}, ..., b_{p_n}^{m_z}\}$. The algorithm presented works as follows. We start at the root of the call graph: the interaction between the test and the SUT. Then we recursively "unfold" the behaviour of the nested interactions, storing the nested downstream behaviours. Ultimately, we determine the complete set of behaviours by the union of the behaviour of the root interaction and all its nested behaviours.

For each request (fault injection point p) in the call graph, we first check if we intend to inject a fault. If that is the case, we directly return the behaviour of the fault, omitting any downstream behaviour.

If we are not intent on injecting a fault at p, we consider all downstream requests in the happy path that are directly related to the fault injection point. If there are no downstream requests, then we have reached a leaf node in the call graph, and as there are no faults to inject, we return a happy path behaviour. This is a base case in the recursion. If there are downstream requests, we recursively unfold their behaviour to determine their expected response and transitively related behaviours.

For each possible sibling fault injection point that has p as the direct causal dependency, we must determine whether it is included or excluded. The downstream requests provide a starting point to evaluate exclusion and inclusion conditions. As fault injection points can rely on the inclusion or exclusion of other fault injection points, we first perform a topological sort based on their dependencies to evaluate the points in a logical order. Then, for each possible fault injection point, we determine whether it should be included or excluded based on whether it was included in the happy path and the inclusion and exclusion conditions. As services can perform actions based on any logical condition, we determine if it is included based on the most complex exclusion and inclusion condition, i.e., the matching condition with the most dependencies. We prioritise the most complex matching condition (either an inclusion or exclusion), giving preference to inclusions in the event of a tie.

If a fault injection point should be included, we unfold it and add it, along with its downstream behaviours, to the expected behaviours. If it should be excluded, we ensure the fault injection point is not part of the expected behaviours.

Finally, we determine if we have observed (erroneous) behaviour that corresponds to the observed direct downstream behaviour. In other words, we check if a fault was propagated based on the observed downstream behaviour. We then return the expected behaviour of the fault injection point (either a propagated fault or the happy path), as well as *all* downstream behaviours.

We use the expected behaviour to determine if we are attempting to inject faults at infeasible fault injection points. In notation: a faultload F is redundant if $P(\tilde{B}(F)) \subset P(F)$.

4.5.6. Service Encapsulation Reduction Policy

The work by Meiklejohn et al. [16] proposes a pruning algorithm and exploration strategy they refer to as "Dynamic Reduction". This pruning policy is based on the idea of service encapsulation: "Service encapsulation states that invoked downstream dependencies of a service can only indicate failure or success to their direct caller, and as long as they do not expose their internal state to the invoker, the invoker can only assert on the responses that are returned to the caller by those dependencies." [16]. In other words, the response of a stateless service is solely affected by the responses of its direct downstream dependencies. In the dissertation by Meiklejohn [32], this policy is referred to as "encapsulated service reduction". We refer to it as a "Service Encapsulation Reduction Policy".

With this pruning policy, we prune faultloads that introduce no new behaviour in the system, than what an earlier test execution already observed. The reduction algorithm is as follows: Given a faultload F, its expected behaviour $\tilde{B}(F) = B_F$, and previously observed behaviours $H = [B_1, B_2, ..., B_n]$ from earlier test executions. It is redundant if: for every interaction ($b_{p_c}^{m_c} \in B_F$) and its direct downstream dependencies $B_{p_c}^{down} = \{b_{p_d}^{m_d} \in B_F \mid p_c \xrightarrow{hb} p_d\}$, it holds that there exists a previously observed execution $B_h \in H$ where all dependencies were observed and had the same observed behaviour ($\forall b_{p_d}^{m_d} \in B_{p_c}^{down}, b_{p_d}^{m_d} \in B_h$). In short: faultload F is redundant if $\forall b_p^m \in \tilde{B}(F),\ \exists B_h \in H,\ \{b_{p_d}^{m_d} \in \tilde{B}(F) \mid p \xrightarrow{hb} p_d\} \subseteq B_h$.

In essence, this covers two intuitions: If a service interface's dependencies behave similarly, we expect the same response (the concept of service encapsulation). And, if the injected faults cause no new behaviour, it is redundant. For example, if the faultload $\{f_{p_7}, f_{p_{6.1}}\}$ has been observed, and we infer

with error propagation that $F \cup \{f_{p_7}\}$ is equivalent to $(F \setminus \{f_{p_7}\}) \cup \{f_{p_6}\}$, then the faultload $\{f_{p_6}, f_{p_{6.1}}\}$ is redundant as $\tilde{B}(\{f_{p_7}, f_{p_{6.1}}\}) \supset \tilde{B}(\{f_{p_6}, f_{p_{6.1}}\})$.

4.5.7. Retry Reduction Policy

Finally, we include a pruning policy based on the findings of [38]: most correctness property violations in retries can be found by testing for only transient or persistent failure. The reasoning is that retries are a resilience pattern based on the assumption that faults can be transient. Correctly implemented retries should cause the system to behave the same for each retried event until a retry limit is reached.

The reduction policy is as follows: If a retry is detected, instead of scheduling the faultload with the new fault injection point, we schedule a persistent fault of the same failure mode instead. The persistent fault is represented by the original fault injection point but with a count of -1 (the wildcard), which instructs the proxies to match on all values of the count field. All faultloads that contain a retry are deemed redundant, as well as the combination of the persistent fault with the transient fault. We detect retries by identifying whether a fault causes the inclusion of a fault injection point that differs from the original by only one increment of the count field, where in the happy path there was only one fault injection point.

Normally, to be exhaustive in the case of retries, we must check for the fault that causes a retry and all combinations with all subsequent failure modes of the retries. Hence, for n retries of f_p^m, given m possible failure modes, there are $1 + m + m^2 + ... + m^n = \sum_{i=0}^{n} m^i$ possible combinations of faults that cover the retried fault injection points. These can be further combined with all other fault injection points. Using the retry reduction, this becomes 2, one transient and one persistent.

An additional benefit is that, when we discover the retry mechanism, it is impossible to know how many retries will be issued. By instructing the proxies to inject faults on all subsequent retries, it can be immediately uncovered how many retries there are, instead of scheduling n faultloads.

Although a trivial counterexample could be created, we assume that any implementation of retries has a distinct control flow based on the type of (transient) failure observed, and that each retry follows the same control flow until the retry limit is reached. For example, a program that behaves differently on the first, second, and third retries would not be correctly covered, but we argue that such programs are uncommon.

As this pruning policy is not strictly sound, it is not enabled by default; however, it can be configured in a test using a flag in the test decorator. If the happy path contains two fault injection points that are distinguishable only by their count, they are not considered for the reduction.

4.5.8. Implementation Details

Similar to the *generator* component, each pruning policy is another component of Reynard. Prune policies must implement a prune function that takes as input a faultload and returns whether it is redundant. Pruning policies can additionally implement a Feedback interface that is called in the analysis stage with details about the observed behaviour of running the test scenario with a specific faultload.

Both interfaces provide some form of context. This context provides an interface to stored information previously inferred by other components. The Feedback context provides the means to store new information, while the *prune* function can only read information.

4.6. Defining Correctness Properties

With the ability to correlate interactions over different runs of the test scenario, we are left with the question: What *is* the correct behaviour? This is known as a test oracle, and it is a non-trivial problem [48], which we discussed in Section 3.2.2.

An additional challenge is defining expected behaviour under the composition of faults. If the expected behaviour for fault f_x is B_x, and for f_y it is B_y, then what is the expected behaviour for the faultload $\{f_x, f_y\}$? One can incorrectly assume it to be $B_x \vee B_y$, but in reality it is $B_x \vee B_y \vee B_z$, where B_z can be an entirely new class of behaviours. As we discussed in our pruning policies, faults can restrict (exclude) the behaviour of the system, but can also extend (include) it.

4.6.1. Testing Resilience Patterns

One application of fault injection testing is verifying resilience patterns. For example, we can verify that if a fault is injected for a request, it is retried. Based on the trace tree, this information is available to perform assertions. This is similar to the assertions made possible by [46].

Verifying more complex resilience patterns, such as *Circuit Breakers* or *Bulkheads*, does not fit well into the fault space exploration provided by Reynard. This is because these faults make the interactions with the target system stateful, resulting in inconsistent test suite results. However, it is feasible to manually perform this type of assertion using the Reynard instrumentation. To test for the effects of a *Circuit Breaker* pattern, it is possible to first introduce enough failure-containing calls to trigger the breaker, and then assess the behaviour. Although this is feasible, manual fault injection testing is not the focus of Reynard.

4.6.2. Testing the Atomicity of State Transitions

In principle, services in a Service-Oriented Distributed Systems are stateless. But they often rely on databases to store state. So, they depend on *a* state to function.

If interactions involve a stateful operation, such as a state transition or creation, other services might rely on the same state. For scalability, multiple copies of the same state can be represented in different data storages. The system must ensure that the state is kept in sync.

Failures can complicate the synchronisation. This is a common challenge in databases, but the same is true for Service-Oriented Distributed Systems. Consider the webshop example. If the webshop administrator intends to remove a product from the product catalogue, the product must also be removed from all users' shopping carts. Otherwise, users might accidentally purchase non-existent products. However, if the removal from the product catalogue is successful, but not from the carts, then this is an issue.

Testing for atomicity involves inspecting the system state, which can be an expensive operation. In the mentioned example, the property could be that once the request completes, either the product is removed from the catalogue and all carts, or an error is returned, and the product still exists. To validate this, the state of both services should be queried and inspected. This can significantly increase the overhead and complexity of the test, but it is possible to test using Reynard.

As such, we can create an automated verification of this correctness property, but we cannot automatically determine the correctness property. It requires a *decision* by the developer to define this correctness property.

4.6.3. Automated Test Oracles in Reynard

Preferably, our automated test strategy is accompanied by *automated* test oracles that apply to all systems [48]. These test oracles can check for common anti-patterns and generate warnings for potential bugs.

Reynard includes two automated test oracles based on common anti-patterns observed during our application to existing systems. These oracles take inspiration from [38], which includes several automated test oracles. The test oracles do not fail a test case, as they can be false positives, but instead issue warnings. The test oracles included in Reynard are:

- **Failures without Cause** - During fault injection, we expect that observed failures can only be caused by direct fault injection or propagation of injected faults. Hence, every observed failure should have an injected failure somewhere downstream. If we encounter a failure without a cause in its causal dependents, we assume this is an indication of incorrect behaviour.

  The underlying cause can differ. Failures can be introduced due to unexpected state deviations. Or, the test scenario's behaviour depends on previous faults, which is a known limitation. Either case should warrant an investigation by the developer.
- **Propagation of Semantically Incorrect Status Codes** - A common failure pattern is error propagation: if a service's downstream dependencies fail, the service responds to its upstream dependency with an erroneous response.

However, a common occurrence is that the errors are directly forwarded, copying (parts of) the message and status code. This can lead to unintended effects due to the semantics of specific status codes. For example, a status code 503 semantically means the message was never sent to the intended server. Hence, it should be safe to retry the request, regardless of whether the endpoint is idempotent, as the message is never processed. However, if a server forwards a 503 status code, this violates these semantics. The server did receive the messages; it was not unavailable, but its downstream dependency was. This might lead to an upstream dependency incorrectly retrying on a non-idempotent endpoint. The correct behaviour is to respond with a 500 status code ("Internal Server Error").

To detect this pattern, Reynard includes an automatic test oracle that checks whether a response contains an unexpected status code that is not due to an injected fault. If we can detect which downstream dependency's status code is forwarded, this is reported. This also covers the case where a service responds with a semantically incorrect status code by default.

Another possible automated test oracle would be to test for **faults that exceed their blast radius**. For example, if an injected fault causes an application-level failure (e.g., a crash). This is not included in Reynard because it is runtime dependent. If the test scenario has access to check whether nodes are still operational, this could be part of a correctness property checked in the test scenario. As we discus in Section 3.1.1, it can be impossible to detect system failures. Therefore, this check can result in false positives.

4.7. Known Limitations

Our method has some inherent limitations in its method and assumptions on which it is based. We discuss each in the following sections.

4.7.1. Indistinguishable Fault Injection Points

Despite the metadata available to distinguish events, systems can perform requests in the happy path that, from Reynard's perspective, only differ in invocation count. As shown in [60], there exist scenarios that require application-level metadata to be uniquely identified. Due to scheduling nondeterminism, using the count to differentiate can cause issues. Furthermore, if these events have retries as a resilience mechanism, as this causes events to "shift" their invocation index between cases. As a result, inferred information regarding the events' effect (e.g., exclusions and inclusions of other fault injection points) is incorrectly attributed to the wrong event.

In general, if the happy path of the test scenario includes non-distinguishable points *and* includes a retry resilience pattern or scheduling nondeterminism, then Reynard's verification can become unsound. Additionally, as the following sections illustrate, solutions to address issues with nondeterminism can lead to indistinguishable events.

4.7.2. Nondeterminism in System Behaviour

Reynard can handle several sources of nondeterminism, but not in all conditions. We know of some conditions in which Reynard cannot explore the fault space soundly. For specific cases, there are solutions or workarounds to restore soundness.

Behavioural Nondeterminism in Downstream Requests

If endpoint X causes a downstream request to either endpoint Y or Z, chosen at random, then Reynard cannot predict which will be called and can issue faultloads with faults that are not injected. In other words, if the behaviour of an interface is probabilistic, then Reynard is unsound.

A similar pattern happens if the SUT includes a load balancer. In that scenario, we assume that the system's behaviour would be equivalent if all events reached the same service, as they are all stateless copies. We can solve this by placing the proxy *before* the load balancer (instead of the service), or by configuring each proxy so that they are the same destination. In that case, Reynard can still soundly cover the fault space, under the assumption that each node is equivalent.

Data Nondeterminism in Upstream Response
Some nodes can respond to requests in a nondeterministic manner. For example, if identifiers have to be generated, or time is involved. This might cause a different behaviour in the upstream service, and make the test scenario nondeterministic.

If it is impossible to avoid the source of the nondeterminism, Reynard can be configured to omit the payload field as part of the fault identifier. As a result, all requests to the same interface become equivalent, regardless of any nondeterminism in the payload. However, if a service in the test scenario issues two calls to the same endpoint of the same service, then these events become indistinguishable.

Scheduling Nondeterminism
In systems with scheduling nondeterminism (e.g., concurrency and/or asynchronicity), Reynard can function, as long as each event is consistently distinguishable by its fault identifier, and the observable effects of re-orderings are deterministic. If non-unique calls are issued, the only difference is their invocation count, which can be inconsistent as their order is nondeterministic.

A communication pattern we observed is the use of the `Promise.all` function in JavaScript. Due to the asynchronous single-threaded runtime of JavaScript, each event is issued in order. In this pattern, the failure of *any* response will return in a failure of the call. However, it can be that requests are not yet issued when a failure is returned. In its implementation, these events are then not issued. Hence, the scheduling nondeterminism of the responses causes nondeterminism in the downstream requests, which leads Reynard to an incorrect model of the system behaviour.

Furthermore, scheduling nondeterminism can result in data nondeterminism. If stateful operations (such as state transitions and state retrieval operations) can have different orderings between test cases, then equivalently identified events might result in a different behaviour.

4.7.3. Assumption of Independence of Test Invocation
To compare the same system interaction with different faultloads, Reynard assumes that each iteration of the test scenario is independent. If this is not upheld, then Reynard cannot function properly. For example, if a fault in a previous test causes the following test invocation to fail. This burdens the user to ensure that each test can be run repeatedly without errors.

4.7.4. Assumption of Equivalent Behaviour for Equal Status Codes
Reynard reasons about the system's behaviour by classifying responses with the same status code as equivalent. However, if a service has different behaviours for the same status code, Reynard can become unsound. Based on its stored information, it can incorrectly model the system's behaviour and prune test cases it should not.

Services that exhibit different behaviour for the same status code are likely to do so based on the data contained in the response. Including (a hash of) the response data in the behaviour definition would be more complete, but is prone to data nondeterminism.

This is a trade-off between precision and practicality. For Reynard, we assume that data-nondeterminism is more likely to be present in a system than behavioural differences for equal status codes. An option would be to allow the user to define equivalence classes of responses. To reduce scope, this is not included in Reynard.

5

Evaluating Reynard

The goal of our evaluation is to answer the research questions and to verify the choices made in our design. The following sections outline the methods used to evaluate Reynard and describe how the evaluation procedures were established.

5.1. Experimental Setup

We want to ensure that each evaluation is representative, repeatable, and fair. Therefore, we took the following precautions:

- All benchmarks were run on the same machine. This was a Windows 10 desktop machine running WSL2 with an AMD Ryzen 7 3700X 8-core CPU and 32 GB of RAM.

  For our industry system, we interacted with a test environment using a provided virtual machine. This was a virtual desktop with three virtual cores of an Intel Xeon Gold 6254 CPU with 32 GB of RAM.
- We closed all unnecessary applications to minimise the influence on the measurements.
- The time measurements are an average of multiple runs to ensure consistent results and an unbiased runtime. For each benchmark, we state the number of iterations used. The overhead of starting and stopping the SUT is excluded from the runtime.

5.2. Finding Benchmarks Systems

To evaluate Reynard, we searched for publicly available (demonstration) Service-Oriented Distributed Systems. In particular, we were interested in systems that included (or could be extended to include) OpenTelemetry and that could be run locally. This significantly reduced the number of potential benchmark system, which is a common challenge [16], [31]. We found and considered the following benchmark systems:

1. The work by Meiklejohn et al. [16] includes a *corpus* of benchmark systems. These are publicly available on GitHub[1]. As their work evaluated their results on this *corpus*, we included these benchmarks as they enabled us to make a comparison.
2. The OpenTelemetry Astronomy Shop[2] is a Microservice Architecture demonstration system with a focus on the use of OpenTelemetry. It is based on the Online Boutique[3], maintained by Google to highlight their Google Cloud products. This demo application comprises more than 13 microservices, developed in 9 different programming languages, and utilises 2 types of inter-service communication protocols (HTTP and gRPC).

[1] https://github.com/filibuster-testing
[2] https://github.com/open-telemetry/opentelemetry-demo
[3] https://github.com/GoogleCloudPlatform/microservices-demo

3. Train Ticket[4] is a benchmark system introduced in [31] which explicitly addresses the need for realistic and representative benchmark systems. This system has 41 microservices, developed in 4 different programming languages. One challenge with this system is that, due to its size, it is difficult to run on a single machine. However, the main challenge is that its tracing solution is based on Zipkin[5], instead of OpenTelemetry. While Zipkin allows context propagation, it does not allow context *state* propagation. Therefore, Reynard cannot be trivially adjusted to work with Zipkin, and thus we had to exclude the benchmark.
4. Another system we identified was Reynard itself. As the instrumentation is composed of a controller and multiple proxies that interact using API's, it can be classified as a Service-Oriented Distributed System.

As anticipated, our choice of benchmark systems was minimal, but still enabled us to explore the application of Reynard in systems with distinct configurations. We decided to use the existing Filibuster *corpus*, the Astromy Shop and Reynard itself as part of our evaluation. Additionally, we created a small number of our own (micro) benchmarks. Finally, we applied our tool to an existing industry system.

5.3. Applying Reynard to Benchmark Systems

To evaluate whether we can apply Reynard to existing systems with a reasonable integration effort (RQ1.1), we performed an empirical evaluation by applying Reynard to different existing systems. Additionally, this allows us to evaluate how efficient and effective Reynard is in the context of benchmark systems (RQ2 and RQ1.4). Furthermore, by empirically evaluating different test scenarios, each with a distinct call graph structure, we can assess their impact on the efficiency of Reynard (RQ2.2).

To this end, we applied Reynard to the Filibuster *corpus*, the OpenTelemetry Astromy Shop, Reynard itself and a micro benchmark. In the following sections, we describe the process required and the scenarios we included.

5.3.1. Applying to the Filibuster Corpus

Filibuster is an automated fault injection tool that simulates service-level and network-level failures using service-level instrumentation. One of the paper's contributions is a suite of benchmarks that encompass certain classes of interactions common in Microservices. Both the tool and the benchmarks ("corpus") are publicly available[6].

The Filibuster *corpus* enables us to compare Reynard to Filibuster, a state-of-the-art work (RQ2.1). Additionally, the *corpus* contains minimal cases that each highlight a common pattern in Service-Oriented Distributed Systems. This allows us to verify our design for these cases. In particular, we can verify if our identification (RQ1.3) and abstract model (RQ1.5) are comprehensive enough to handle the presented cases.

Our goal is to evaluate whether Reynard can efficiently visit the fault space compared to the state-of-the-art. As Filibuster applies service-level instrumentation, it can access runtime information that Reynard cannot. Furthermore, we include the same pruning policy introduced by Filibuster. Hence, we hypothesise that Filibuster performs as well as, or better than, Reynard in terms of reducing the search space if we do not include our retry reduction policy.

Configuring the Filibuster Corpus

We made a small number of changes to the corpus to ensure it functioned properly again. These are tracked in a fork of the corpus project[7].

To run Reynard on the Filibuster corpus, we created a different fork of the corpus[8], and changed the runtime configuration to enable Reynard to run. This process involved changing configuration variables and removing the original Filibuster instrumentation. To alleviate this process, we created a small script that could update existing Docker Compose configurations to use Reynard. The rest of the updates were

[4] https://github.com/FudanSELab/train-ticket
[5] https://zipkin.io/
[6] https://github.com/filibuster-testing
[7] https://github.com/delanoflipse/filibuster-corpus/pull/3
[8] https://github.com/delanoflipse/filibuster-corpus/pull/5

essentially a few replace-all operations. The pull request maintains the list of changes that have been made.

Configuring Filibuster
To make a fair comparison, we had to re-run Filibuster's results with an altered configuration. Specifically, we had to alter it to only simulate the network-level faults that Reynard can simulate. Furthermore, we had to run both tools on the same machine to ensure a fair comparison of the runtime duration. For reproducibility, we created a fork of Filibuster and recorded our changes in a branch[9].

As the Python implementation of Filibuster was developed four years prior to our project, we had to make minor adjustments to get the project running again. Specifically, a few Python dependencies were unversioned, and their newer versions were incompatible. We pinned the versions to the latest available at the time of creation. Additionally, a few issues related to the functionality of an older version of Docker Compose had to be resolved.

We ran into an issue with the larger benchmarks in the corpus. When we attempted to run them, we observed connection issues in the services and instrumentation. This resulted in inconsistent and incorrect results for the two largest benchmarks. After some effort, we determined that the issue was likely due to the exhaustion of available ephemeral TCP ports and resource exhaustion resulting from the instrumentation running as a single-threaded Python service. Our solution involved adjusting the instrumentation to reuse TCP connections by utilising a connection pool and running the instrumentation service with a multi-threaded WSGI tool. We assume these changes do not influence the tool's functionality and should only increase efficiency.

5.3.2. Applying to the OpenTelemetry Astronomy Shop

We had to update a small number of configuration files to run the *Astronomy Shop* with Reynard. We created a fork, and documented all changes in a branch[10] for reproducibility.

To identify test scenarios, we ran the application and searched for interactions that covered a substantial subset of the services or processed a significant number of requests. We identified three scenarios of interest:

1. A simple interaction between four services to determine the shipping costs, with data nondeterminism. This type of interaction, where only a small number of services are included, is common in Service-Oriented Distributed Systems, as services and interactions are decoupled. Furthermore, it is similar to the scale of integration tests, as mocking reduces the number of services involved.
2. An interaction with the recommendation service that issues asynchronous requests and randomised responses. This interaction was chosen as it represents a challenging case for consistent fault identification.
3. A complex interaction that simulates a webshop checkout. This interaction was chosen because it required a setup, covered a significant number of services, and represented a key interaction with the system.

5.3.3. Applying Reynard to Itself

As Reynard is in itself a Service-Oriented Distributed System, we included it as part of our benchmark systems. We created a test scenario of the faultload registration interaction. In this test scenario, the controller sends the faultload definition to each proxy asynchronously. If any call to the proxy fails, it is retried up to a configurable number of times. This is a challenging scenario for our fault space reduction, as it involves concurrent calls with retries that do not influence each other. To enable running Reynard on itself, we added OpenTelemetry instrumentation to the controller and the proxy.

5.3.4. Applying to a Microbenchmark

Based on observations from other systems, we created several microbenchmarks to mimic communication patterns that are challenging for Reynard to handle in a minimal setting. In the end, we created three microbenchmarks. The first covered all common resilience patterns within a single test scenario. The

[9] https://github.com/delanoflipse/filibuster-comparison/pull/1
[10] https://github.com/delanoflipse/opentelemetry-demo-ds-fit/pull/1

second was a test case to experiment with all possible exploration orders. The third highlighted a scenario that was a known limitation. Of these microbenchmarks, we included the first in our evaluation, as it allows us to compare our different optimisations.

We created the microbenchmarks using minimal services, implemented as minimal TypeScript programs with automatic OpenTelemetry instrumentation. Each service only has one interface, whose behaviour follows a specific pattern in the call graph. This way, we could *compose* a system from different minimal services.

5.4. Applying Reynard to an Industry System

To evaluate the effectiveness of Reynard in an industry system (RQ3), we performed a case study by applying Reynard to an existing industry system. Specifically, we applied Reynard on ASML's data analytics platform. This system is designed as a Microservice Architecture and has in the order of 50 to 100 (micro)services. We first addressed the engineering challenges of integrating Reynard's instrumentation into the system's runtime environment. As OpenTelemetry instrumentation was available for the *test environments* of the system, this simplified this process. Next, we identified two test scenarios that cover two key interactions with a specific service within the system, which cover a significant number of other services. We then ran the scenarios multiple times, with different parameters to collect our results.

5.5. Measuring the Overhead of Reynard

To evaluate the overhead introduced by instrumenting a SUT with Reynard (RQ2.3), we measured the impact of our instrumentation by running performance tests for several scenarios. We created a minimal Go web server that acts as the destination for the proxy. The web server exposes an endpoint that responds with status code 200 and the message "OK". This ensures that the receiving server is not the bottleneck in the performance test, and that the proxy dominates the measured overhead. Furthermore, we added two endpoints that mock an interaction with the controller. That way, we can re-use the web server to mock the controller, minimising the logic and overhead of the proxy-to-controller interaction.

We created the following benchmark scenarios:

1. Sending requests directly to the proxy destination, bypassing the proxy. This provides an estimate of how much of the overhead is caused by interacting with the proxied service.
2. Sending requests to the proxy without any instrumentation headers included. After inspecting the headers, the requests are directly forwarded to the proxied web server. This indicates the overhead of proxying the request without additional actions.
3. Sending requests to the proxy with tracing information for an unknown trace. This is similar to the previous test, as it should only forward the request to the proxied service.
4. Sending requests to the proxy that are related to a registered faultload, but for which no fault matches the interaction. This represents the most common overhead introduced, where the message is thoroughly inspected but still forwarded.
5. Sending requests to the proxy that match a preconfigured fault. This represents the overhead of simulating faults in the system. If a fault is injected, no message is forwarded, so this is not influenced by the overhead of the proxied web server.
6. Sending requests to the proxy that match a fault, at the end of a large faultload. This should indicate how much the fault matching procedure influences the proxy.

To assess the average latency and latency distribution, we used the commonly used benchmarking tool wrk[11]. We ran each scenario for 1 minute, which allows us to take the average of a significant number of interactions.

[11] `https://github.com/wg/wrk`

6

Evaluation Results

In this chapter, we present the results of our evaluation of Reynard, based on the methodology described in Chapter 5.

6.1. Comparing Reynard to Filibuster using the Filibuster Corpus

Table 6.1 shows the results of directly comparing Reynard with Filibuster for the benchmarks in the Filibuster corpus. We included every benchmark that was included in the original paper, and ran them with and without our retry reduction policy when relevant. The fault space columns indicate the number of fault injection points, and how many are conditional (in brackets). Based on this, we determine the size of the complete fault space, which consists of all combinations of fault injection points and 4 failure modes. For both Reynard and Filibuster, we denote the number of concrete test scenario executions, the number of test cases in the search uniquely pruned due to the service encapsulation reduction policy (SER) and the the total average runtime of the test suite over at least 10 executions of the complete test suite (excluding the overhead of starting and stopping each benchmark).

Table 6.1: Comparison of the search space exploration by Reynard and Filibuster and their respective runtimes.

		Fault space		Reynard (our tool)			Filibuster		
Benchmark	**Variant**	**Injection points**	**Fault space size**	**Explored**	**#SER**	**Time (s)**	**Explored**	**#SER**	**Time (s)**
cinema-1		2 (0)	25	9	0	0.8	9	0	1.0
cinema-2		2 (0)	25	8	1	0.7	8	1	1.0
cinema-3	normal	4 (2)	625	27	38	1.5	27	38	3.7
	w/ retry reduction	4 (2)	625	19	32	1.3	27	38	1.4
cinema-4		5 (0)	3 125	8	1	1.3	8	1	2.9
cinema-5		2 (0)	25	25	0	1.3	25	0	5.3
cinema-6		3 (1)	125	41	0	1.9	41	0	5.5
cinema-7		4 (1)	625	45	0	2.1	45	0	2.2
cinema-8	normal	2 (1)	25	21	0	1.2	21	0	2.2
	w/ retry reduction	2 (1)	25	9	0	0.8	21	0	2.2
expedia		2 (1)	25	21	0	1.2	21	0	2.2
audible		8 (0)	390 625	30	35	2.4	30	35	5.2
mailchimp		6 (2)	15 625	192	1	9.6	192	5	31.8
netflix	w/o bugs	9 (3)	1 953 125	2 440	1	463.9	2 440	1	644.8
	w/ bugs	10 (4)	9 765 625	2 444	9 473	580.9	2 444	65	887.9

Overall, Reynard covers the fault space in the same number of cases as Filibuster, with only a single deviation in the number of redundancies found. In cases where the retry reduction policy can be applied, we observe a lower number of test cases explored compared to Filibuster. Additionally, the runtime of Reynard is in the same order of magnitude as, but consistently lower than, Filibuster. The following sections detail each specific benchmark, highlighting its similarities and differences.

Cinema-1 Benchmark - Synchronous Sequential Calls

This benchmark represents a synchronous and sequential communication between multiple services. The call graph and search tree are shown in Figure 6.1. The complete fault space represents all possible combinations of failure modes for both injection points. However, a failure in the call to the *Bookings* service excludes a call to the *Movies* service. Hence, we do not have to test for the combination, but only for each individual fault injection point in the happy path. There are 4 failure modes, 2 fault injection points, and one test case for the happy path, resulting in a total of $2 \times 4 + 1 = 9$ test cases.

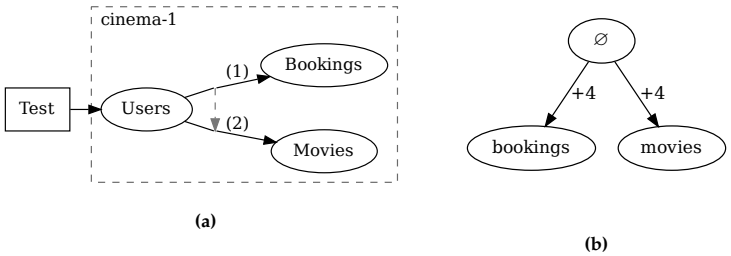


Figure 6.1: The call graph (a) for the "cinema-1" benchmark from the Filibuster corpus and the corresponding search tree (b) for Reynard.

Reynard can correctly detect the exclusion effect of faults f_{p_2} given $\{f_{p_1}\}$, and visits the search space in 9 test cases. Furthermore, it visits events originating from the same service in the order in which they are reported to the controller, starting with the event related to the *Bookings* service and then the *Movies* service. Filibuster does this in a reverse order due to its different exploration order, but ends up exploring the same search space.

Cinema-2 Benchmark - Nested Calls

This benchmark is similar to *cinema-1*, but instead of having calls originating from the same server, the calls are now done in a nested fashion. It represents a trivial case of a transitive causal dependency of events. The call graph and search tree are shown in Figure 6.2. As with *cinema-1*, there are only two events, given an equal complete fault space.

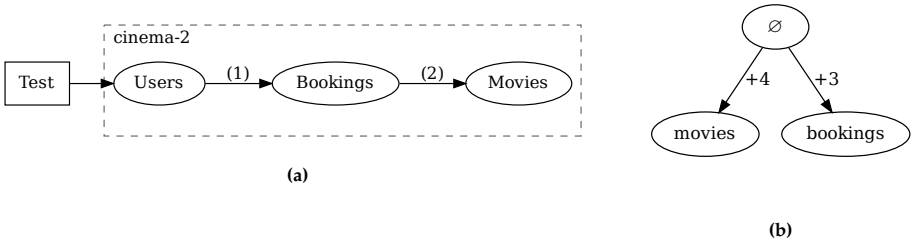


Figure 6.2: The call graph (a) for the "cinema-2" benchmark from the Filibuster corpus and the corresponding search tree (b) for Reynard.

Due to causality, a failure in the call to the *Bookings* service excludes a call to the *Movies* service. Hence, we do not have to test for their combination, resulting in a similar search tree. Reynard can correctly detect the causality dependency between faults f_{p_2} and f_{p_1} and infers that they cannot be combined.

Reynard, by default, visits fault injection points in the fault space in a depth-first post-order of the call graph. As a result, the exploration order is reversed when compared to *cinema-1*: faults at the *Movies* services are explored before those at the *Bookings* service. In this case, the search tree is equal to that explored by Filibuster.

Any failures at the *Movies* service result in the propagation of an error with a fixed status code at the *Bookings* service. This allows for a single case of reduction for the service encapsulation reduction policy,

as injecting that fault at p_1 yields the same behaviour, while omitting the call at p_2, thus revealing at most the same information. This service encapsulation reduction is only possible with a depth-first exploration of the call graph. In a breadth-first exploration, simulating the propagated fault at p_1 is scheduled before discovering that it is propagated by failures at p_2, preventing the reduction.

Cinema-3 Benchmark - Retry Resilience Pattern

This benchmark represents a common resilience pattern: retries. The call graph and search tree are shown in Figure 6.3. In this variation of the *cinema-2* benchmark, any failure at p_1 results in a single retry ($p_{1.1}$).

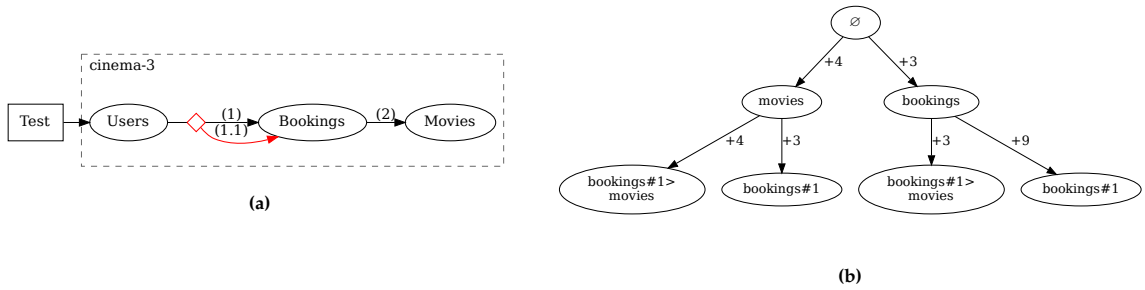


Figure 6.3: The call graph (a) for the "cinema-3" benchmark from the Filibuster corpus and the corresponding search tree (b) for Reynard.

The fault space comprises 4 fault injection points. In case of a retry, p_2 is repeated, but with a different cause ($p_{1.1}$ instead of p_1), hence resulting in a different identifier (which we denote as $p_{2\#1}$).

Similar to *cinema-2*, it can be directly inferred that p_1 or $p_{1.1}$ cannot be combined with p_2 or $p_{2\#1}$ respectively. Furthermore, any faults at p_1 or p_2 (due to fault propagation) cause the inclusion of $p_{1.1}$, which in turn causes $p_{2\#1}$. Hence, testing any failure mode at p_2 in combination with any fault in the path of $p_{1.1}$ results in the same behaviour as testing the propagated fault at p_1 in combination with any fault in the path of $p_{1.1}$. This is the reduction we see in the results. Finally, the same error propagation occurs at $p_{2\#1}$ as with p_2, allowing for a further reduction.

This scenario can be applied with the retry reduction policy. Figure 6.4 shows the search tree when this pruning policy is enabled. When injecting faults at p_2, this results in a transient fault at p_1. As such, we do not need to test the combination of faults at p_2 with faults at $p_{1.1}$, as we do not expect a difference in behaviour, except for more retries. This results in a reduction of 3 distinct cases. Additionally, we only test different failure modes for either transient or persistent faults at p_1, reducing the number of combinations to check from $(4 + 4 * 4) = 20$ to $4 + 4 = 8$ cases. The effective reduction in this case is 8 cases, due to overlap with other test cases and pruning policies. The retry reduction policy results in a 30% decrease in test cases to cover.

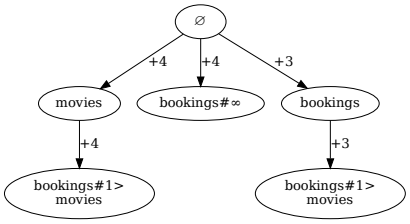


Figure 6.4: The search tree for the "cinema-3" benchmark when applying the retry reduction policy.

Cinema-4 Benchmark - External Dependencies

In this variation of the *cinema-2* benchmark, a call to an external service is issued before each call to other internal services. The call graph and search tree are shown in Figure 6.5.

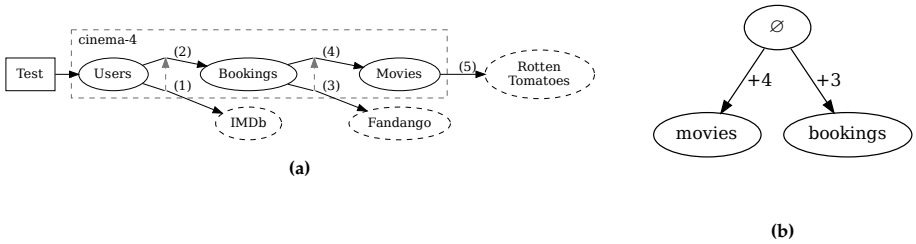


Figure 6.5: The call graph (a) for the "cinema-4" benchmark from the Filibuster corpus and the corresponding search tree (b) for Reynard.

This benchmark could not be updated to create an accurate comparison. Due to the way that Filibuster's instrumentation works, only application-level faults can be simulated for the external calls. But these failures are not the same as those simulated by Reynard. Reynard could simulate network-level failures by placing proxies in front of the external dependencies. However, this would be an inaccurate comparison, as Filibuster cannot simulate the same errors.

Leaving the benchmark as-is, the search space is essentially the same as the *cinema-2* benchmark, as can be seen in the results.

Cinema-5 Benchmark - Internal Fallback Resilience Pattern

In this variation of the *cinema-1* benchmark, the *Users* service can recover from partial failures by using an internal fallback value. As opposed to *cinema-1*, a failure in the *Bookings* service does not exclude a call to the *Movies* service. Therefore, their full combination has to be explored, resulting in a higher case number. The call graph and search tree are shown in Figure 6.6.

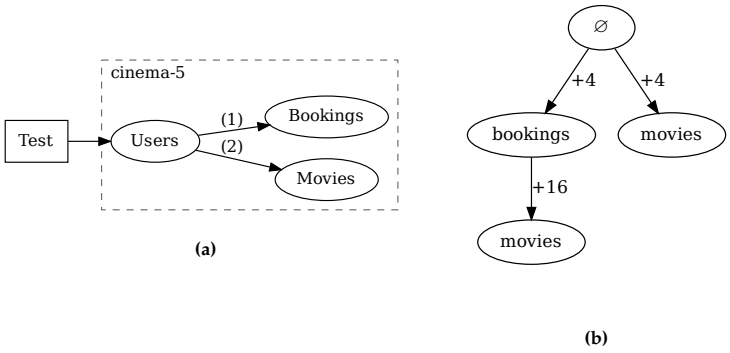


Figure 6.6: The call graph (a) for the "cinema-5" benchmark from the Filibuster corpus and the corresponding search tree (b) for Reynard.

Semantically, the call to either dependency is fault-tolerant, but we cannot assume that their combination is fault-tolerant. Conversely, we cannot assert that the composition of the failures at p_1 and p_2 results in no new behaviours. It might be that the *Users* service only allows for the failure of one of its dependencies. Without additional information available, we must cover all combinations to ensure completeness. This is reflected in the search tree of Figure 6.6b. There is no reduction possible, and all possible faultloads in the fault space have to be explored.

Cinema-6 Benchmark - External Fallback Resilience Pattern

In this variation of the *cinema-1* and *cinema-5* benchmarks, the *Users* service can recover from a fault in the *Bookings* service by using an external fallback: a replica. As the call to the secondary replicate is distinct, we cannot trivially infer that it has the same behaviour as the primary replica. To be complete, all combinations of faults for both the primary and secondary have to be tested. The call graph and search tree are shown in Figure 6.7.

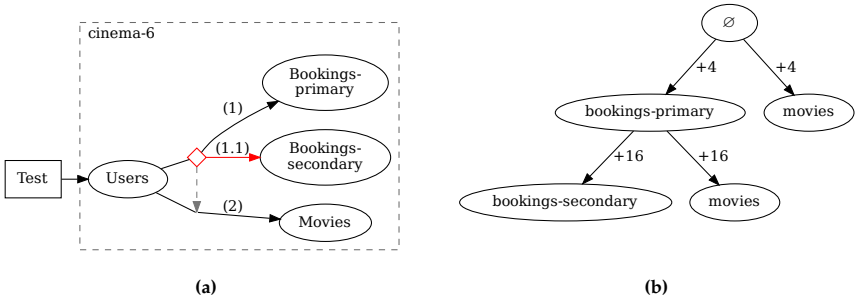


Figure 6.7: The call graph (a) for the "cinema-6" benchmark from the Filibuster corpus and the corresponding search tree (b) for Reynard.

Reynard correctly infers that if the calls to both the primary and secondary fail (at p_1 *and* $p_{1.1}$), this excludes a call to the *Movies* service. If both *Bookings* services were configured as the same service, we could apply our retry reduction policy to reduce the search space further.

Cinema-7 Benchmark - Healthcheck

In this variation of the *cinema-6* benchmark, a health check determines whether the primary or secondary is used to perform the request. The resulting search tree is similar to *cinema-6*, with the healtcheck replacing the call to the primary in the search tree. The call graph and search tree are shown in Figure 6.8.

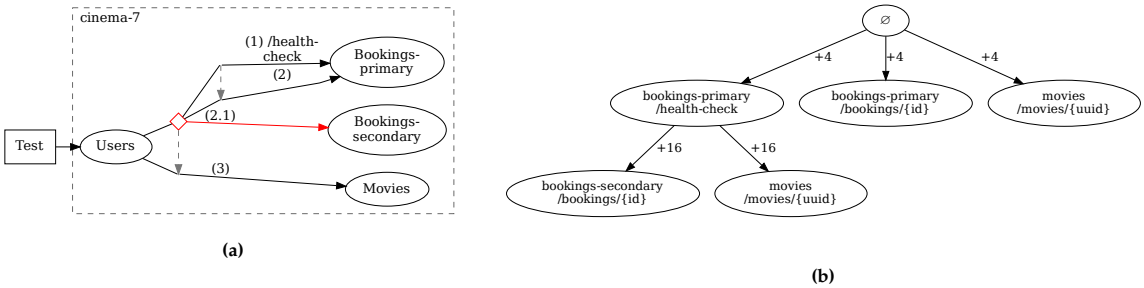


Figure 6.8: The call graph (a) for the "cinema-7" benchmark from the Filibuster corpus and the corresponding search tree (b) for Reynard.

Additionally, we must explore all failures where the health check passed, but directly after the primary fails. Luckily, the implementation of this benchmark does not have a TOCTTOU (Time Of Check To Time Of Use) bug as classified by [17]. The *User* service includes error handling for the scenario where the health check succeeds but the *Bookings Primary* service is unavailable when it is invoked.

Cinema-8 Benchmark - Monolith and Retries

This benchmark, as represented in Figure 6.9, corresponds to a system structure where a single server exposes a monolithic service (one that encompasses all functionality). If a call to the monolith fails, a retry is issued. Since these are the only calls, the fault space comprises only faults injected in the first call, or both the first and second call. As expected from the preceding benchmarks, Reynard can infer the dependency of the retry event on the first failing event.

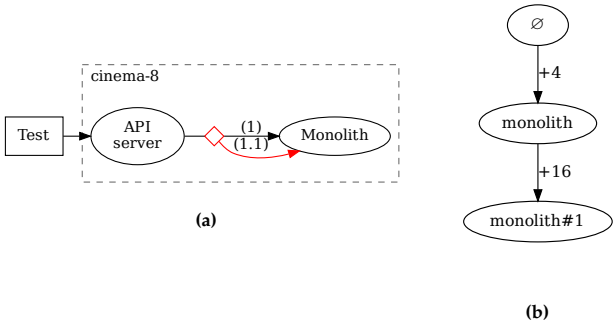


Figure 6.9: The call graph (a) for the "cinema-8" benchmark from the Filibuster corpus and the corresponding search tree (b) for Reynard.

Similar to *cinema-3*, we can apply the retry reduction policy to reduce the search space to transient and persistent failures. As can be seen in Figure 6.10, this reduces the search subspace related to the retries from $4 + 4 * 4 = 20$ to $4 + 4 = 8$, a 60% reduction.

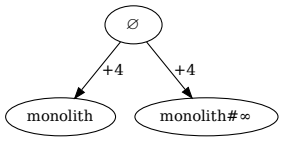


Figure 6.10: The search tree for the "cinema-8" benchmark when applying the retry reduction policy.

Expedia Benchmark

The *Expedia* benchmark represents an industry implementation of an external fallback. Aside from variation of the same example system, the Filibuster corpus includes recreations of bugs or resilience tests presented by the industry. The call graph and search tree are shown in Figure 6.11. Specifically, it represents a scenario in which a service employs a machine learning algorithm for sorting reviews. In the event of a failure, another service is called that provides reviews sorted in reverse chronological order. As these are distinct services, we must test for all combinations of possible failures. Reynard correctly infers the inclusion effect for these faults and checks for the combination of both fault injection points.

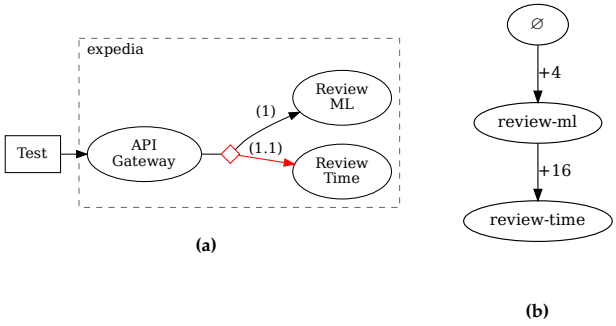


Figure 6.11: The call graph (a) for the "expedia" benchmark from the Filibuster corpus and the corresponding search tree (b) for Reynard.

This benchmark scenario is designed to illustrate how organisations assess their resilience patterns.

In the test written for Filibuster, an assertion checks that if a failure is inserted, but the response is a status code 200, the response should match that of the *Review Time* service. With Reynard, we can assert properties about the structure of the call graph. Thus, we can explicitly check for the property that either there was no failure simulated at the *Review ML* service, and no request to the *Review Time* service is part of the call graph, or it was and there is a request.

Our automatic oracle detects that the *API Gateway* responds with a 503 error if both services are unavailable. This should not lead to a state divergence bug, as both endpoints are read-only and therefore idempotent. Yet, the error code is still semantically incorrect, as the *API Gateway* is available.

Audible Benchmark

The *audible* benchmark represents a system with both synchronous and nested calls. The benchmark system has no resilience patterns. Furthermore, all subsequent calls are causally dependent on previous calls, with failures excluding all subsequent calls. The call graph and search tree are shown in Figure 6.12.

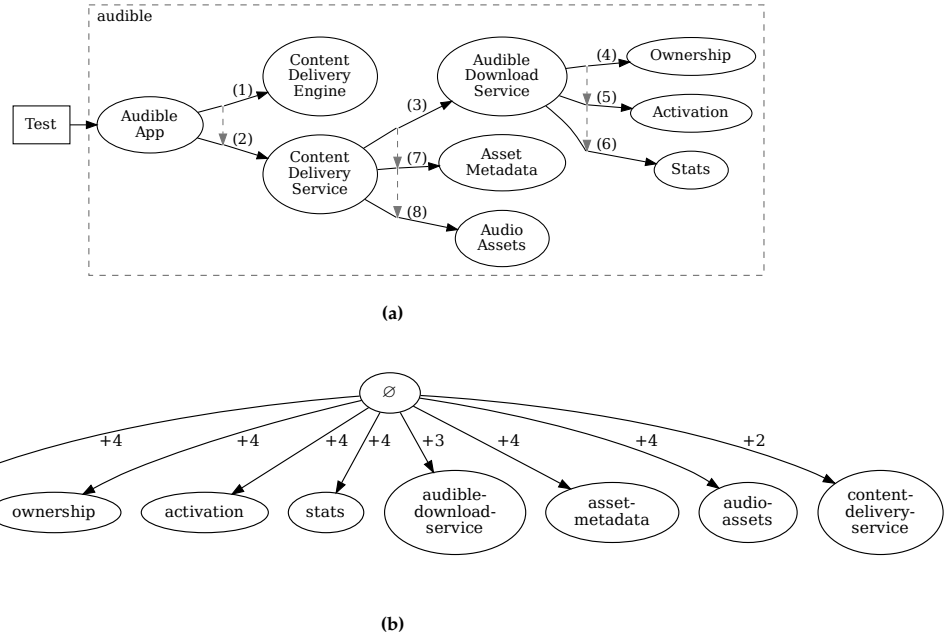


Figure 6.12: The call graph (a) for the "audible" benchmark from the Filibuster corpus and the corresponding search tree (b) for Reynard.

As such, the search space is drastically reduced; only single failures have to be explored. Furthermore, the nested structure and the propagation of errors result in the reduction of many cases due to the service encapsulation reduction policy. Calls that depend on the response of the *Audible Download Service* do not have to be tested in combination with any dependencies of the call to the *Content Delivery Service*. Only the failure modes of the direct call to the *Audible Download Service* have to be tested. Additionally, because faults at the *Audible Download Service* propagate to the *Content Delivery Service*, we can omit testing these propagated faults.

Mailchimp Benchmark

The *mailchimp* benchmark represents a more complex system interaction. The *App Server* performs most of the actions. A first call to the *Requestmapper* is a causal dependent of all subsequent events. The intention of the *App Server* is to read and subsequently write to a database, of which a secondary exists. If the read or write operation fails on the primary, the read and write operations are performed on the secondary (after the read and write operations to the primary). For the secondary, if the read fails, the write is aborted. The call graph and search tree are shown in Figure 6.13.

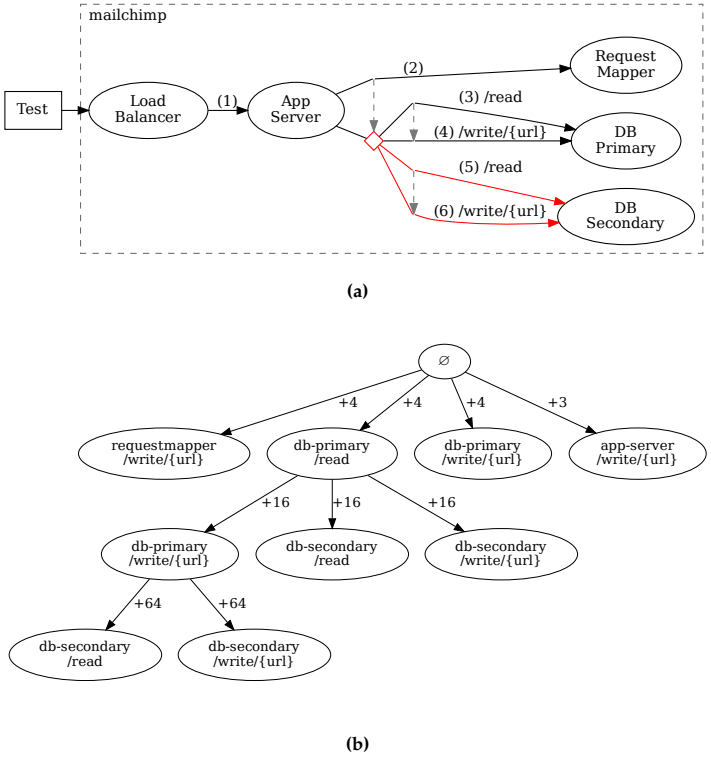


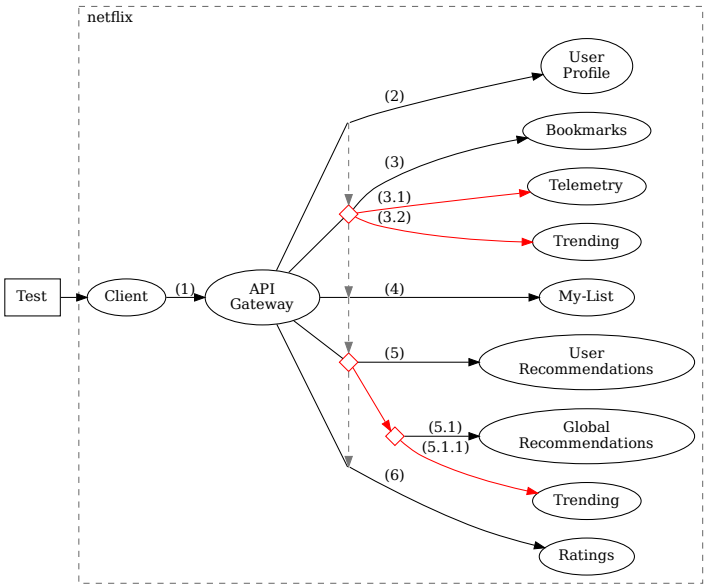
Figure 6.13: The call graph (a) for the "mailchimp" benchmark from the Filibuster corpus and the corresponding search tree (b) for Reynard.

This interaction is reflected in the search tree, allowing faults to be injected into the primary's read and write requests simultaneously, and subsequently into the secondary's read or write requests. In the event of a failure, the error is propagated from the *App Server* to the *Load Balancer*, enabling the reduction of a single test case.

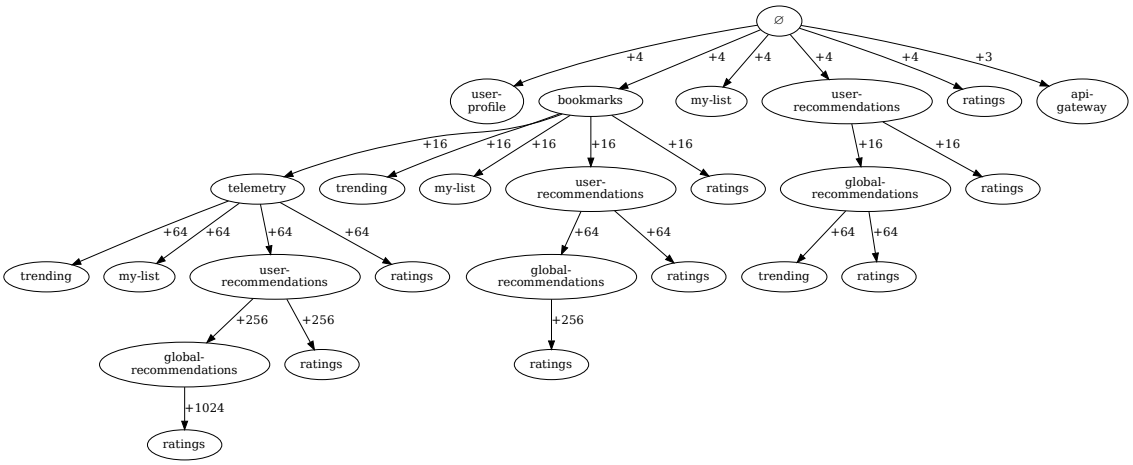
It is unclear whether, semantically, it was intentional that a failure of the read call should still result in a write call in the primary. This deviation in how the primary and secondary are used might be unintentional. Nonetheless, given these semantics, we can correctly detect all distinct cases.

Netflix Benchmark

The *netflix* benchmark is the most complex benchmark in the corpus. It represents a wide (fanout) call graph, with multiple fallback mechanisms. The call graph and search tree are shown in Figure 6.14.



(a)



(b)

Figure 6.14: The call graph (a) for the "netflix" benchmark from the Filibuster corpus and the corresponding search tree (b) for Reynard.

Most significantly, many events can fail together, without influencing other events. The most impactful is the combination of failing requests to the *Ratings*, *Bookmarks*, *User Recommendations*, *Telemetry* and *Global Recommendations* together, which result in 1024 test cases. By failing the request to the *Bookmarks* service, the *Telemetry* and *Trending* services are called as a fallback. Faults in the call to the *Telemetry* service do not influence the outcome, but the *Trending* service call must succeed. The fallback for the *User Recommendations* service is the *Global Recommendations* service, whose fallback is the *Trending* service (too). If the latter is without failure, we can finally simulate failures at the *Ratings* service, resulting in an erroneous response. As we must test for all combinations of these failures, this results in $4^5 = 1024$ combinations to be complete.

An additional challenge is the two distinct calls to the *Trending* service. They happen for different

reasons (as a fallback for different calls). Without using the predecessors field in the fault identification, we can incorrectly infer exclusion and inclusion effects. For example, the failure in the call to the *Trending* service, serving as a fallback for the *Bookmarks* service, results in excluding a call to the *User Recommendations* service. But, the call to the *Trending* service, as a fallback for the *Global Recommendations* service, depends on the failure of a call to the *User Recommendations* service. If the goal is to infer the behaviour of the faults at the *Bookmarks*, *User Recommendations*, *Global Recommendations*, *Trending* (in that order), then we might incorrectly infer that the call to the *User Recommendations* service is excluded and incorrectly prune the faultload. Thus, this scenario requires the predecessors field to differentiate the events correctly and infer the system behaviour. Note that this works in this specific case because there are no asynchronous calls. If the calls to the *Trending* service had no reliance on each other, and their paths were asynchronous functions, then this identification would not work and Reynard would not be complete, while Filibuster could be.

Netflix Benchmark with Faults

The corpus includes a variation of the *netflix* benchmark, where the *User Profile* service calls the *Telemetry* service. Faults injected in the request to the *Telemetry* service do not influence the behaviour of the *User Profile*. As such, we only have to additionally visit the 4 faults corresponding to all 4 failure modes for this additional fault injection point. Figure 6.15 shows the new search tree, with one additional node.

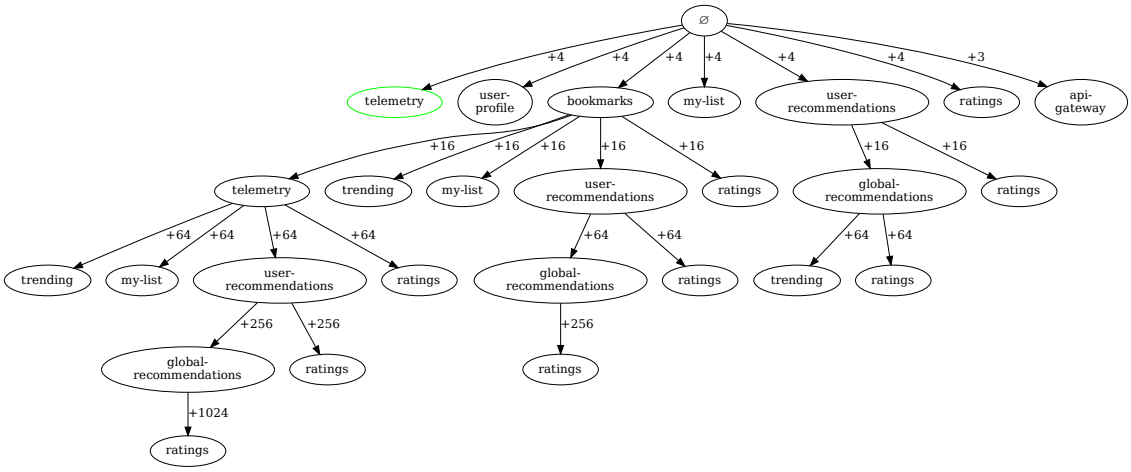


Figure 6.15: The search tree for the "netflix" benchmark with faults from the Filibuster corpus. Highlighted in green is the difference in the search tree.

In the results, we see a difference in the number of cases that can be uniquely pruned due to the service encapsulation reduction policy, when compared to Filibuster. After investigation, we found this is due to the different exploration order that Filibuster takes. Specifically, the reverse order of child nodes and depth-first exploration of the search tree means that the call to the *Telemetry* service from the *User Profile* is explored as one of the last nodes in the search tree. As this call does not influence the system's behaviour, any subsequent combination with this call is deemed redundant. However, because Reynard considers the fault injection points in another order, there are more cases to consider that are all redundant (for the same reason). This explains the difference in reduction cases.

We found that visiting the search tree in a depth-first order does result in the same test cases *and* reductions due to the service encapsulation reduction policy. However, the visited faultloads differed, as the new fault injection point was included in nearly all visited nodes. This is less ideal, as we want to find the minimal faultload that causes incorrect behaviour.

6.1.1. Discussion

Looking at the results of our comparison to Filibuster, we can conclude that Reynard can infer the same information that Filibuster can in the specific cases included in the Filibuster *Corpus*. These scenarios

cover frequently seen interaction and resilience patterns seen in Service-Oriented Distributed Systems.

While this holds for the scenarios in the Filibuster *Corpus*, we cannot conclude that this holds for all systems. As noted for the *netflix* benchmark, an adjusted benchmark could prove Reynard unsound, while Filibuster can remain sound. This is to be expected, as Filibuster has application-level metadata available to differentiate fault injection points more accurately.

We believe that our dynamic model could remain sound if Reynard could gain access to this information. The main limitation is not the abstraction or model, but the access to metadata. Still, Reynard can be complete for many different Service-Oriented Distributed Systems that keep within the known limitations.

Additionally, we find that Reynard can run in the same order of time as Filibuster, with Reynard consequently running faster. As both are proof-of-concepts, numerous opportunities exist to reduce runtime, so we do not consider this a significant benefit. Specifically, it can be reasoned that this is largely due to Filibuster using a Python runtime, whereas Reynard uses Java and Go. A more fair comparison in runtime would be to use Filibuster's Java implementation[1], but this cannot be trivially applied to the corpus.

More impactful is the effect of the retry reduction policy, resulting in at least a 30% decrease in cases for benchmarks with retries. A benefit of Reynard is that it can be applied in combination with the service encapsulation reduction. This would be more complex to implement in Filibuster's Python implementation.

Influence of the Call Graph on the Search Space

When comparing the call graphs of the benchmarks and their effect on the search space, we find that the largest combination of independent interactions with the same direct causal dependency dominates the search tree. That is, the events with the same causal dependency that can be feasibly combined when exposed to partial failures. This is evident in the *netflix* benchmark, where multiple calls originating from the *API Gateway* are unaffected by partial failures.

Conversely, the search space is significantly reduced in the case of the *audible* benchmark. This is due to two differences. First, having a more nested structure results in fewer fault injection points for the same causal dependency. Second, all events depend on other events, preventing them from being combined. If all events were dependent in the case of the *netflix* benchmarks, then the search space would still be significantly smaller, even with a "wider" call graph.

Meiklejohn [32] classifies causal dependencies as hard and soft dependencies. This definition is based on whether failures in dependencies cause a failure to propagate to their causal dependent event. We believe a similar distinction holds for events that prevent the continuation of related events with the same dependency. As such, we state that, using the service encapsulation reduction, the search space is dominated by the largest set of soft dependencies with the same direct causal dependency.

6.2. Applying Reynard to Other Benchmark Systems

Table 6.2 lists the results of applying Reynard to the other benchmark systems, as described in Section 5.3. In the table, the fault space columns denote the complete fault space when considering all combinations of fault injection points and 4 failure modes. The Reynard columns represent the exact number of cases visited by the tool, the runtime per benchmark scenario and the average runtime per test scenario execution. For each benchmark variant, the reported time represents the average runtime of at least 10 test suite executions.

We compared the results of each run to validate their consistency. The test cases explored are consistent, with one logical exception. Benchmarks with scheduling nondeterminism have inconsistencies in the order of the search space exploration, as this depends on the ordering of the call graph. In the following sections, we provide further details on these results. Furthermore, we discuss the effort required to instrument the system to work with Reynard.

[1] `https://github.com/filibuster-testing/filibuster-java-instrumentation`

Table 6.2: The search space and runtime of applying Reynard to benchmark systems.

			Fault space		Reynard		
System	Benchmark	Variant	Injection points	Fault space size	Explored	Time (s)	Time per test (ms)
Astronomy Shop	Shipping		4 (0)	625	15	1.3	87.0
	Recommendations		7 (0)	78 125	635	35.3	50.3
		Custom pruner	7 (0)	78 125	31	2.3	75.4
	Checkout		13 (0)	1 220 703 125	65	8.2	126.9
		w/ predecessors field	13 (0)	1 220 703 125	65	9.2	141.6
Microbenchmark	Resilience patterns		7 (2)	78 125	548	20.8	38.0
		w/ predecessors field	7 (2)	78 125	555	22.5	40.6
		w/ retry reduction	7 (2)	78 125	536	21.0	39.2
		w/ predecessors field and retry reduction	7 (2)	78 125	543	22.3	41.1
Reynard	Register (3 proxies)	0 retries	3 (0)	125	125	3.3	26.2
		2 retries, w/ retry reduction	9 (6)	1 953 125	729	23.1	31.6
		4 retries, w/ retry reduction	15 (12)	30 517 578 125	729	29.3	40.2

6.2.1. Applying Reynard to the OpenTelemetry Astronomy Shop

Applying Reynard to the OpenTelemetry Astronomy Shop took little effort. The demonstration application was preconfigured with OpenTelemetry, which reduced instrumentation effort. What remained was to adjust the runtime configuration to enable Reynard to work.

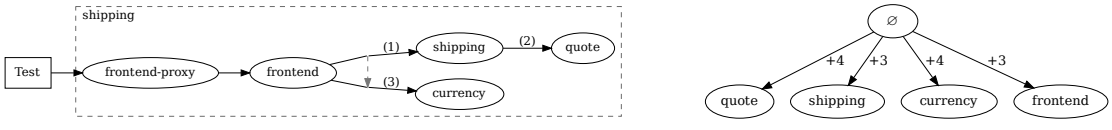
One issue we ran into when running the benchmarks was that the gateway proxy used in the demonstration application has a built-in limit on the number of concurrent connections. As we reuse the connection in the test scenario and quickly send numerous requests, we hit a limit in the proxy. Therefore, we had to update this configuration to enable our test suite to work. In theory, we could bypass the gateway proxy; however, to provide a more comprehensive test, we opted to test the whole interaction end-to-end.

In the following sections, we detail each test scenario in the demo application to which we applied Reynard.

Shipping Scenario

Getting the shipping cost of an order in the Astronomy Shop requires four synchronous calls. The implementation of Astronomy Shop generates random values for the shipping cost, which are further used in subsequent requests. Therefore, this scenario is subject to data nondeterminism. However, if we disable the payload as part of the fault identification, we can still uniquely identify each request. This can be done through a configuration parameter in the test decorator.

Figure 6.16 displays the call graph and search tree for this interaction. All calls are synchronous, and any failure results in a direct failure propagation to the upstream service. As a result, we only need to visit faultloads of size 1 to cover the whole fault space, as any subsequent event is excluded on a failure. Additionally, as failures are propagated upstream, we can omit two cases (for the shipping and frontend service). All cases can be covered in under 2 seconds, with each test case taking around 90 milliseconds to complete. The overhead of starting and stopping the application is significantly larger than the time it takes to run the benchmark scenario.



(a) The call graph for the "shipping" scenario.

(b) The search tree explored by Reynard.

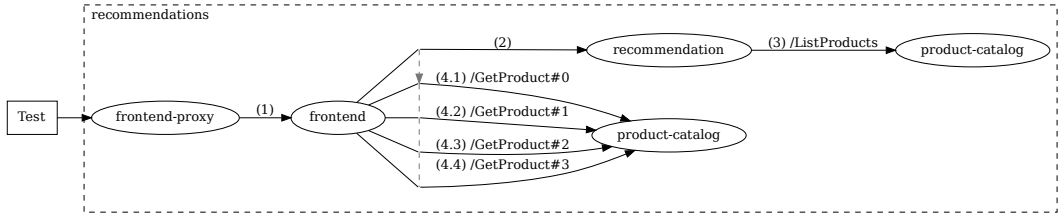
Figure 6.16: The call graph for the "shipping" scenario in the OpenTelemetry Astronomy Shop and corresponding search tree explored by Reynard.

Recommendations Scenario

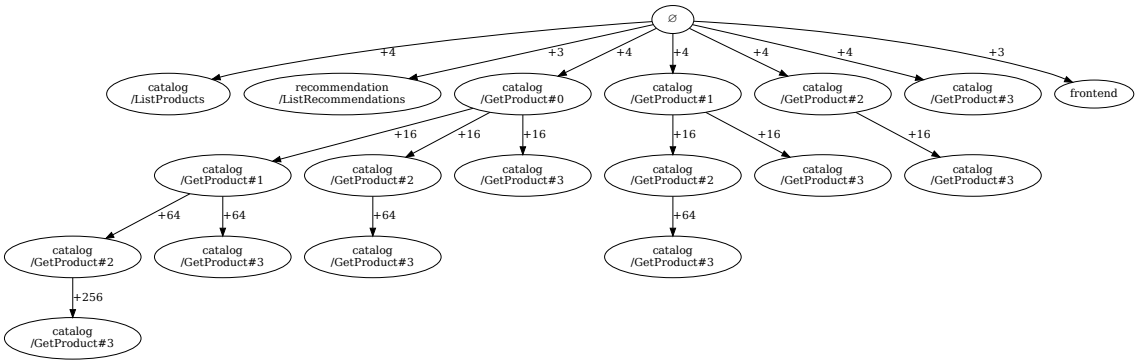
To provide product recommendations, the *recommendation* service first retrieves a list of products, selects four recommended products, and sends their identifiers to the *frontend*. The *frontend* then asynchronously requests the full product data for each of the four products. Figure 6.17a displays the call graph for this interaction.

The recommendation algorithm creates data nondeterminism for the test scenario. We can make the fault injection points consistent by disregarding the payload. However, this makes the four requests to get the product data indistinguishable. As such, there are four asynchronous requests with the same identifier (up to invocation count) in the happy path that do not affect their subsequent calls. As we discussed in Section 4.7, this can be problematic. If there is scheduling nondeterminism, then the n-th call does not always correspond to the n-th product in the list. As the frontend is written in JavaScript, we can be sure that each request is issued in order (as the JavaScript runtime is single-threaded), but no guarantees can be given about the order of responses.

To explore the complete fault space, we must test all combinations of all failures for each request. Even with data and scheduling nondeterminism, this is semantically correct: We want to investigate whether any specific combination of failures causes unexpected behaviour in the SUT. Visible are the nested combinations of product requests, which are the most significant contributors to the fault space.



(a) The call graph for the "recommendations" scenario.



(b) The search tree explored by Reynard.

Figure 6.17: The call graph for the "recommendations" scenario in the OpenTelemetry Astronomy Shop and corresponding search tree explored by Reynard.

With expert knowledge, we can define a pruning strategy similar to the retry reduction policy, where we either try only one fault or all faults for the asynchronous calls to the product catalogue. As we can see from the results, this drastically reduces the search space, from 625 to just 31 test cases. This results in a search tree, as presented in Figure 6.18, and a runtime that is an order of magnitude faster. Of course, this requires manual effort by the developer to reduce the search space, which can be prone to errors.

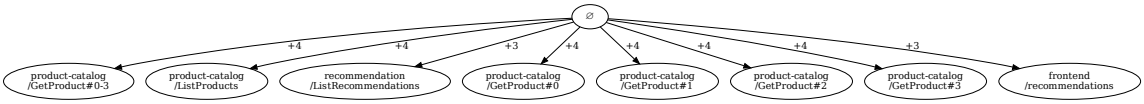
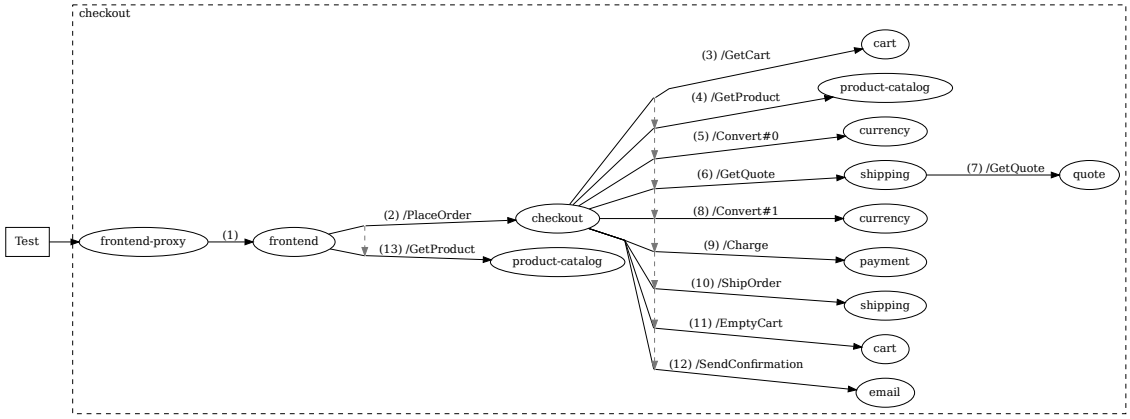


Figure 6.18: The search tree for the "recommendations" scenario with a custom pruning policy.

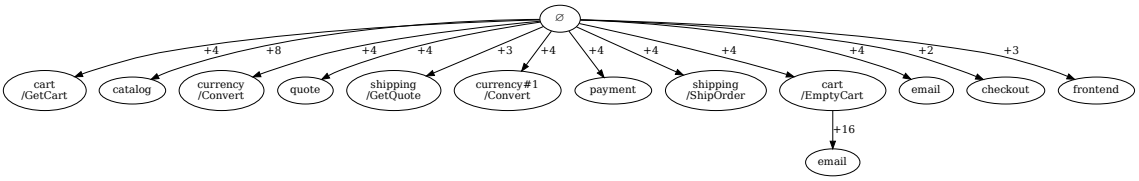
Checkout Scenario

This interaction is representative of complex interactions within a Service-Oriented Distributed System. The Astronomy Shop has no built-in resilience patterns, but the complete full search space still encompasses over a billion cases. However, this webshop interaction primarily involves synchronous, dependent, and nested calls. Hence, the fault space can be significantly reduced, with only 65 cases, which can be run in about 8 seconds.

Figure 6.19 illustrates the call graph and search tree for this interaction. We can see the flat structure of the search space, corresponding to the synchronous calls. Furthermore, we can see from the search tree that the response to the call to empty the user cart is ignored, allowing it to be combined with the next sequential interaction.



(a) The call graph for the "checkout" scenario.



(b) The search tree explored by Reynard.

Figure 6.19: The call graph for the "checkout" scenario in the OpenTelemetry Astronomy Shop and corresponding search tree explored by Reynard.

If this were a real system, it could contain a number of latent bugs, depending on the intended semantics of the services. For example, what should happen if the user's card is charged, but the order shipment service fails? Or, what should happen if no confirmation email is sent? And what if the cart is not emptied?

Furthermore, our automatic test oracle detected that the checkout service emits a 503 ("Service Unavailable") error code when the order shipment request fails. If the frontend were instructed to perform a retry on a 503 (a valid assumption), the user's card could be charged twice, and no products

would ever be shipped.

6.2.2. Applying Reynard to Itself

To enable Reynard to be applied to itself, we had to add OpenTelemetry instrumentation to its controller and proxy service. Both are implemented in Go, so we followed the tutorial provided by OpenTelemetry to add the instrumentation.

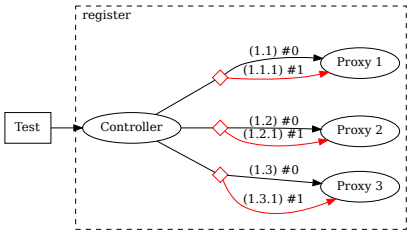
Instrumentation in Go requires some effort, as automatic instrumentation is still a work in progress. We had to copy over a few lines of code from the tutorial and adjust the configuration parameters to suit our purposes. Additionally, we had to manually instrument the HTTP client with OTel. All in all, this required a non-zero but feasible effort.

To run the test scenario, we used Reynard in an integration test setup using Testcontainers[2]. This enables the test to run in a CI/CD environment, as Testcontainers manages the task of starting and stopping the services. As one of our earlier implementations focused on this type of setup, Reynard includes helper functions to automatically instrument services in a Testcontainers setup, simplifying the setup.

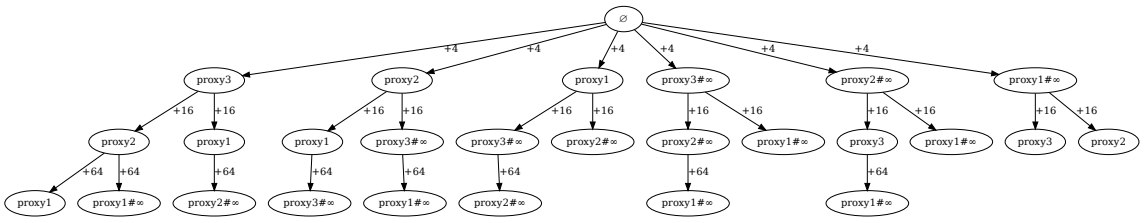
Registering Faultloads

Our fault injection tool is a Service-Oriented Distributed System itself, and therefore we can verify its resilience patterns. In particular, to ensure faults are correctly registered at all proxies, the controller applies retries if it cannot reach them. The same network instabilities that can cause network faults can influence the test environment.

Figure 6.20 displays the search tree for the faultload registration of one controller and three proxies with the retry reduction policy enabled. This scenario is generally challenging, as the fault space cannot be significantly reduced as the events are unrelated. When examining the search space explored for this scenario without retries, we must cover *all* combinations of faults; there is *no* reduction possible. Additionally, the events are prone to scheduling nondeterminism.



(a) The call graph for the "register" scenario with one retry.



(b) The search tree explored by Reynard.

Figure 6.20: The call graph for the "register" scenario with one retry in Reynard and corresponding search tree explored by Reynard.

We *had* to enable the retry reduction policy to run the test in a feasible amount of time. The complete search space without it is equal to $(1 + \sum_{i=1}^{r} m^i)^n$, where r is the number of retries, m the number of failure modes and n the number of proxies. For 4 retries, this becomes $(1 + 4 + 4^2 + 4^3 + 4^4)^3 = 39\,651\,821$

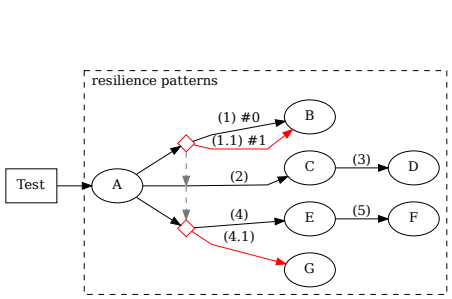
[2] `https://testcontainers.com/`

cases, which would take over 18 days to complete. Using the retry reduction policy, this can be reduced to just 729 cases, as it reduces the search space to $(1 + 2m)^n$ cases, which take around half a minute to complete. The average runtime of each test case (including up to n retries for each request) takes around 40 milliseconds to complete, which consists of the overhead registering and unregistering the faultloads in the auxiliary controller and four proxies (one for each proxy and one for the controller) and the overhead of their communication for fault injection.

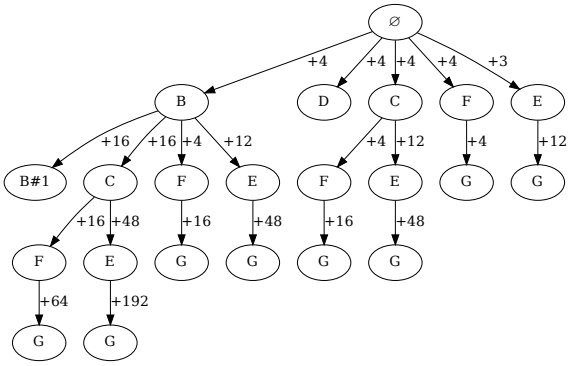
6.2.3. Applying Reynard to a Microbenchmark

We created a minimal system that incorporates several common resilience patterns, including retries, external fallbacks, internal fallbacks, and error propagation. All services in this system are auto-instrumented with OpenTelemetry, and the runtime is adjusted using the helper functions for Testcontainers available in Reynard, resulting in a minimal instrumentation effort.

Figure 6.21 illustrates the call graph and corresponding search tree explored by Reynard for the variant without the predecessors field and without the retry reduction policy. For this test scenario, a failed request to service B results in a retry. Faults in the requests to both C and D are resolved by using a default response value, and faults does not influence other calls. Faults in the request from E to F cause a propagated erroneous response to A. If either the call from E or F fails, a call to G is performed as an external fallback.



(a) The call graph for the "resilience patterns" scenario.



(b) The search tree explored by Reynard.

Figure 6.21: The call graph for the "resilience patterns" scenario in our microbenchmark and corresponding search tree explored by Reynard.

We ran this scenario in four variations: with the default configuration, with the predecessors field, with the retry reduction policy, and with both. The retry reduction decreases the number of test cases explored due to the retry to service B, regardless of the predecessors field. Adding the predecessors field increases the number of test cases, as the retry changes the predecessors field for the subsequent events. As a result, equivalent calls are considered distinct events. In this case, the predecessors field is not required and is detrimental to the efficiency of Reynard.

The search space is dominated by the combination of fault injection points that have fallbacks. Because the call to service C does not exclude any subsequent calls, this significantly increases the search space.

6.2.4. Discussion

As can be seen from the results, it is feasible to exhaustively cover the relevant fault space in a time comparable with the overhead of starting and stopping the required services.

However, it should be noted that these are idealised systems that have little additional functionality. This results in the low runtime per test scenario. We expect real systems to have a significantly higher runtime per test case. We explore this further in Section 6.3.

Additionally, we find that Reynard is practical to use in existing systems. Applying it to the OpenTelemetry

Astronomy Shop system, which incorporates 9 programming languages and 2 protocols, highlights that it can be used in a wide variety of systems. Similarly, setting up the instrumentation for Reynard to work on itself is a relatively small effort, and the same applies to our microbenchmark. Furthermore, the ability to run tests using Testcontainers demonstrates the flexibility of this approach, enabling testing at earlier stages of development in an automated manner.

Configuring both OTel and setting up the runtime for Reynard requires a non-zero but reasonable effort, which is flexible enough to be applied in various use cases. Furthermore, it is not a requirement that every service must be instrumented. Although it is required for Reynard to be complete, it is possible to adopt the instrumentation incrementally. Reynard ignores a service with no instrumentation or proxy configured.

6.3. Applying Reynard to an Industry System

Table 6.3 presents the results of applying Reynard to an industry system for two scenarios, each with different variants. For each test scenario, we denote the number of fault injection points involved and the number of fault injection points that only appear conditionally due to injected faults. Based on this, we state the size of the complete fault space, which represents all combinations of fault injection points and 4 failure modes. Next, we state how many test cases were executed by Reynard to cover the fault space exhaustively. Furthermore, we state the average time for the complete test suite and the average time per test execution. The runtime is the average of only 2 executions of the test suite, due to the significant runtime.

Table 6.3: The search space and runtime of applying Reynard to an industry system.

		Fault Space		Reynard		
Scenario	**Variant**	**Injection points**	**Fault space size**	**Explored**	**Time (s)**	**Time per test (ms)**
Scenario 1	no optimizations	18 (9)	3 814 697 265 625	517	1 664.2	3 218.9
	w/ retry reduction	18 (9)	3 814 697 265 625	493	1 620.8	3 287.6
	w/ predecessors field	18 (9)	3 814 697 265 625	601	2 068.4	3 441.6
	w/ predecessors field and retry reduction	18 (9)	3 814 697 265 625	594	2 083.3	3 507.2
Scenario 2	w/ retry reduction	20 (10)	95 367 431 640 625	378	1 218.3	3 223.0
	w/ predecessors field and retry reduction	16 (8)	152 587 890 625	509	2 001.9	3 933.1

Scenario 1

The first scenario is a create operation that involves 6 services and 8 endpoints. For 6 interactions, a single retry is attempted for a specific failure modes. With subsequent calls on these retries, this results in 9 conditional fault injection points. The complete search space, comprising all combinations of faults, is in the trillions of cases; however, with our reduction policies, this is significantly reduced. We ran this scenario with four variants: the combinations of enabling the retry reduction and including the predecessors field as part of the identifiers.

We find that the retry reduction policy reduces the search space, but to a lesser extent than in other scenarios observed in the benchmark systems. This is due to retries only being performed on specific status codes, whereas in the benchmarks, they often occurred on *all* failure modes.

Furthermore, we find that including the predecessors field in the identifiers results in *more* test cases being explored. This is similar to what we saw in the microbenchmarks. The retries influence the predecessors field, which prevents us from correlating information observed about equivalent fault injection points. Hence, we see an increase in the number of test cases. Additionally, there is an increase in runtime per test case, as predecessors field make the analysis computationally more expensive.

In this scenario, we do not *need* to include the predecessors field, as there are no indistinguishable interactions with the same identifier, unlike in the *netflix* benchmark. However, this could only be determined by manually inspecting the semantics of the interactions. It can be safer to include it when running a new test suite.

Scenario 2
The second scenario is an update operation that involves 5 services and 9 endpoints. For the first interaction, a single retry is attempted when it fails with a specific failure mode. After this first interaction, the service publishes an Apache Kafka[3] event. The other interactions are part of the consumer of the Kafka message, which, on failure, are all automatically retried.

We do not intercept the interaction where an event is published to Kafka, as Kafka uses a proprietary binary protocol. As a result, no new parent span is created. This leads to the interactions being directly related to the root interaction, rather than a new interaction. As a result, we need the predecessors field to accurately identify events, as otherwise we incorrectly correlate different events.

Kafka is an *asynchronous* message system. But Reynard only waits for the response of the root interaction before collecting reports from the instrumentation. This means that we cannot be certain that all events are completed before we collect the reports. As a result, this could make the test execution flaky. In our case, we found that it consistently performed the same number of retries within the time window of the test execution. Therefore, this is not an ideal test scenario, but we have included it because it still allows us to investigate interesting combinations of faults.

This scenario benefits from the retry reduction due to the retries on any failure of events in the Kafka message consumer. We could not run the test suite without it, as it would take an excessive amount of time.

Similar to scenario 1, including the predecessors field in the identifier increases the number of test cases explored, as well as the average time per test. Yet, we do observe that the fault injection is more precise, as we can correctly identify the fault injection points. In this case, it is beneficial to include the predecessors field because we can be certain that we do not accidentally omit test cases. At the same time, we visit potentially redundant test cases, but we cannot omit them due to the lack of correlated information.

Higher Runtime per Test
In general, we find that the runtime per test is an order of magnitude higher than in the idealised benchmark systems. More functionality, complex computations, and database calls result in longer runtimes. Moreover, multiple requests are required to bring the system into the right state before a test case can be executed. In particular, 4 and 5 requests were required for scenarios 1 and 2, respectively, to get the system in the right state. Additionally, faults that result in a retry incur a one-second retry interval, and each test execution can have multiple retries. All of this results in an order increase in average runtime per test case. Nonetheless, it is possible to reduce the search space from trillions of cases to just under 500 test cases, which can run in around 30 minutes.

Data Nondeterminism
Responses in both scenarios had data nondeterminism in the form of generated UUIDs. To resolve this, we omitted the payload from the fault identification. This made the identification consistent, but led to an indistinguishable event in the case of scenario 1. As the runtimes for these scenarios are synchronous, we attempted to utilise the predecessors field in our identification to resolve this issue. But, in this case, all indistinguishable events are successive. In particular, two sequential interactions have the same identifier, and both have retries on specific failure modes. If they were asynchronous calls, this could have led to identification issues. In this synchronous case, Reynard is still able to infer the system's behaviour correctly without the inclusion of the predecessors field.

6.3.1. Uncovered State Divergence Bug
For scenario 1, we detected an unexpected behaviour using our automated test oracles. While Reynard only injects 5XX HTTP failures, the system responded in some cases with a 404 status code. After inspection, we uncovered a state divergence bug [38] in the interaction between three services. That is, there was a mismatch between the expected and the real state of a service.

Figure 6.22 illustrates the high-level interaction between these services. For service A to communicate with service C, it must first request some information from service B, which relies on service C. If an

[3] `https://kafka.apache.org/`

error occurs when service *B* interacts with service *C*, then service *B* performs an internal state update and propagates an erroneous response.

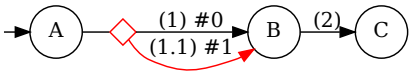


Figure 6.22: The high-level interaction that caused a resilience bug. Service *A* calls service *B*. If the call to *B* returns a status code 503 ("Service Unavailable"), it performs a retry. Service *B* calls service *C* when called.

The issue occurred because service *A* utilised an HTTP client that automatically performs a retry on a 503 status code, in an attempt to resolve a transient fault. Under normal circumstances, this would be safe for non-idempotent endpoints: a 503 status code indicates that the services never received the request. However, service *B* propagated the status code from service *C* in its response, which led to an unexpected retry from service *A*. The endpoint of service *B* is not idempotent, and will return a 404 response due to the state update it performed on the first attempt.

Component testing did not cover this unintended behaviour. A component test for service *B* covered its behaviour when a mocked service *C* was unavailable. The test suite for service *A* covered the behaviour of an unavailable service *B*. Yet, only under composition does this emergent behaviour occur.

Our two automatic oracles can help uncover this issue. We detect the propagation of a semantically incorrect status code by service *B*, when faults are injected at service *C*. Moreover, we observe a failure (the 404 response) without a direct cause on the retry. After the investigation, we could conclude that they are related.

6.3.2. Discussion

Our industry results show a discrepancy between the idealised benchmark systems and real systems. Due to increased functionality, the search space is more complex, and the overhead to prepare the system's state can be significantly higher. Nonetheless, even in complex cases, an exhaustive test can run in a tractable amount of time. Furthermore, we can identify bugs using our methods, even when our experiments were primarily designed to evaluate tractability and runtime efficiency.

Yet, it depends on the organisation if the runtime is reasonable. Especially in companies that deploy often, a half-hour window might be prohibitive. However, exhaustively testing key scenarios using automated fault injection testing can still be performed, but maybe not for every release.

It is worth noting that the test scenarios primarily involved synchronous and dependent interactions, which helps keep the search space tractable. As we have seen from the Filibuster *corpus*, specifically the *netflix* benchmark, if the system contains independent events, the search space grows more quickly. On the other hand, there are still techniques that can be applied to reduce the search space and increase runtime efficiency. For example, if we identify independent events, we can create a reduction policy that applies combinatorial testing [67] to significantly reduce the search space. Furthermore, if test scenarios can be run in parallel, we could speed up the runtime by running candidates in batches.

We do not expect Reynard to be applied to cover the complete fault space at every stage of the Software Development Life Cycle (SDLC). For example, we can run Reynard with only a subset of the services, some of which can be mocked. Moreover, we can configure Reynard with specific reduction policies (e.g., only applying faultloads up to a specific size) to ensure the tests run in a reasonable time for that specific stage, while catching lower complexity bugs. Only for releases would we expect the exhaustive test to be run.

6.4. Measuring the Overhead of Reynard

Table 6.4 lists the results of running stress tests on Reynard to measure its overhead. For relevant scenarios, we tested with a mocked controller or included the flag to mark it as the initial event, or both. Furthermore, we include the throughput in requests per second, and the mean latency and the 50th, 90th and 99th percentiles of response time. We ran the experiments for 1 minute. As such, the average latency is the average of all messages in that minute, which is equal to 60 times the requests per second.

We can see from the table that the latency introduced by the proxied server is minimal, with its request

Table 6.4: Measured overhead of Reynard in a stress test for different scenarios.

Scenario	Initial event?	Mocked controller?	Req/s	Average Latency (ms)	P50 (ms)	P90 (ms)	P99 (ms)
Direct to service			13 243	0.30	0.30	0.40	0.49
No instrumentation metadata			7 214	0.56	0.53	0.72	1.06
Unknown trace			6 968	0.59	0.55	0.76	1.25
Known trace, no fault			964	4.29	3.76	7.74	12.05
	✓		945	4.35	3.87	7.99	11.83
		✓	2 669	1.52	1.44	2.02	3.00
	✓	✓	2 657	1.53	1.45	2.04	3.14
Known trace, matching fault			903	4.90	3.90	10.04	17.37
Matching fault, large faultload			911	4.78	3.95	9.66	15.92

and response taking only 300 μs on average. The impact of the proxy for requests that are not part of a trace of interest is also minimal, with at most 1.06 ms of latency to the request, of which at most 490 μs can be attributed to the proxied server.

The overhead is higher for events that are part of a registered trace. Different parts of the request must be inspected, and a request to the controller is required to determine the unique identifier and to report the request. This results in an expected increase in overhead.

In the stress test, all requests are interpreted as events of the same trace. This leads to an increasing complexity over time in both the proxy and the controller. The overhead is more linear in the variant where the controller is replaced with a mock.

The influence of a larger faultloads is minimal, with an overhead in the same order as a small faultload. Most differences in results can be attributed to measurement noise.

6.4.1. Discussion

It can be concluded that the proxies are as transparent as possible to requests that are not part of the traces of interest. This implies that it is feasible to keep the instrumentation running in a test environment without influencing the SUT.

For traces of interest, the proxies introduce some expected overhead. Considering that every event passes through a proxy, the overall overhead increases linearly with the number of sequential fault injection points in a test scenario.

What is less optimal is that there are outliers that have significantly more overhead. There are several reasons why this might be the case. For one, the proxy and controller are written in Go, which has a garbage collector. Due to the stress test, a significant amount of memory is allocated to handle the requests. Especially in the case of an event of interest, the response body is retained in memory to be reported, rather than being streamed directly as a response. At one point, the garbage collector might be triggered, resulting in a significant increase in latency. Furthermore, for the controller to determine the invocation count atomically, the implementation uses a lock. Similarly, it keeps a lock on its internal datastore. If a specific request is kept in contention for the lock for an extended period, it causes delays. As we observe fewer significant outliers in the scenarios with a mocked controller, we deduce that there may be performance gains to be made in the controller.

The overhead in an actual test scenario can differ. In this benchmark, we run the controller, proxies and proxied server on the same machine. An additional network hop might have a more significant impact in a realistic system, where services can run on different machines. On the other hand, the measured latency represents the system's performance under stress. In non-stress situations, the overhead might be *less*.

There may be systems with strict timing requirements, for which the latency introduced by Reynard is prohibitive. However, for most Service-Oriented Distributed Systems, we assume that the overhead introduced by Reynard is acceptable as it is similar to the overhead introduced by a service mesh[4].

[4] `https://istio.io/latest/docs/ops/deployment/performance-and-scalability/#latency-for-istio-124`

6.5. Discussion Summary

We conducted multiple empirical evaluations of our tool for different evaluation purposes. Based on these evaluations, we can draw general conclusions related to our research questions.

We conclude that Reynard and its network-level instrumentation are practical to apply to existing Service-Oriented Distributed Systems (RQ1). This is supported by our empirical application to a wide variety of systems, comprising services written in various programming languages, utilising different communication protocols, and in different runtimes. Our application to an industry system additionally supports this. Furthermore, we find that our model can support a comprehensive set of interaction patterns, as highlighted by the distinct call graphs and system interactions covered in our empirical evaluation.

We can further conclude that Reynard is effective and efficient in assessing the resilience of a Service-Oriented Distributed Systems (RQ2). When compared to a state-of-the-art tool, we find that our reduction policies can reduce the search space to an equal number of concrete test executions, with a consistently lower runtime (RQ2.1). This provides confidence in the correctness of the dynamic analysis and model of the system's behaviour. When applying our retry reduction policy, we can further reduce the search space beyond the state of the art. However, we do find that there exist communication patterns which would be more challenging for Reynard to understand, but which are not part of the benchmark used for comparison. In general, we find that the search space, when reduced using our reduction policies, is dominated by the largest set of faults with the same direct causal dependency that can be feasibly combined (RQ2.2). Furthermore, we find that the instrumentation required by Reynard minimally influences the SUT, adding approximately 4 ms of delay per request of interest in high-load conditions (RQ2.3).

When applying Reynard to an industry system, we find that it can be effectively applied to a real system (RQ3). We find that Reynard can significantly reduce the search space in realistic and complex scenarios, making exhaustive testing tractable. Yet, it might not be as effective in every industry system, as the result depends on the system's interaction patterns, which influence the reduction of the search space. When compared to benchmark systems, we find that test scenarios have a significantly higher runtime. This is due to the increased complexity and functionality of system interactions, as well as the complexity of bringing the system into the correct state.

7

Related Work

The design of our tool is inspired by and builds upon previous work. In this chapter, we review the related work and its relevance to our own. We summarise their contributions and compare their difference to Reynard.

FATE and DESTINI (2011) [13] is a framework for exhaustively testing partial failures. It has two components: FATE is a fault injection and exploration service, DESTINI is a specification language to define correctness properties that can be validated using FATE. It uses unique identifiers for faults as an abstraction, and it automatically explores the full fault space. Furthermore, it addresses the exponential explosion of the fault space. To more efficiently visit the fault space, FATE primarily uses prioritisation using combinatorial testing [67]. FATE uses service-level fault injection, by intercepting I/O events.

PreFail (2011) [14] is a *programmable* fault injection framework. It is a continuation of the work of [13] and uses FATE as the underlying fault injection tool. PreFail enables users to program pruning policies, utilising expert knowledge to prune the fault space. This can be done as either a filter policy or a cluster policy. In a filter policy, a predicate function can be defined that filters test cases. In a cluster policy, a function can be defined that equates two faultloads. Using the cluster policy, only one is selected for each equivalence class. PreFail visits the search space in order of increasing size of faultloads. The different pruning policies are applied to all faultloads of the same size. The exploration strategy served as inspiration for Reynard.

Jepsen (2013) [53] is an automated random fault injection tool. Its primary focus is validating consistency models for distributed databases. Over the years, it has been used to perform over 46 analyses, finding numerous consistency bugs in various popular databases. Its failure model allows for a wide range of process-level and network-level failures by perturbing the containerised runtime. In particular, it allows for network partitions and clock modification that can help to uncover consistency bugs. Additionally, the authors have created a range of analysis tools to find and explain consistency bugs in the observed behaviour.

Lineage-Driven Fault Injection (2015) [15] and its corresponding tool Molly present an automated fault injection technique to drastically reduce the fault space. In particular, it encodes the system's lineage (behaviour) as logical statements, and uses a SAT Solver to find new test cases that could violate a property. It proves that, given the runtime conditions, it can give a parametrised guarantee that the property holds for all admissible faults. In the original paper, the runtime is significantly constrained. Programs must be written in a self-made logical language and free of concurrency and data nondeterminism. Furthermore, the failure model only allows for the complete failure of a service, not accounting for transient failures. In its original form, it is more functional as a model checker.

In a follow-up work, the authors collaborated with Netflix to apply Lineage-Driven Fault Injection (LDFI) in a real-world scenario [33]. They show that LDFI can be used to effectively explore the significantly large fault space found at Netflix. An interesting takeaway from the paper is that classifying behaviour by status code can be unsound, as they encountered errors with a 2xx status code in a real system.

Gremlin (2016) [46] is a network-level fault injection and assertion tool. Its reasoning convinced us that a practical fault injection tool utilises network-level instrumentation. Although the tool is systematic, it is not automated. It primarily supports systematic assertions on the correct functioning of resilience patterns, but does not systematically explore the fault space. The blast radius of injected faults in Gremlin is the service, not requests. As later works argue, a smaller blast radius is preferred. The tool Gremlin should not be confused with the Gremlin resilience platform-as-a-service[1] that was started by the author of [33].

IntelliFT (2020) [57] is an automated guided search fault injection tool. IntelliFT uses LDFI as a reduction algorithm and to find the relevant fault injection points. First, it injects faults at each found injection point in isolation. It then applies an evolutionary search algorithm to address the exponential explosion. Specifically, it starts with a batch of randomly selected faults and mutates them to find more impactful combinations of faults.

Filibuster (2021) [16] is a service-level automated fault injection testing tool that is closely related to our work. The primary difference lies in the method of fault injection. Filibuster extends the OpenTelemetry instrumentation to simulate service- and network-level failures. Furthermore, it has access to the services' runtime information, allowing it to infer causal dependencies and precisely identify fault injection points. Reynard relies on unaltered OpenTelemetry instrumentation to provide correlation, and applies network-level fault injection testing. Although this is less precise, it is more practical. Even as a proof-of-concept, our tool can operate in various runtime environments, whereas Filibuster requires rewriting its instrumentation implementation for each programming language and framework used.

3MileBeach (2021) [17] is a distributed tracing solution with service-level fault injection capabilities. It can trace and inject faults by replacing the serialisation and deserialisation libraries. Most programming languages have only one or a small number of serialisation libraries per protocol. Still, this requires adjusting all serialisation libraries in all relevant protocols and languages. Instead, Reynard relies on engineering effort already performed[2].

3MileBeach's main contribution is its ability to handle transient faults rather than only performing persistent failures. The paper mentions investigating how 3MileBeach could be used with LDFI, but to our knowledge, no such work has been performed at the time of writing.

RainMaker (2023) [38] is an automated fault injection tool that simulates partial failures of dependent cloud services. It proposes a taxonomy of error-handling bugs for cloud-backed services. Furthermore, it proposes several automated test oracles to verify the resilience properties of systems. Reynard takes inspiration from one of the automated test oracles, and we based our retry reduction policy on their fault injection policy.

MicroFI (2024) [41] is a network-level automated fault injection tool. It uses the Istio service mesh, and its fault injection capabilities[3]. Furthermore, it uses LDFI to reduce the search space. Its main contribution is a prioritisation algorithm based on Google's PageRank algorithm to prioritise impactful faultloads using runtime information. Reynard takes a similar approach to MicroFI in utilising OpenTelemetry's context propagation to help identify which message should be considered for fault injection. However, MicroFI cannot inject faults as precisely as Reynard, as they cannot precisely inject transient faults per service, as this is not tracked. Additionally, because it relies on Istio, it can only be used in environments that utilise Istio.

MicroFI shows that it can be feasible to parallelise and prioritise faultloads to improve the efficiency of an automated network-level fault injection tool. Furthermore, MicroFI takes a more holistic approach and uses the result of other test scenarios that have overlapping functionality to reduce their search space. Similar effort could be used to improve Reynard in future work.

[1] https://www.gremlin.com/
[2] https://opentelemetry.io/ecosystem/registry/
[3] https://istio.io/v1.20/docs/tasks/traffic-management/fault-injection/#injecting-an-http-abort-fault

8

Conclusion

At the start of this thesis, we formulated three research questions. In this chapter, we provide a concise answer to those questions.

(RQ1) How can we practically assess the resilience of a Service-Oriented Distributed System using automated fault injection testing?

Based on the insights of previous work, we conclude that network-level fault injection can make automated fault injection testing more practical. This is because network-level fault injection is agnostic of the runtime and only dependent on the used communication protocols. In Service-Oriented Distributed Systems, we expect this to be more uniform than the number of programming languages and frameworks used.

Combining network-level fault injection testing with existing distributed tracing instrumentation, such as OpenTelemetry, can make it more precise and reduce the blast radius. The instrumentation effort of including distributed tracing is non-zero, but it is feasible in most systems.

However, network-level instrumentation lacks access to service-level metadata. As such, it requires a more sophisticated representation of system interaction to infer the same information. We propose a search space exploration method, fault injection point identification and an abstract model to dynamically infer how faults influence the system behaviour, based on earlier observations. Using a search tree representation and applying different pruning policies, we can efficiently represent and reduce the search space. Based on our evaluation of benchmark systems and industry systems, we find that our method can accurately represent and understand the system's behaviour. Still, it can be expected that there exist test scenarios in real systems where service-level metadata is required to distinguish events correctly.

(RQ2) How effective and efficient is it to assess the resilience of a Service-Oriented Distributed System using automated network-level fault injection testing?

To validate the practicality of a network-level automated fault injection testing tool, we implemented a prototype of our design called Reynard. Based on our evaluation of Reynard, we conclude that using automated network-level fault injection testing for Service-Oriented Distributed Systems is effective and efficient. It is practical to apply Reynard to various Service-Oriented Distributed Systems, which would be impractical with the existing state of the art. Furthermore, we demonstrate that, compared to a state-of-the-art tool, Reynard can accurately and efficiently determine the minimal search space, even when lacking access to service-level metadata. Additionally, we show that Reynard enables the composition of a state-of-the-art reduction algorithm with a new reduction algorithm, further reducing the search space beyond the state of the art.

Our results indicate that the instrumentation required by Reynard is nearly transparent for existing services, thus reducing the risk of influencing the test results.

(RQ3) How effective is Reynard when applied to an industry system?

We show that applying Reynard to an industry system is practical. We can exhaustively cover the fault space for complex and key system interactions in a tractable amount of time. Using Reynard, we identified a previously undetected bug in the system. However, it depends on the release frequency and behavioural characteristics of a system to determine whether an exhaustive test is feasible for every release.

Furthermore, we find a discrepancy between idealised benchmark systems and real industry systems. Runtimes are higher as industry systems contain complex and unexpected behaviours. It would be beneficial to have more realistic benchmark systems to experiment with.

8.1. Future Work

Automated fault injection testing is a helpful tool for testing Service-Oriented Distributed Systems. While we address a subspace of the problem by aiding the technique's potential adoption, several open questions remain that future work could address.

We believe the following research directions are relevant to explore (in no particular order):

- **Add support for asynchronous communication** - Reynard was made to verify interactions that use a request-response communication pattern. However, many Service-Oriented Distributed Systems use distributed message queues and asynchronous communication. Although events can be traced through a message queue, the main challenge is determining whether the interaction is complete. Furthermore, most message queues have proprietary communication protocols. Injecting faults at the creation of a message in the queue requires extending the proxy to understand these protocols.
- **Define a general model of system interactions** - Reynard includes an abstract model of the effect of faults on the call graph. Currently, this model is evaluated by covering a large number of different systems and service interactions. However, some systems might exhibit an unseen type of complex logical behaviour, which the model might incorrectly interpret.

  To our knowledge, there does not exist an abstract model that covers all feasible behaviours of a service endpoint in Service-Oriented Distributed Systems. This makes it difficult to verify the completeness or soundness of our approach beyond empirical evaluation. In the context of distributed algorithms, models exist that cover processes, but they lack a standard model for the observable effects of failures. This is complicated by the fact that real system processes are implemented in general-purpose programming languages and can perform any arbitrary logical or computational step or condition.
- **Investigate other automated test oracles** - As the test suite is automated, it is beneficial to automatically detect incorrect behaviour instead of relying on expert developers. It might not be trivial to define correctness properties for Service-Oriented Distributed Systems, but there exist multiple anti-patterns that could be detected.
- **Extend the model to include other failure types** - Our current model and tool could be extended to cover other failure modes, such as timing and response omission failures. Adding these failure modes affects the operation of current pruning policies and necessitates new policies to reduce the search space based on the expected behaviour of such failures. A challenging aspect is the composition of the different failure types. To allow new failure types, the implementation of the proxy can be extended to cover additional failure modes.
- **Control both scheduling and partial failure nondeterminism** - As all messages go through a controlled proxy, extending the instrumentation to control the ordering of events is possible. If fault identification is accurate, a list of happens-before requirements for concurrent events could be used to instruct the instrumentation. The instrumentation could then ensure this ordering, similar to what current state-of-the-art tools in the tangential field of concurrency testing can perform.

8.2. Functional Improvements of Reynard

While it is a prototype, we believe that Reynard has the potential to be used in practice. Yet, functional improvements could be added to Reynard to extend its functionality, effectiveness and practicality. Due

to the project's scope and time constraints, we were unable to incorporate all of these ideas.

We believe that the following improvements are feasible and useful (in no particular order):

- **Parallelize test execution** - Due to the request-level blast radius, it would be feasible to parallelise test execution. This is not a novel idea; a similar concept is proposed by [14] and implemented in [41].

  A challenge with this approach is identifying the root cause in the event of a failure. When a failure exceeds its blast radius, it might be unclear which concurrent test case is responsible. [41] proposes re-running all candidates sequentially when a violation is detected.
- **Allow the prioritization of faultloads** - Multiple related works have proposed methods to prioritise faultloads that are more likely to cause property violations. This can be useful even if we intend to test exhaustively, as it can find a counterexample faster.

  In the current implementation of Reynard, this could be a separate component and interface for comparing two faultloads. A generator component could use these components to sort its queued faultloads.

  A potential challenge of sorting faultloads is that it can influence the efficiency of pruning policies, as they rely on historic observations. Prioritisation might be more efficient in finding bugs faster, but it might increase the search space when no bugs are present.
- **Extend the proxy to work with additional protocols** - Our current implementation can instrument messages that use the HTTP or gRPC protocols. These protocols are commonly used for communication between (micro) services. However, communication between services and databases or other (distributed) data structures often relies on proprietary protocols.

  To enable fault injection between services and their data stores, it would be beneficial to incrementally add different protocols so that Reynard can correctly determine their identifier and inject realistic faults.
- **Enable users to define equivalence classes of behaviours** - Reynard has a built-in assumption that system interactions are equivalent if they have the same status code. If a system violates this assumption, then Reynard is not guaranteed to be complete. This could be resolved if the user can define an equivalence relation for responses that is specific to their system.

  Furthermore, a similar approach could be used to automatically prune system test cases, similar to the pruning strategy presented in PreFail [14].
- **Provide tooling to deploy Reynard in different runtimes** - To decrease the cost of instrumentation, it would be useful to provide tools to enable automatic instrumentation. For our evaluation, we built a script to automatically adjust Docker Compose files for use with Reynard. This could be integrated into a more functional tool that performs similar functionality for Kubernetes or other distributed runtimes.
- **Provide tooling to visualize reports** - Currently, the users are mostly provided with a report in the form of several JSON files. We developed scripts to visualise parts of these reports for our evaluation. It would be beneficial to enhance these tools by providing detailed visual reports that enable users to understand what was explored and aid in resolving uncovered bugs. Furthermore, it could aid the user in understanding the resilience patterns in their system.

References

[1] M. Isaac and S. Frenkel, "Gone in Minutes, Out for Hours: Outage Shakes Facebook," *The New York Times*, Oct. 2021.

[2] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. DOI: 10.1145/359545.359563.

[3] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*, 2nd ed. Springer-Verlag Berlin Heidelberg, 2011.

[4] M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed. distributed-systems.net, 2023.

[5] M. Kleppmann, *Designing Data-Intensive Applications*, 1st ed. O'Reilly Media, 2017.

[6] R. Vitillo, *Understanding Distributed Systems*. Roberto Vitillo, 2021.

[7] J. Waldo, G. Wyant, A. Wollrath, and S. C. Kendall, "A Note on Distributed Computing," Sun Microsystems, Inc., Technical Report, Nov. 1994.

[8] D. Yuan, Y. Luo, X. Zhuang, *et al.*, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *11th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2014, pp. 249–265.

[9] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, *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*, Nov. 2014, 7:1–7:14. DOI: 10.1145/2670979.2670986.

[10] M. Waseem, P. Liang, M. Shahin, A. Ahmad, and A. R. Nasab, "On the nature of issues in five open source microservices systems: An empirical study," in *Evaluation and Assessment in Software Engineering*, Jun. 2021, pp. 201–210. DOI: 10.1145/3463274.3463337.

[11] C. Lou, P. Huang, and S. Smith, "Understanding, detecting and localizing partial failures in large system software," in *USENIX symposium on networked systems design and implementation*, Feb. 2020, pp. 559–574.

[12] J. Soldani, G. Montesano, and A. Brogi, "What went wrong? Explaining cascading failures in microservice-based applications," in *Service-Oriented Computing*. Springer International Publishing, 2021, pp. 133–153. DOI: 10.1007/978-3-030-87568-8_9.

[13] H. S. Gunawi, T. Do, P. Joshi, *et al.*, "FATE and DESTINI: A framework for cloud recovery testing," in *USENIX Symposium on Networked Systems Design and Implementation*, Mar. 2011.

[14] P. Joshi, H. S. Gunawi, and K. Sen, "PREFAIL: A programmable tool for multiple-failure injection," in *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2011, pp. 171–188. DOI: 10.1145/2048066.2048082.

[15] P. Alvaro, J. Rosen, and J. M. Hellerstein, "Lineage-driven fault injection," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 2015, pp. 331–346. DOI: 10.1145/2723372.2723711.

[16] C. S. Meiklejohn, A. Estrada, Y. Song, H. Miller, and R. Padhye, "Service-level fault injection testing," in *ACM Symposium on Cloud Computing*, vol. 40, Nov. 2021, pp. 388–402. DOI: 10.1145/3472883.3487005.

[17] J. Zhang, R. Ferydouni, A. Montana, D. Bittman, and P. Alvaro, "3MileBeach: A tracer with teeth," in *ACM Symposium on Cloud Computing*, Nov. 2021, pp. 458–472. DOI: 10.1145/3472883.3486986.

[18] D. E. Flipse, *Automated fault injection testing for service-oriented distributed systems*, Jan. 2025. [Online]. Available: https://github.com/reynard-testing/reynard.

[19] S. Ghosh, *Distributed Systems: An Algorithmic Approach, Second Edition*, 2nd ed. Chapman & Hall, 2014.

[20] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, "TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, Mar. 2016, pp. 517–530. DOI: 10.1145/2872362.2872374.

[21] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi, "SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems," in *USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2014, pp. 399–414.

[22] J. F. Lukman, H. Ke, C. A. Stuardo, *et al.*, "FlyMC: Highly scalable testing of complex interleavings in distributed systems," in *Proceedings of the EuroSys Conference*, ACM, Mar. 2019, 20:1–20:16. DOI: 10.1145/3302424.3303986.

[23] B. K. Ozkan, R. Majumdar, and S. Oraee, "Trace aware random testing for distributed systems," *Proceedings of the ACM on Programming Languages*, vol. 3, 180:1–180:29, Oct. 2019. DOI: 10.1145/3360606.

[24] D. Wang, W. Dou, Y. Gao, C. Wu, J. Wei, and T. Huang, "Model checking guided testing for distributed systems," in *Proceedings of the European Conference on Computer Systems*, ACM, May 2023, pp. 127–143. DOI: 10.1145/3552326.3587442.

[25] G. Wang, L. Zhang, and W. Xu, "What can we learn from four years of data center hardware failures?" In *IEEE/IFIP International Conference on Dependable Systems and Networks*, Jun. 2017, pp. 25–36. DOI: 10.1109/DSN.2017.26.

[26] A. Rotem-Gal-Oz, *Fallacies of Distributed Computing Explained*, Jul. 2006.

[27] S. Swoyer and M. Loukides, *Microservices adoption in 2020*, Jul. 2020.

[28] "Microservices Architecture: Have Engineering Organizations Found Success?" Gartner. (Jul. 2023), [Online]. Available: https://www.gartner.com/peer-community/oneminuteinsights/omi-microservices-architecture-have-engineering-organizations-found-success-u6b.

[29] A. Seth, A. R. Singla, and H. Aggarwal, "Service oriented architecture adoption trends: A critical survey," in *Contemporary Computing*, Springer Berlin Heidelberg, 2012, pp. 164–175.

[30] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, *et al.*, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*, Springer, 2017, pp. 195–216. DOI: 10.1007/978-3-319-67425-4_12.

[31] X. Zhou, X. Peng, T. Xie, *et al.*, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, Feb. 2021. DOI: 10.1109/TSE.2018.2887384.

[32] C. Meiklejohn, "Resilient Microservice Applications, by Design, and without the Chaos," Thesis, Carnegie Mellon University, Jun. 2024. DOI: 10.1184/R1/25901422.V1.

[33] P. Alvaro, K. Andrus, C. Sanden, C. Rosenthal, A. Basiri, and L. Hochstein, "Automating failure testing research at internet scale," in *Proceedings of the ACM Symposium on Cloud Computing*, Oct. 2016, pp. 17–28. DOI: 10.1145/2987550.2987555.

[34] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 277–288, Nov. 1984. DOI: 10.1145/357401.357402.

[35] J. Gray, "Why do computers stop and what can be done about it?" In *Symposium on Reliability in Distributed Software and Database Systems*, 1986, pp. 3–12.

[36] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions," in *USENIX conference on file and storage technologie*, Feb. 2017, pp. 149–166.

[37] V. J. M. Manès, H. Han, C. Han, *et al.*, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, Nov. 2021. DOI: 10.1109/TSE.2019.2946563.

[38] Y. Chen, X. Sun, S. Nath, Z. Yang, and T. Xu, "Push-button reliability testing for cloud-backed applications with Rainmaker," in *USENIX symposium on networked systems design and implementation*, Apr. 2023, pp. 1701–1716.

[39] Y. Gao, W. Dou, F. Qin, *et al.*, "An empirical study on crash recovery bugs in large-scale distributed systems," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Oct. 2018, pp. 539–550. DOI: 10.1145/3236024.3236030.

[40] M. J. Fischer, N. A. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985. DOI: 10.1145/3149.214121.

[41] H. Chen, P. Chen, G. Yu, X. Li, and Z. He, "MicroFI: Non-intrusive and prioritized request-level fault injection for microservice applications," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 5, pp. 4921–4938, Sep. 2024. DOI: 10.1109/TDSC.2024.3363902.

[42] T. Dai, J. He, X. Gu, and S. Lu, "Understanding real-world timeout problems in cloud server systems," in *IEEE International Conference on Cloud Engineering*, Apr. 2018, pp. 1–11. DOI: 10.1109/IC2E.2018.00022.

[43] Y. Chen, F. Ma, Y. Zhou, M. Gu, Q. Liao, and Y. Jiang, "Chronos: Finding timeout bugs in practical distributed systems by deep-priority fuzzing with transient delay," in *IEEE Symposium on Security and Privacy*, May 2024, pp. 1939–1955. DOI: 10.1109/SP54263.2024.00109.

[44] J. He, T. Dai, and X. Gu, "TScope: Automatic timeout bug identification for server systems," in *IEEE International Conference on Autonomic Computing*, Sep. 2018, pp. 1–10. DOI: 10.1109/ICAC.2018.00010.

[45] J. He, T. Dai, and X. Gu, "TFix: Automatic timeout bug fixing in production server systems," in *IEEE International Conference on Distributed Computing Systems*, Jul. 2019, pp. 612–623. DOI: 10.1109/ICDCS.2019.00067.

[46] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: Systematic resilience testing of microservices," in *IEEE International Conference on Distributed Computing Systems*, Jun. 2016, pp. 57–66. DOI: 10.1109/ICDCS.2016.11.

[47] P. Alvaro and S. Tymon, "Abstracting the geniuses away from failure testing," *Communications of the ACM*, vol. 61, no. 1, pp. 54–61, Dec. 2017. DOI: 10.1145/3152483.

[48] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015. DOI: 10.1109/TSE.2014.2372785.

[49] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Educational, 2002.

[50] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 872–923, May 1994. DOI: 10.1145/177492.177726.

[51] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How Amazon web services uses formal methods," *Communications of the ACM*, vol. 58, no. 4, pp. 66–73, Mar. 2015. DOI: 10.1145/2699417.

[52] J. Bornholt, R. Joshi, V. Astrauskas, *et al.*, "Using lightweight formal methods to validate a key-value storage node in Amazon S3," in *ACM Symposium on Operating Systems Principles*, Oct. 2021, pp. 836–850. DOI: 10.1145/3477132.3483540.

[53] K. Kingsbury, *Distributed Systems Safety Research*, Oct. 2024. [Online]. Available: https://jepsen.io.

[54] R. Majumdar and F. Niksic, "Why is random testing effective for partition tolerance bugs?" *Proceedings of the ACM on Programming Languages*, vol. 2, 46:1–46:24, Dec. 2018. DOI: 10.1145/3158134.

[55] C. Rosenthal and N. Jones, *Chaos Engineering: System Resiliency in Practice*. O'Reilly Media, 2020.

[56] H. Chen, W. Dou, D. Wang, and F. Qin, "CoFI: Consistency-guided fault injection for cloud systems," in *IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 536–547. DOI: 10.1145/3324884.3416548.

[57] Z. Long, G. Wu, X. Chen, C. Cui, W. Chen, and J. Wei, "Fitness-guided resilience testing of microservice-based applications," in *IEEE International Conference on Web Services*, 2020, pp. 151–158. DOI: 10.1109/ICWS49710.2020.00027.

[58] Y. Gao, W. Dou, D. Wang, *et al.*, "Coverage Guided Fault Injection for Cloud Systems," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, ser. ICSE '23, IEEE, Jul. 2023, pp. 2211–2223. DOI: 10.1109/ICSE48619.2023.00186.

[59] W. Feng, Q. Pei, Y. Gao, *et al.*, "Faultfuzz: A coverage guided fault injection tool for distributed systems," in *Proceedings of the IEEE/ACM International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 129–133. DOI: 10.1145/3639478.3640036.

[60] C. S. Meiklejohn, R. Padhye, and H. Miller, "Distributed Execution Indexing," Sep. 2022. DOI: 10.48550/ARXIV.2209.08740.

[61] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany, "An analysis of network-partitioning failures in cloud systems," in *USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2018, pp. 51–68.

[62] S. Dawson and F. Jahanian, "Deterministic fault injection of distributed systems," in *Theory and Practice in Distributed Systems*, vol. 938, Springer, 1994, pp. 178–196. DOI: 10.1007/3-540-60042-6_13.

[63] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services," in *USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2014, pp. 217–231.

[64] B. H. Sigelman, L. A. Barroso, M. Burrows, *et al.*, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Tech. Rep., 2010.

[65] D. Bittman, E. L. Miller, and P. Alvaro, *Co-evolving Tracing and Fault Injection with Box of Pain*, Mar. 2019. DOI: 10.48550/ARXIV.1903.12226.

[66] J. Yang, T. Chen, M. Wu, *et al.*, "MODIST: Transparent model checking of unmodified distributed systems," in *Proceedings of the USENIX symposium on Networked systems design and implementation*, Apr. 2009, pp. 213–228.

[67] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The combinatorial design approach to automatic test generation," *IEEE Software*, vol. 13, no. 5, pp. 83–88, Sep. 1996. DOI: 10.1109/52.536462.