Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Simulation-based Evaluation of Resilience Antipatterns in Microservice Architectures

Samuel Beck

Course of Study:	Softwaretechnik
Examiner:	Dr.-Ing. André van Hoorn
Supervisor:	Thomas Düllmann, M.Sc., Dr. rer. nat. Teerat Pitakrat, Dipl.-Ing. Markus Blumenthal
Commenced:	February 15, 2018
Completed:	August 15, 2018

Abstract

Monolithic applications are gradually getting replaced by systems built after the emerging microservice architectural pattern. Microservices are designed for failure since highly distributed systems come with a diverse range of potential failure points. Resilience patterns are the most effective measure to keep failures from spreading through a system once they occur. Some companies that build many of their systems as microservices employ approaches like Chaos Engineering in which faults are injected directly into the production system to uncover flaws in the services' design. But the decision making for the selection of fault injecting experiments remains an open challenge and these experiments take a lot of time and effort. This thesis is created in the context of the Orcas project that aims for efficient resilience benchmarking of microservice architectures by incorporating architectural knowledge and knowledge about the relationships between (anti) patterns and suitable fault injection into the decision making.

An efficient way to explore those fault injection experiments could be to simulate them before promising fault injections are conducted on the real system. That is the approach proposed by this thesis. To achieve this, an existing microservice resilience simulator is extended and refined to be able to accurately and realistically simulate microservice architectures, fault injections and the circuit breaker resilience pattern. Furthermore, an architectural model extraction approach is developed which uses tracing to gather architectural information and condense it into automatically created architectural models that serve as input for the simulator and the Orcas Decision Engine, which suggests the next fault injection experiment. Finally, the work of this thesis is connected with the Orcas Decision Engine to enable efficient and fast testing of fault injections.

In the evaluation, multiple experiments are conducted on two exemplary microservice systems to investigate the accuracy of the extracted architectural models and the extended simulator. The results of the evaluation reveal that the developed architectural model extraction approach is promising, but shows some limitations due to its employment of dynamic analysis through tracing. Furthermore, they show that the expectations to the simulator's accuracy are not satisfied acceptably. However, it shows realistic behavior in terms of fault injections and state transitions of the circuit breaker pattern during the evaluation what makes it possible to use the simulator to uncover hidden antipatterns in microservice systems and, therefore, to assess the potency of fault injections, what is its primary use-case in the Orcas context.

Kurzfassung

Monolithische Anwendungen werden nach und nach durch Systeme ersetzt, die nach dem hervortretenden Microservice Architekturmuster entwickelt wurden. Microservices werden bereits mit einem Fokus auf Fehlertoleranz entworfen, da verteilte Systeme mit einer vielfältigen Reihe an potentiellen Fehlerpunkten einhergehen. Resilience Entwurfsmuster sind die effektivste Methode, um Fehler an der Ausbreitung durch ein System zu hindern. Manche Unternehmen die viele Microservicesysteme betreiben wenden Ansätze wie Chaos Engineering an, in welchen Fehler direkt in das Produktionssystem injiziert werden, um sicherzustellen, dass diese Mechanismen wie angenommen funktionieren. Doch die Entscheidungsfindung für die Auswahl von Fehlerinjektionsexperimenten ist weiterhin eine offene Herausforderung und die diese Experimente nehmen viel Zeit und Aufwand in Anspruch. Diese Arbeit entsteht im Kontext des Orcas Projekts, welches das effiziente Resilience-Benchmarking von Microservicearchitekturen durch Einbeziehung von Architekturwissen und Wissen über die Beziehungen zwischen (Anti-) Mustern und geeigneten Fehlerinjektionen in die Entscheidungsfindung zum Ziel hat.

Eine effiziente Weise diese Fehlerinjektionsexperimente zu untersuchen könnte sein sie zu simulieren bevor vielversprechende Fehlerinjektionen auf dem echten System durchgeführt werden. Dies ist der von dieser Arbeit verfolgte Ansatz. Um dies zu erreichen wird ein existierender Microservice Resilience Simulator erweitert und verbessert, um realistische und genaue Simulationen von Microservicearchitekturen, Fehlerinjektionen und dem Circuit Breaker Resilience Entwurfsmuster zu ermöglichen. Des Weiteren wird ein Architekturmodellextraktionsansatz entwickelt, welcher Tracing verwendet, um Informationen über Architekturen zu sammeln und automatisch in einem Architekturmodell zusammenzufassen. Dieses dient als Eingabe für den Simulator und die Orcas Decision Engine, welche das nächste Fehlerinjektionsexperiment vorschlägt. Abschließend, wird die Arbeit dieser Thesis mit der Orcas Decision Engine verknüpft, um effizientes und schnelles Testen von Fehlerinjektionen zu ermöglichen.

In der Evaluation werden mehrere Experimente an zwei Microservicesystemen durchgeführt, um die Genauigkeit der extrahierten Architekturmodelle und des erweiterten Simulators zu untersuchen. Die Ergebnisse der Evaluation ergeben, dass der entwickelte Architekturmodellextraktionsansatz vielversprechend ist, aber Einschränkungen aufgrund seiner Verwendung von dynamischer Analyse durch Tracing aufzeigt. Des Weiteren ergeben sie, dass die Erwartungen an die Genauigkeit des Simulators nicht zufriedenstellend erfüllt werden. Jedoch weißt der Simulator in der Evaluation ein realistisches Verhalten bezüglich Fehlerinjektionen und der Zustandsübergänge des Circuit Breaker Entwurfmusters auf, was es ermöglicht den Simulator zu verwenden, um versteckte Antipatterns ins Microservicesystemen aufzudecken und die Wirksamkeit von Fehlerinjektionen zu beurteilen — sein primärer Anwendungsfall im Orcas Kontext.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1. Motivation and Context

In software engineering, a pattern documents a proven recurring solution to a common design problem [Gam95]. In a similar fashion, antipatterns document recurring solutions to common problems which have negative consequences, such as lowering the performance or reducing the resilience of a system [SW03]. Monolithic applications are gradually getting replaced by systems built after the emerging microservice architectural pattern, in which an application is developed as a composition of self-contained services [FL14; New15]. But highly distributed systems come with a diverse range of potential failure points; therefore, microservices are designed for failure and with resilience in mind [FL14].

Resilience patterns are the best measure to keep failures from spreading further through the system once they happen and to enable services to recover after a failure occurred [Nyg07]. Some companies that build many of their systems as microservices go through a lot of effort to ensure that these mechanisms work as supposed [RHB+17]. The evaluation of resilience patterns and antipatterns in microservice architectures is currently mostly done by injecting failures into the production system and monitoring its behavior afterwards. This approach, heavily employed by companies like Netflix, is called Chaos Engineering [RHB+17]. One of its disadvantages is that real experiments take a lot of time and effort [NCM16]. Especially the decision making for the selection of experiments remains a challenge; the state of the art is to simply inject failures randomly into the system [BBR+16].

This thesis is created in the context of the Orcas project [Orc18] that aims for efficient resilience benchmarking of microservice architectures to improve this situation. By incorporating architectural knowledge and knowledge about the relationships between (anti) patterns and suitable failure injections resilience vulnerabilities should be detected

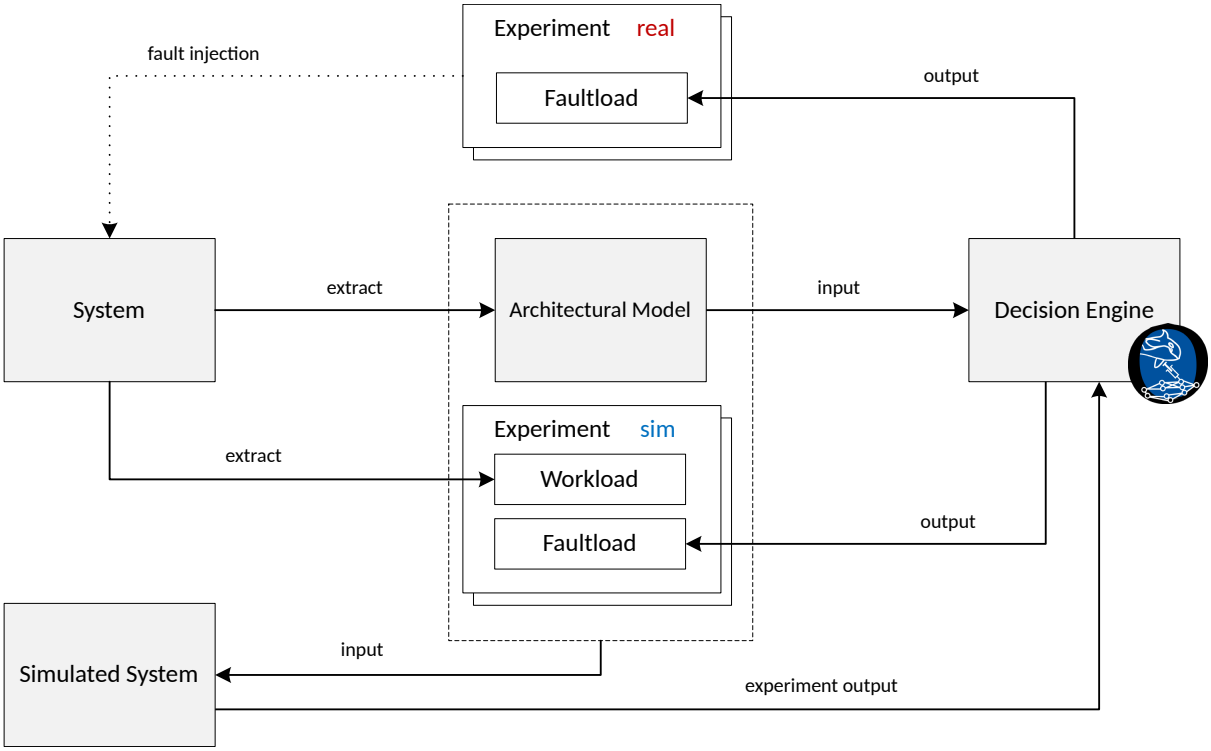


Figure 1.1.: Context and open challenges.

more efficiently [HDAP18]. An efficient way to run resilience experiments could be to simulate them on an architectural model instead of executing them on the real system, what is the open challenge addressed by this thesis. The faultload, which is needed for resilience testing, can be determined by the Orcas Decision Engine [HDAP18]. For this, an architectural model needs to be extracted from the respective system to serve as its input. Such an extraction approach is not yet integrated in the Orcas project. This faultload can not only be used for failure injection, but together with the architectural model also as input for a microservice resilience simulator [BGZ17], which simulates the resilience experiments and visualizes the results.

The context of this work and the open challenges, which will be addressed in order to meet the goals of this thesis, is illustrated in Figure 1.1.

The basis of the approach is an existing microservice application in production. The first step is the automated extraction of an architectural model from the application, which will serve as input for the Orcas Decision Engine and the microservice resilience simulator. The Decision Engine then uses the architectural model to create faultloads, which can be used in experiments on the production system or in the resilience simulator. To execute those simulations it is necessary to extract a model of the system's workload, too. The architectural model and the simulation experiment, which consists of a

faultload and workload model, serve as input for the simulator. After the simulation is finished, a report is created, which helps to investigate the application's resilience mechanisms. Furthermore, the output created by the simulator can also be used to improve future faultloads computed by the Orcas Decision Engine, which help to prioritize relevant resilience experiments that could be executed on the actual system for further evaluation.

Those faultloads can — via fault injection — just as well be used in experiments on the real microservice system, what will be used to evaluate the results of the simulator approach at the end of this thesis.

Two representative microservice applications will be used in the evaluation of the approach. First, an architectural model will be extracted from the system and then compared with an already existing or manually created model of the architecture to inspect the extracted model for completeness and correctness. Then experiments will be run on the real application to gather performance data that will be compared to the results of the simulation of the same experiments. The collected datasets will then be compared to evaluate how accurately the simulation corresponds to the real execution, if the behavior of the simulated circuit breaker matches the actual behavior of a real implementation of the circuit breaker pattern and how well the behavior of the simulated system corresponds to the real system's after failures have been injected into it.

1.2. Goals

This thesis addresses the simulation-based evaluation of resilience antipatterns in microservice architectures with focus on the extraction of architectural models and the simulation of resilience experiments in microservice systems. The goal is to extend the microservice resilience simulator developed by Beck et al. [BGZ17] — especially to improve the simulation of the circuit breaker resilience pattern — and to connect it with the Orcas Decision Engine, in order to simulate experiments that are created by the Decision Engine. The advantage which this approach promises in comparison with other resilience engineering and fault injection approaches for microservice systems is that the simulation-based evaluation has the capability to investigate resilience issues in a semi-automated and efficient way that does not negatively impact the user experience of the production system. Furthermore, the approach helps in context with the Orcas project to make the resilience benchmarking of microservice architectures more efficient by contributing to a better prioritization of the relevant experiments.

1.3. Research Questions and Methodology

The following research questions will be addressed in this thesis:

RQ1 How can the simulation of the circuit breaker pattern in the microservice resilience simulator be refined to reflect the behaviour of a popular implementation of the pattern?

RQ2 How can tracing be used to extract architectural models from microservice applications?

RQ3 Do the extracted architectural models accurately represent the real corresponding application?

RQ4 How accurately does the simulation of a microservice system through the microservice resilience simulator correspond to the execution of the real system?

RQ4.1 Does a simulated microservice system show the same behavior as the corresponding real microservice system after a fault has been injected into a service?

RQ4.2 Does the simulation of the circuit breaker resilience pattern through the microservice resilience simulator correspond to the actual behavior of the real implementation of a circuit breaker?

To solve RQ1, the functionality and parameters of the most popular software library that implements the circuit breaker pattern will be examined and compared with the current state of the circuit breaker in the simulator. Subsequently, the simulator will be adjusted to employ the same configuration parameters as the library and to reflect the same behaviour during simulation.

In order to solve RQ2, state-of-the-art approaches for architectural model extraction from microservice systems and distributed tracing tools will be studied and a new approach that takes advantage of tracing for extracting architectural information will be developed.

All other research questions will be explored in Chapter 6 based on various experiments that will be conducted on two selected microservice applications. To investigate RQ3, we will compare architectural models that have been extracted from the applications with the developed approach with existing models of the system's architectures. To solve RQ4, RQ4.1 and RQ4.2, the execution of the example systems will be compared with the output of the microservice resilience simulator under defined scenarios, e.g., failure injection. Additionally, supplementary material, containing software, dataset, and results, is publicly available online [Bec18].

1.4. Thesis Structure

This thesis is structured as follows:

Chapter 2 – State of the Art outlines the foundation of the thesis and related work. We briefly introduce the microservice architectural style, a collection of resilience patterns and antipatterns, the microservice resilience simulator and the OpenTracing standard.

Chapter 3 – Extension of Microservice Resilience Simulator describes the extensions made to the simulator. They compromise the division of the input model into an architectural model and an experiment model and the refinement of the circuit breaker pattern implementation after the model of Hystrix.

Chapter 4 – Architectural Model Extraction Approach presents the developed architectural model extraction approach that uses dynamic system analysis to extract architectural models automatically from collected OpenTracing traces.

Chapter 5 – Connection of Simulator and Orcas Decision Engine gives an overview over the decision-making algorithm of the Orcas Decision Engine and describes how the extended simulator and the developed architectural model extraction approach are connected with the decision engine.

Chapter 6 – Evaluation presents the evaluation of the proposed approach and its results. Furthermore, the obtained results are discussed and possible threats to validity are named.

Chapter 7 – Conclusion concludes the thesis by summarizing the experience and results that were gained throughout the thesis. Finally, possible future work is put in prospect.

Chapter 2

State of the Art

This chapter provides a technical foundation for the rest of the thesis and presents related work. Section 2.1 gives a description of the term *resilience* and explains how it is related to availability. Afterward, Section 2.4 will detail the *microservice architectural style*. Section 2.3 presents an explanation of the idea of *patterns* in software engineering and defines two patterns which will be in the focus of this thesis. Subsequently, Section 2.2 describes the notion of *antipatterns* and defines the three antipatterns which this thesis covers. Section 2.7 gives an overview of the *extraction of architectural models* from software systems and Section 2.5 presents the microservice resilience simulator which will be extended as part of this thesis. Section 2.8 gives an overview over the technologies used in this thesis and Section 2.9 presents related work at the end of the chapter.

2.1. Concepts of Dependable Computing

Avižienis et al. [ALRL04] define dependability as the ability to deliver service that can justifiably be trusted and, further, the dependability of a system as the ability to avoid service failures that are more frequent and more severe than is acceptable. They also define multiple attributes that make up the concept of dependability, such as availability and reliability, and threats to dependability. The following subsections present those definitions. Afterward, the term resilience is explained as well as how it is connected with the concepts defined by Avižienis et al.

2.1.1. Fault, Error und Failure

A *fault* is an underlying flaw in a system that has the potential to cause problems. It can be internal or external to the system. When it is activated during system execution,

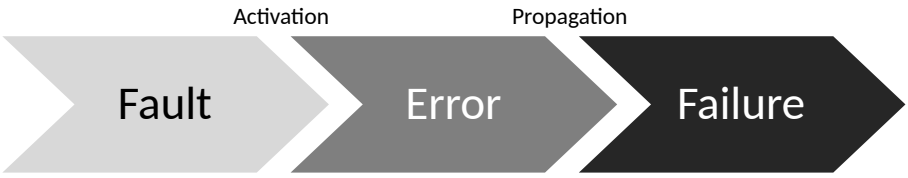


Figure 2.1.: Relationship between fault, error and failure, adopted from [HE17].

a fault leads to an *error* [HE17]. An *error* is the part of the total state of the system that may lead to a subsequent failure [ALRL04]. A *failure* occurs if an error reaches the service interface of a system. It results in system behavior that is inconsistent with the system's specification [HE17], i.e., the delivered service deviates from the system's correct service [ALRL04]. Figure 2.1 illustrates the relationship between fault, error and failure.

2.1.2. Availability, Reliability and Resilience

Avižienis et al. define *availability* as the readiness for correct service [ALRL04], i.e., it defines the proportion of time a system provides a correct service [HE17].

The availability of a software system is formally defined as:

$$availability = \frac{MTTF}{MTTF + MTTR} \tag{2.1}$$

Where MTTF stands for *Mean Time To Failure* and MTTR stands for *Mean Time To Recovery* [HE17].

Reliability is defined as the continuity of correct service [ALRL04], i.e., it is the property of a system that defines its probability to have an error or failure. It provides information about the error- or failure-free time period [HE17].

Resilience has been defined in literature in many different ways. In this thesis, resilience is defined after Haimes [Hai09], who defines it as *the ability of the system to withstand a major disruption within acceptable degradation parameters and to recover within an acceptable time and composite costs and risks*. That means, resilience is the ability of a system to quickly recover from a failure either without the user noticing or with graceful degradation of its services [Fri15]. Therefore, it is an attribute that increases the availability of a system.

Many approaches for increasing a system's availability usually have the goal to increase the MTTF of different system components and, thus, try to minimize the probability of a failure occurring. The following four categories [Lyu07] summarize common approaches to achieving reliable software systems:

1. Fault prevention: Avoid fault occurrences by design
2. Fault tolerance: Provide service complying with the specification despite faults having occurred
3. Fault removal: Detect the existence of faults and eliminate them
4. Fault forecasting: Estimate the presence of faults and their likely consequences

Fault tolerance is a synonym for resilience [ALRL04]. In today's complex and distributed systems it is impossible to completely prevent failures. They are not the exception, but rather the normal case [Fri16]. Every network call in a distributed system will eventually fail and, therefore, should already be seen as a potential point of failure during development [Nyg07].

Resilience does not increase availability by reducing the probability that a failure occurs; instead, it reduces the MTTR, i.e., reducing the mean time between the occurrence of a failure and the system's recovery from it [Fri16].

2.2. Antipatterns

An antipattern describes a recurring solution to a common problem, but on the contrary to design patterns (Section 2.3) this solution generates negative consequences for the system. It is a method to map a specific situation to a general class of problems. The definition of an antipattern usually describes the symptoms associated with the problem as well as its underlying root cause. In addition, antipatterns provide a common vocabulary for the discussion of problems and their solutions [BMMM98].

This thesis will confine itself to three antipatterns, which will be presented in the following subsections.

2.2.1. Cascading Failures

A cascading failure occurs when a failure in one component of the system causes problems in other parts of the system which call the failing component. Therefore the failure cascades through various components of the system, amplifying the problem. Preventing failures from cascading through the system is one of the key activities to increase the system's resilience. Cascading failures happen after something else has already gone wrong. Therefore, one approach to prevent them is to use the circuit breaker pattern at the system's integration points, where failures are bound to happen, to contain those failures in the component where they occur [Nyg07].

2.2.2. Integration Points

Every integration point like sockets, RPC, or REST will fail at one point of the system's life cycle. To counteract this, each integrated interface has to be treated like a potential failure point. Even worse, integration point failures are often difficult to debug at the application layer, because they occur mostly in the higher-level transport or communication protocols. Besides, failures at integration points usually evolve into cascading failures during the further execution of the application, if the system is not prepared to handle them. Therefore, countering failures at a system's integration points helps to prevent cascading failures. This can be achieved by defensive programming and, for instance, usage of the aforementioned circuit breaker pattern [Nyg07].

2.2.3. Slow Responses

Quick failures allow calling system components to process their call and quickly retry the transaction or fail as well. Slow responses, however, block resources on the calling component, which can lead to blocked threads and consequently to cascading failures. They usually evolve from excessing demand, i.e., when no more free request handlers are available, or happen as a system of underlying problems in the code. Slow responses can be the manifestations of memory leaks, network congestion or full TCP receive buffers. Hunting for these faults, using timeouts and failing fast helps to counter them [Nyg07].

2.3. Resilience Patterns

The term *pattern* originates from Christoph Alexander, an architect who was the first to divide architectural structures into different patterns in his book *A Pattern Language* from the year 1977 [Ale77]. In software engineering, a pattern documents a proven recurring solution to a common design problem. The documented solution is generally applicable and can be elaborated for specific circumstances to fit all needs and used technologies. It records expert knowledge to be reused and provides abstractions which simplify the discussion about software design and implementation [SW03].

Resilience patterns are design patterns that provide architecture and design guidance to reduce, eliminate, or mitigate the effects of faults in the system [Nyg07]. There is a wide range of them that can be applied to microservice architectures, far too wide for the scope of this thesis. Therefore, this work focuses on the circuit breaker and bulkhead pattern — two popular resilience patterns described by Nygard [Nyg07] — which will be

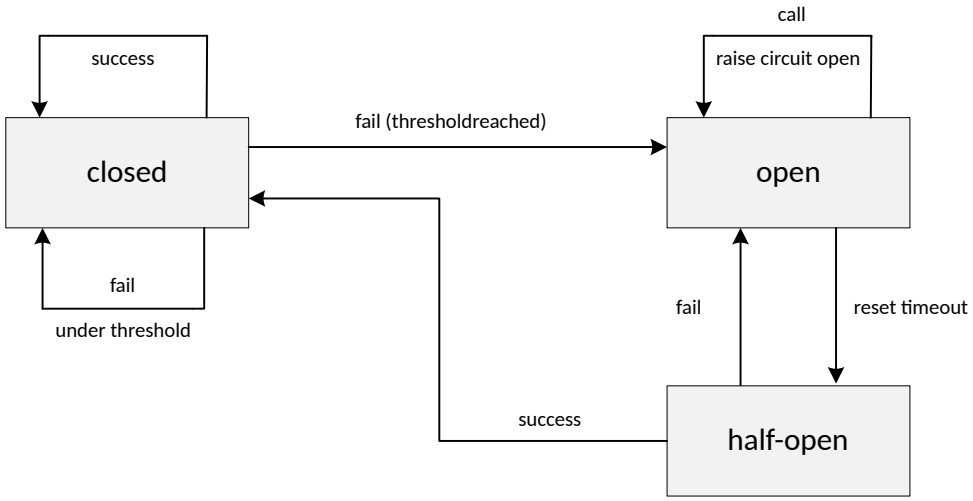


Figure 2.2.: Circuit breaker state diagram, adopted from [Fow14].

detailed hereafter. These design patterns were selected, because they are implemented by a popular latency and fault tolerance library [Hys18], which will be used later in the thesis.

2.3.1. Circuit Breaker Pattern

Many software systems, e.g., microservice applications, are distributed across multiple machines and need to make remote calls to communicate with different components. One of the big disadvantages of distributed systems is that every remote call can fail or hang without a response. If multiple components send calls to an unresponsive supplier this can lead to cascading failures across multiple components, especially in highly interconnected systems like microservices [Fow14].

The circuit breaker resilience pattern is used in distributed systems to prevent a failure from cascading from one component to others that depend on it and eventually from cascading through the entire system. To achieve this, remote function calls are wrapped in circuit breaker objects which monitor the requests for failures. Once a threshold is exceeded, the circuit breaker trips and requests are handled with a fall-back mechanism. This gives the requested service time to recover and eventually handle requests again. The circuit breaker checks the connection after a timeout and allows remote calls to pass through again when the receiver answers eventually [Nyg07]. An example for this can be found in Subsection 2.3.3.

The states and state transitions that a circuit breaker performs are illustrated in Figure 2.2.

2.3.2. Bulkhead Pattern

The bulkhead pattern is named after the bulkheads used in ships, which are metal partitions that divide the ship into multiple watertight compartments. This way, if one compartment is taking in water, the other compartments are protected and the ship does not irrevocable sink. Similarly, by partitioning a software system, a single failure is kept from bringing the entire system down. The most common application of the bulkhead pattern is physical redundancy in servers or entire data centers to prevent hardware failures from impacting the system's availability [Nyg07]. Another common example of the bulkhead pattern are separate thread pools. They serve the same purpose as redundant servers. If the threads of one thread pool are consumed through a failure, there are still threads available in the other thread pool to handle requests [Net18]. A thread pool contains multiple threads waiting for tasks to be concurrently executed. The advantage of maintaining a pool of threads is that latency, which is due to the constant creation and destruction of threads, is avoided and, therefore, performance increased [Goe02].

2.3.3. Hystrix

A popular library that implements the circuit breaker and bulkhead patterns is Hystrix[1]. Its development started in 2011 at Netflix and by now it is publicly available under an open source license. Since the company started using the library, uptime and resilience of its services reportedly experienced a dramatic improvement. The library is designed to give protection from latency and failures from dependencies between services and to stop cascading failures in distributed systems. This is achieved by failing fast, enabling a quick recovery and gracefully degrading if possible. At the same time, Hystrix also enables near real-time monitoring of microservice applications and displays the monitored data on a dashboard to improve operational control over the system [Net18].

Hystrix's Implementation of the Circuit Breaker Pattern

Assuming that the volume across a dependency, that implements a circuit breaker, meets a certain threshold and that the error percentage of that dependency exceeds the set error percentage threshold. In that case the circuit breaker transitions from the *closed* state to the *open* state, i.e., requests are no longer forwarded to the dependency; instead, they are directly answered through a fall-back method. In other words, all requests

[1] https://github.com/Netflix/Hystrix

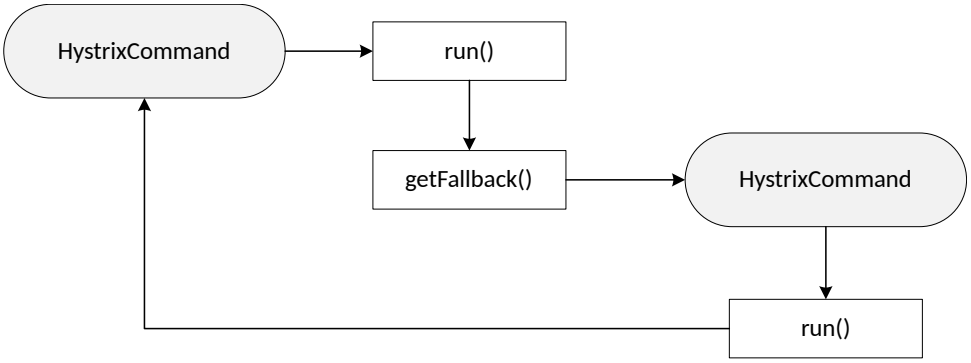


Figure 2.3.: Fallback: Cache via Network, adopted from [Net18].

made against the circuit breaker get short-circuited while it is open. After a defined amount of time, the next single request is let through and the circuit breaker transitions into the *half-open* state. If the request fails, the circuit breaker returns to the *open* state for the duration of the sleep window. Otherwise, if the request succeeds, the circuit breaker transitions back to the *closed* state and requests are sent to the dependency as usual [Net18].

To use Hystrix's implementation of the circuit breaker, network calls and other operations that can fail are wrapped inside a *HystrixCommand* object. The object has a *run* method, which executes the critical network call or operation. If an exception is thrown inside this method, the *HystrixCommand* object falls back to an alternative *fallback* method, which lets the operation degrade gracefully. The *fallback* method can also return a new *HystrixCommand* object, e.g., if the fallback operation can also fail. In that case, the new object must also provide a *fallback* method, as shown in Figure 2.3 [Net18].

2.4. Microservice Architectural Style

The microservice architectural style [New15] has been becoming popular throughout the past years and many organizations migrated their systems away from monolithic architectures towards microservices [BHJ16]; Amazon, Netflix, and The Guardian being among the pioneers [FL14]. This section will provide a brief explanation of the architectural style as a foundation for the remainder of the thesis.

In short, a microservice application is developed as a composition of decentralized, self-contained services that are independently deployed and communicate with each other through lightweight protocols, often HTTP. Furthermore, each service is developed around a business capability and can be written in a different programming language, which allows every development team to maintain their own technology stack. Each

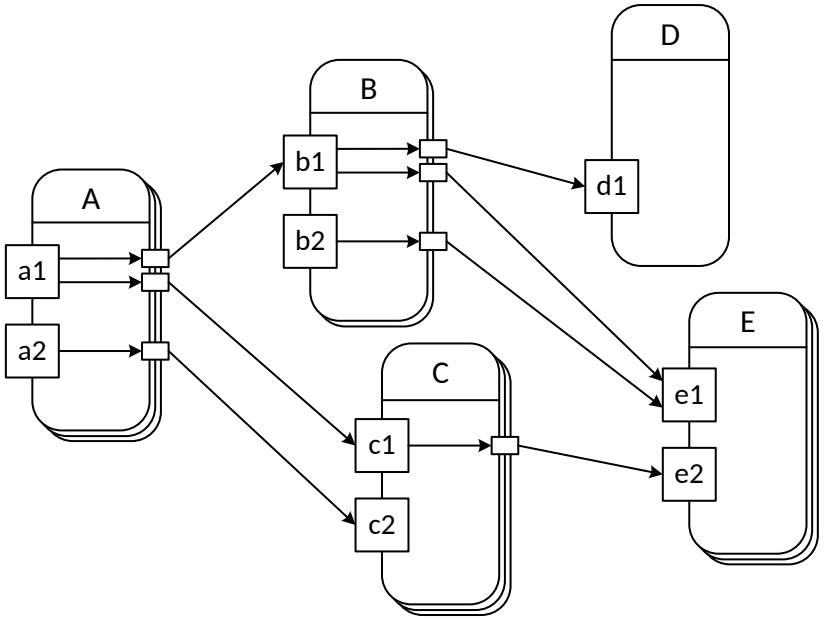


Figure 2.4.: An examplary microservice architecture.

service only has to adhere to the API requirements, otherwise the choice of technologies is flexible. Microservices are independently deployable through a fully automated continuous deployment pipeline [FL14]. Figure 2.4 shows an examplary microservice architecture. There is no formal definition of microservices; instead, in literature, they are often characterized by certain aspects and principles, e.g., by Fowler and Lewis [FL14], Rajesh RV [Raj16] and Newman [New15]. Various of these aspects have already been mentioned in the above explanation. We will now go into greater detail about the two that are the most important in the context of this thesis.

2.4.1. Autonomous Services

A system is often designed into several technological layers, e.g., presentation layer, business layer and database layer [Raj16]. Teams are created around these layers, too, leading to UI teams, server-side logic teams and database teams [FL14]. This is an example of Conway's Law, which says that "any organization that designs a system will inevitably produce a design whose structure is a copy of the organization's communication structure" [Con68]. Microservices on the other hand are developed around business capabilities. A single service has its own user interface, logic and database and is consequently developed by a cross-functional team, which includes all the skills necessary to develop such a service [Raj16]. This also means that, while monolithic systems usually employ one large database to store all the application data,

most services maintain their own database with the data that is relevant for themselves [FL14].

Furthermore, no application logic should take place within the system's communication mechanisms, but exclusively in the microservices, to keep them independent and loosely coupled with the rest of the system. Therefore, HTTP requests and REST are most commonly used to implement lightweight communication between the various services. This use of standard protocols, as they are also used to build the world wide web, enables the services to be interoperable with each other [Raj16].

2.4.2. Design for Failure

Microservice systems are highly distributed systems. This brings a whole range of potential failures with it, that developers have to prepare for. Any service call could fail at anytime due to unavailability of the called service or high network latency. The services need to be designed in a way that they can respond to the failure of services as gracefully as possible. The handling of such service failures adds additional complexity compared to traditional monolithic systems and is another disadvantage of the microservice architectural style. As a consequence, development teams constantly reflect on the repercussions of service failures on the user experience of the application [FL14]. An example for this is Netflix, who developed their Simian Army, a tool that randomly disables microservice instances in production, to make sure the system is prepared to survive the failure without customer impact. This goes along with monitoring of each service (Section 2.8.2) to detect failures as quickly as possible and to possibly automatically repair the faulty service [Net11].

Therefore, microservices are designed to be particularly resilient. One way to increase the resilience of a service is to implement the resilience patterns introduced in Section 2.3, which help to contain failures in the service they're occurring.

2.5. Microservice Resilience Simulator

We developed a microservice resilience simulator in our paper *Simulation-based Resilience Prediction of Microservice Architectures* [BGZ17], that is capable of simulating microservice architectures and failures of one or multiple service instances of the architecture and the repercussions of such a fault for the rest of the system. Furthermore, the simulator can also simulate the integration of certain resilience patterns like the circuit breaker pattern and, therefore, allows an investigation of the effectiveness of a systems resilience mechanisms.

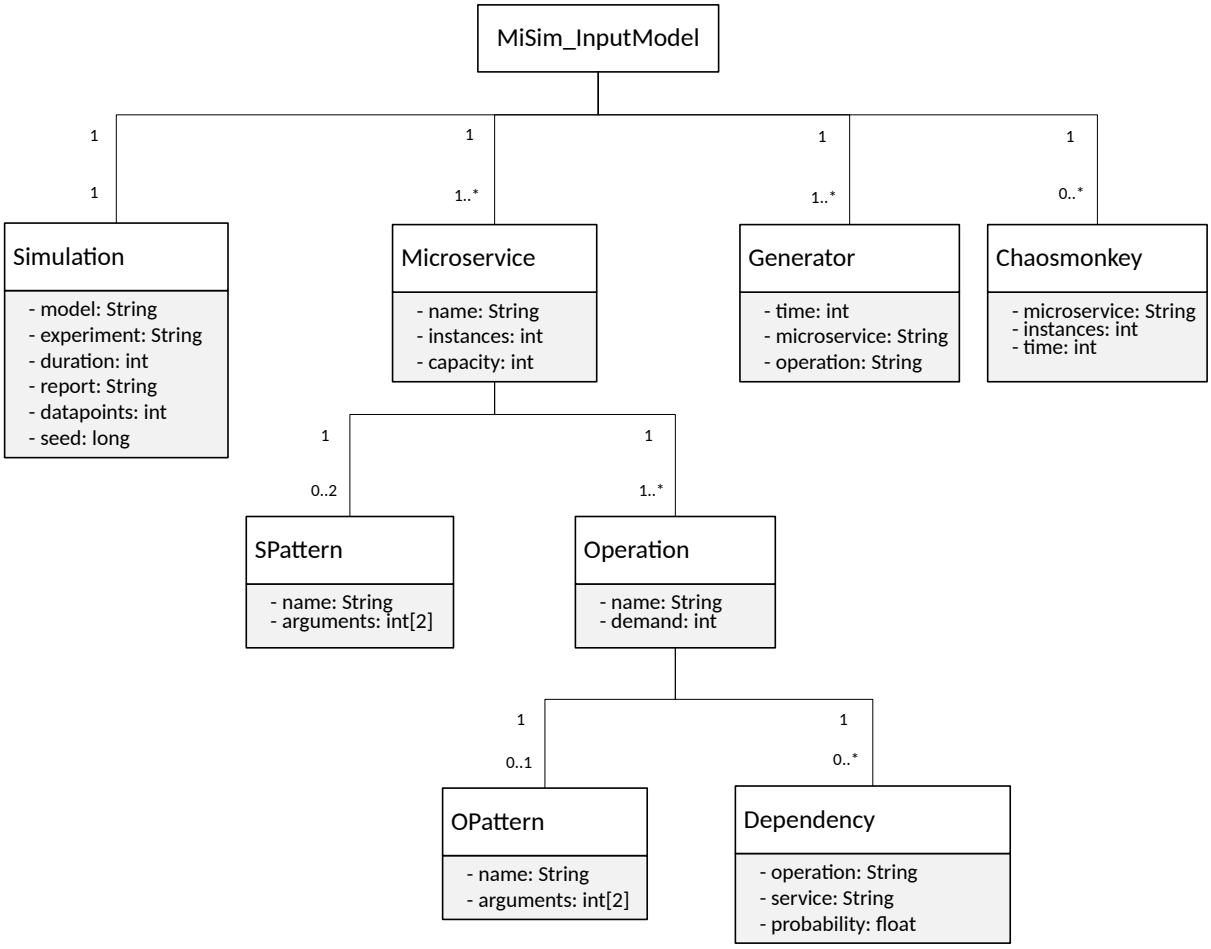


Figure 2.5.: UML class diagram illustrating the input model of the microservice resilience simulator [BGZ17].

The simulator was built using the Java-based Desmo-J framework, which allows the development of simulation models and, furthermore, supports discrete-event simulation as well as process-oriented simulation. It was created by the University of Hamburg in 1999 and has since been constantly maintained with updates [Uni17].

The simulator will be extended as part of this work to also support latency failure modes, besides, the simulation of the circuit breaker pattern will be improved and further parameters will be added to depict the resilience pattern more realistically.

As part of the simulator we developed an input model, which allows for an abstract description of microservice architectures, including services, service operations, service dependencies and applied resilience patterns. Further, the model contains meta information about the experiment as input for the simulator. It is also possible to define chaos monkey objects, that describe, which service instances should fail and when in the

simulation that is supposed to happen. Figure 2.5 gives an outline of the input model in the shape of an UML class diagram.

Instances of the input model are declared in the JSON format, an exemplary configuration can be found in Listing 2.1. Lines 2 to 9 contain meta data about the simulation. After that, the microservices of this example architecture are listed in lines 10 to 26. Following that is the specification of the request generators for this experiment in lines 27 to 32 and the specification of a chaos monkey in lines 33 to 38, which will shut down a service instance of the frontend service during the simulation.

The implementation of the circuit breaker pattern in the simulator will be refined in Chapter 3 and further parameters for this pattern will be added to the input model.

```
{
 "simulation": {
  "experiment": "experiment_name",
  "model": "simulation_model_name",
  "duration": 200,
  "report": "",
  "datapoints": 200,
  "seed": 1234
 },
 "microservices": [
  {
   "name": "frontend",
   "instances": 2,
   "capacity": 1000,
   "spatterns": [{"name": "Thread Pool", "arguments": [10, 5]}],
   "operations": [
    {
     "name": "login",
     "demand": 100,
     "opatterns": [{"name": "Circuit Breaker", "arguments": [2, 3]}],
     "dependencies": [
      {
       "operation": "save",
       "service": "logic",
       "probability": 0.1
      }]}]}],
 "generators": [
  {
   "time": 0.1,
   "microservice": "frontend",
   "operation": "login"
  }],
 "chaosmonkey": [
  {
   "microservice": "frontend",
   "instances": 1,
   "time": 50
  }]}
```

Listing 2.1: An instance of the input model of the simulator.

2.6. Fault Injection

Software fault injection is a method to anticipate worst-case scenarios caused by faulty software through the deliberate injection of software faults. It can be used for assessing and improving resilience mechanisms in software systems. The approach is to inject faults or software-, hardware-, or environmental-related errors into the system. Three aspects need to be considered when injecting a fault: *what* to inject, *where* to inject and *when* to inject. There are three important properties for fault injection: *representativeness*, *usability* (the ability to use fault injection in a new target system), and *efficiency* (ability to achieve useful results with a reasonable experimental effort). A fault injection system usually compromises a *load generator*, which sends inputs to the target system that will processed during a fault injection, and an *injector*, which introduces a fault in the system. The inputs and faults submitted to the system are referred to as workload and faultload. A fault injection campaign is formed by the execution of several experiments or fault runs. Typically, only one fault is injected per experiment. Another component of a fault injection system is the *monitor* entity. It measures data from the target system that allows the tester to assess the outcome of the experiment; whether the injected fault has been tolerated or the system has failed. In order to make that assessment, measured data is usually compared with data obtained from fault-free experiments. All three named components of a fault injection system are orchestrated by a *controller*, which is also responsible for storing the results of an experiment for later analysis [NCM16].

2.6.1. Chaos Engineering

Netflix engineers have developed an approach called *Chaos Engineering* [BBR+16] to build confidence in the capability of a distributed software system to withstand disruptive events that affect production environments. They define four principles of Chaos Engineering: (i) build a hypothesis around steady state behavior, i.e., hypothesize that a defined steady state will continue during resilience experiments, (ii) vary real-world events, i.e., any event capable of disrupting the steady state is a potential variable in a chaos experiment, (iii) run experiments in production, (iv) automate experiments to run continuously. Experiments are run in production to provide a realistic environment and workload to the experiment. The harder it is to disrupt the steady state, the more confidence is built in the behavior of the system. Chaos Engineering is applied by organizations such as Netflix, Amazon, Google, Microsoft and Facebook [BBR+16].

2.6.2. Fault Injection in Microservice Systems

Netflix developed the Netflix Simian Army [Net16] to induce various kinds of failures into their Amazon Web Services cloud infrastructures. Some of the simians introduced by Netflix the Chaos Monkey, which randomly shuts down production instances, Latency Monkey, which induces artificial delays in the system's communication layer, and Chaos Gorilla, which simulates an outage of an entire Amazon availability zone [Net11]. Similar tools are available for Microsoft Azure [Nak15].

In those approaches, the faults are injected randomly; the decision making of what experiments to run is stated as an open challenge [BBR+16].

2.7. Architectural Model Extraction

The microservice resilience simulator described in Section 2.5 requires a model as input that contains architectural information. The same applies to the Orcas Decision Engine [HDAP18]. At the moment, those models are created manually by hand. But the creation of architectural models is a tedious and error-prone task and models of complex systems can quickly grow vast and unclear [BHK11]. As described in Section 2.4, services of a microservice application are deployed continuously by independent teams. That means that the overall architecture of such a system undergoes frequent changes; an architectural model that represents the system must be adapted regularly. Gathering the necessary information is a challenging task, due to the independent nature of development teams that create and maintain services. The answer to these issues is to automatically extract architectural models from the system. The extraction process can be divided into three disciplines. The system's structure and behavior need to get extracted, which includes the system's components, the available resource landscape and all inter-component interactions. Furthermore, a resource demand estimation has to take place, where resource demand is the amount of hardware resources that are needed to process a unit of work like user requests or system operations. And finally, a workload characterization has to be conducted, for the extracted model to give an accurate representation of the system's usage profile. An extraction approach needs to solve all those three problems in order to correctly represent the real system [BHW+15].

2.8. Technologies

This section provides an overview over the technologies that are used in this thesis.

2.8.1. Docker

Docker[2] is a software platform that performs operating-system-level virtualization. It packages software into standardized units called *containers* that contain software libraries, system tools, code and runtime. Containers are isolated but can communicate with each other trough defined channels. They are more lightweight than virtual machines, because they virtualize the operating system of a host instead of virtualizing physical hardware. Multiple containers share the OS kernel of the host machine, each running as isolated processes [Doc18].

Docker Compose is a tool for defining and running multi-container Docker applications. A single file can be used to configure all the services of an application. They are created and started with a single command. Docker Compose allows for multiple isolated environments on a single host and preserves volume data when containers are created [Com18].

Both microservice applications used in the evaluation of the thesis (Chapter 6) run in Docker environments.

2.8.2. OpenTracing

There are many benefits to the switch from old monolithic systems towards microservice architectures. Services can be changed and deployed independently without impacting the rest of the application what gives their developers more individual freedom, a single service is organized around one business capability, which makes it easier to understand it since it represents only a small piece of functionality, microservices can be efficiently scaled, etc [FL14]. But once a system is split into multiple services and, therefore, makes the transformation to a distributed system, some previously simple tasks suddenly grow more complex, like latency optimization, debugging of backend failures or analysis of the communication between different system components [OT18].

Distributed tracing systems like Zipkin or Dapper try to address these issues [SBB+10]. Tracing is similar to logging and collects information about a program's execution. This information can be used for debugging purposes and to track the path that transactions take as they propagate through a distributed system [Dav95; Sig16]. But the downside of these tracing implementations is that their APIs are incompatible with each other [OT18]. Since another characteristic of microservices is that each developer team can choose their preferred technology stack for their service, especially also the programming

[2] https://www.docker.com/

language [New15], developers should not be limited in their choice by the particular distributed tracing implementation they are using [OT18]. Also, it is unreasonable to assume that all software packages included in a project use the same tracing vendor. In this case the traces would be incomplete and may miss function calls that are crucial for understanding or debugging a transaction [Sig16].

This is where OpenTracing[3] comes in. It is an open standard that offers consistent, vendor-netral APIs for many popular platforms and makes it easy to switch between tracing implementations with only marginal configuration changes [OT18]. With OpenTracing, development teams of services or software packages can instrument their code with the tracing implementation of their choice, as long as it supports the OpenTracing standard, and the implementations from different tracing vendors will be able to understand each other [Sig16].

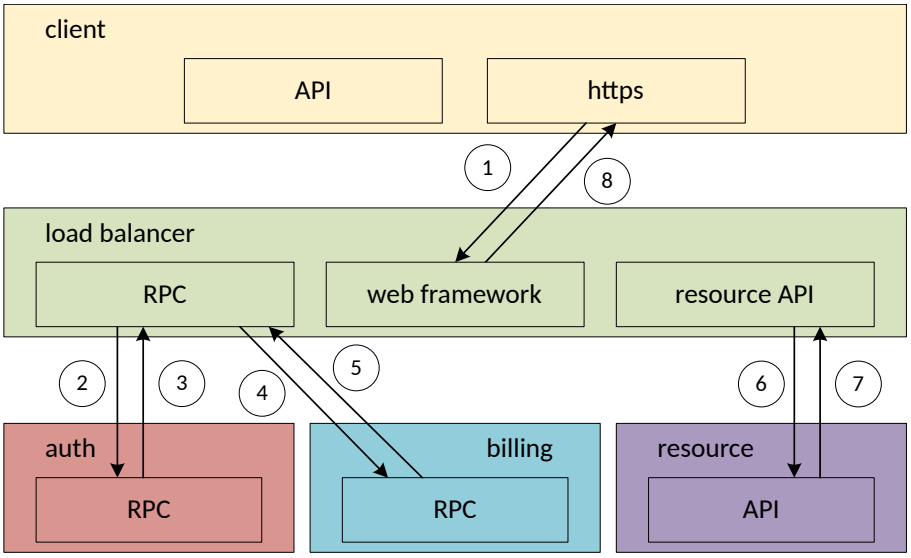
To understand how tracing works and which standards OpenTracing provides we will now look into what a trace is in OpenTracing.

Trace

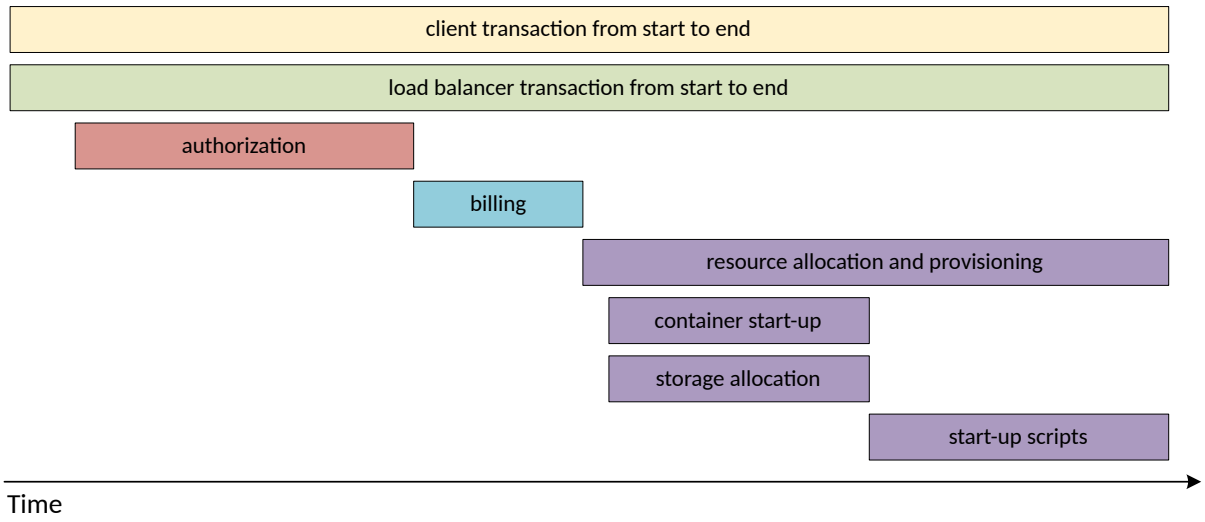
A trace [BL94] represents a transaction as it propagates through a system, which might be distributed. In OpenTracing, a trace is a directed acyclic graph of spans, which are named and timed operations that represent a coherent segment of work in that trace [OT18]. A span encapsulates the following state: an operation name, a start timestamp, a finish timestamp, a set of span tags and span logs that contain information about the span, a span context and references. A span may reference zero or more other spans that are causally related to it [OTS1818]. In a distributed trace, each service contributes its own span or even multiple spans. For instance, both the client and the server represent their respective role in the workflow of a remote call as at least one span. A span can also start child spans, either in serial or parallel [OT18].

Figure 2.6 visualizes what the basic trace of a transaction in a distributed application looks like. The first illustration shows the directed acyclic graph of spans, which makes it comprehensible how various services in the system are dependent on each other. A drawback to this kind of visualization is that time durations do not become apparent and that it does not scale well once more remote calls are undertaken in a transaction. The second illustration depicts the trace in a timeline; this kind of visualization is more useful in most cases. It adds the context of time, the hierarchy of the called services and it shows if operations are called in a serial or parallel manner [OT18].

[3] http://opentracing.io/



(a) Directed acyclic graph.



(b) Timeline.

Figure 2.6.: Basic trace visualized in different ways.

Scope of Standardization

OpenTracing allows developers to instrument their services and packages with the tracing implementation of their choice by standardizing three main activities during tracing. First, it offers a standard for span management. It provides programmatic APIs to start, finish and decorate spans. Second, it standardizes inter-process propagation by defining programmatic APIs to aid in the transfer of tracing context across process

boundaries. And third, it offers standardized active span management. That means APIs to store and retrieve the currently active span across package boundaries in a single process. Therefore, as illustrated in Figure 2.7, OpenTracing provides the standards that are needed for distributed tracing in microservice applications, including software packages and other third-party code, while giving developers the freedom of choosing their preferred tracing implementation or even to switch to another implementation with only a minimum of configuration changes needed [Sig16].

2.8.3. Jaeger

Jaeger[4] is one of the distributed tracing systems that implements the OpenTracing standard. It was originally developed by Uber Technolgies before it was released as open source and is now — just as OpenTracing — hosted by the Cloud Native Computing Foundation. Jaeger can be used for distributed context propagation, distributed transaction monitoring, root cause analysis, service dependency analysis and performance and latency optimization [Ja18].

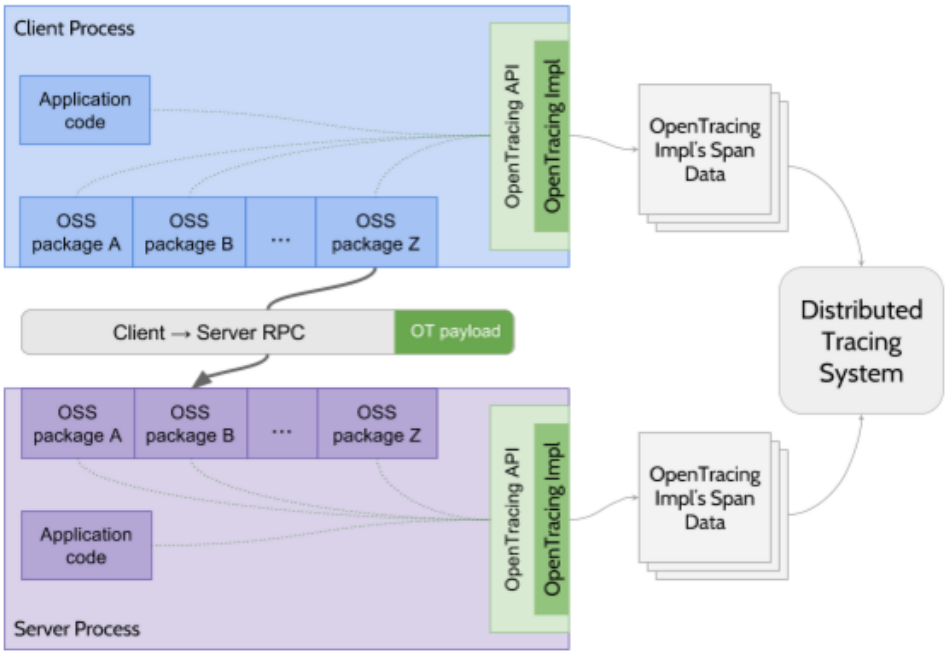


Figure 2.7.: OpenTracing offers standardized APIs for tracing [Sig16].

[4]https://www.jaegertracing.io

Jaeger's backend is implemented in Go, while the tracing system also features a UI created with React.js. In terms of storage it supports in-memory, Cassandra and Elasticsearch stores. Jaeger's architecture is built with scalability and parallelism in mind [Šab18b].

Figure 2.8 portrays the architecture of Jaeger. The client, which is integrated in the instrumented application, sends traces to the agent, which listens for inbound spans and informs the client about the used sampling strategy. The agent routes the obtained spans to the collector, which validates, transforms and stores the spans. The query service offers a REST API through which the tracing data in the storage can be accessed. The Jaeger UI uses this query service to present the tracing data [Shk17].

Users can use the UI to search for traces and filter the results after specific services or operations. As shown in Figure2.9, it supports different visualizations for traces, like a time sequence, a directed acyclic graph and a critical graph diagram [Shk17].

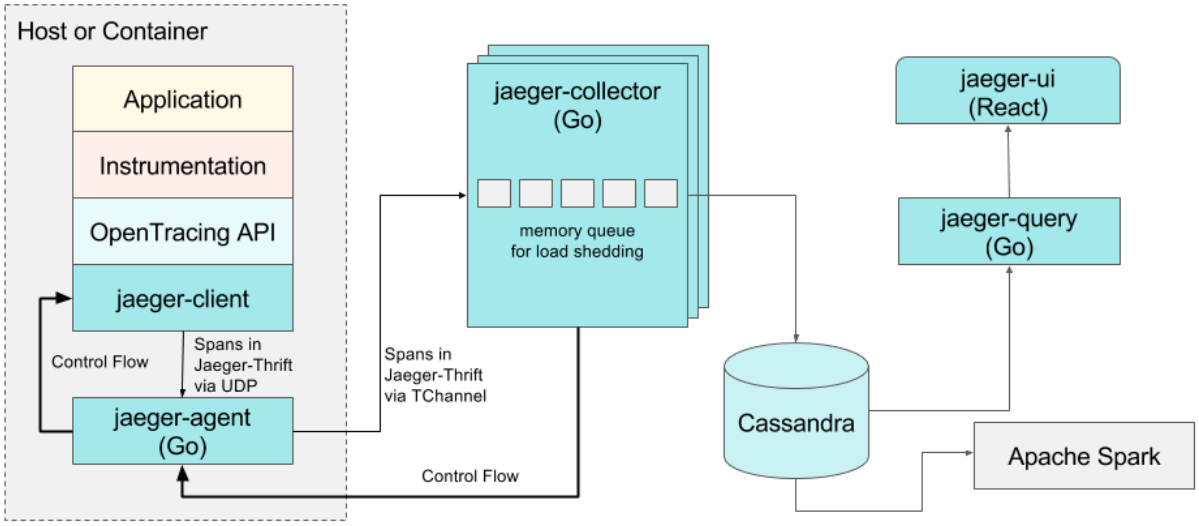


Figure 2.8.: The architecture of Jaeger [Shk17].

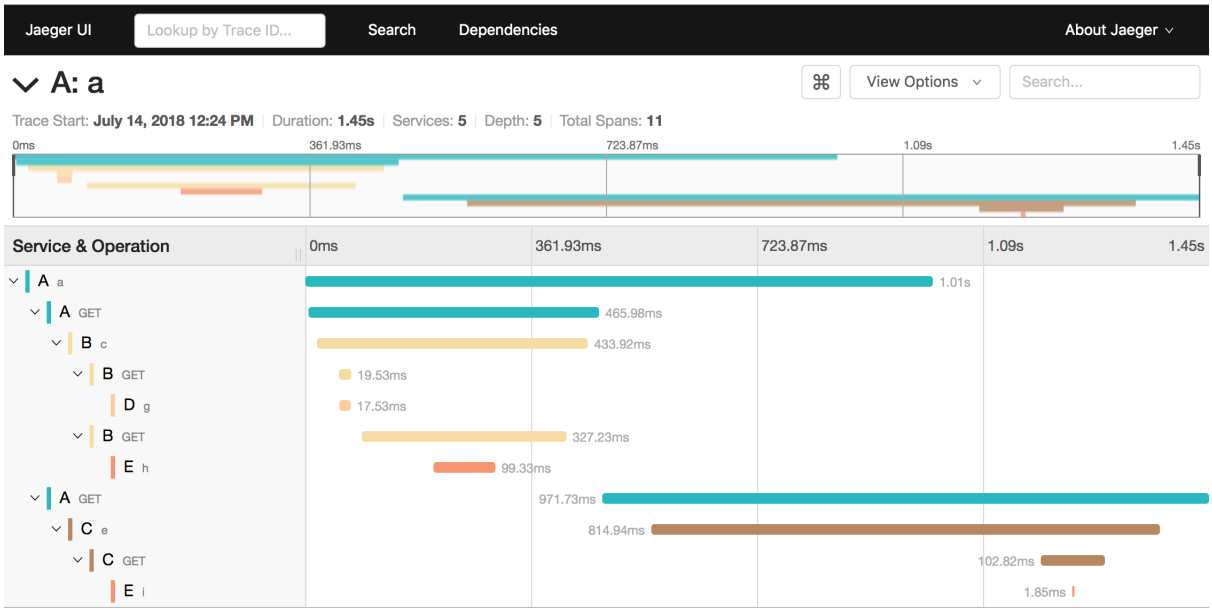


Figure 2.9.: A single trace in the Jaeger UI.

2.9. Related Work

This section presents approaches that are related to the topic of the thesis. Subsection 2.9.1 focuses on an approach for extracting architecture models from microservice systems, followed by an approach for architectural analysis of microservices via patterns in Subsection 2.9.2. Subsection 2.9.3 gives a short overview over two additional simulation tools in addition to the discussed microservice resilience simulator.

2.9.1. Architecture Extraction from Microservice Systems

Granchelli et al. [GCD+17] present an approach for recovering the architecture of microservice-based systems and a prototypical implementation of said approach named MicroART. It is able to automatically extract the architecture of a microservice-based system from a GitHub repository that contains its Docker-based source code and a reference to the Docker container engine that manages it. Their approach consists of two phases that are illustrated in Figure 2.10. Namely they are the architecture recovery phase, that creates an architecture model, and the architecture refinement phase, which outputs a refined architecture model.

The architecture recovery phase is based on the extract-abstract-present paradigm. In the extraction phase, information is extracted from artifacts like source code, documentation or the architect's knowledge. The abstraction phase is about grouping and filtering

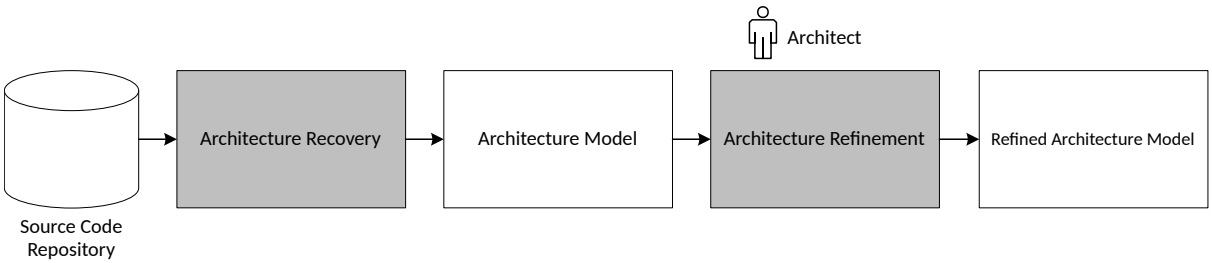


Figure 2.10.: Architecture extraction approach by Granchelli et al., adopted from [GCD+17].

the obtained information to get a focused set of information. At last, presentation is about organizing the information in a clear way. The resulting, automatically generated, architecture model then gets refined in the second phase, namely the architecture refinement phase. This phase is semi automatic, because it requires intervention by a software architect. Its purpose is to produce another architecture model which the architect considers more significant for its purposes. This refined architecture model has several advantages, e.g., for obtaining different views of the system's architecture customized on specific components. Granchelli et al. also developed a meta-model for describing extracted microservice architectures, which does a good job at displaying dependencies between services Granchelli et al. [GCD+17].

2.9.2. Architectural Analysis of Microservices via Patterns

Rombach [Rom17] proposes a microservice architectural template catalog. He documents different microservice-specific architectural patterns and their characteristics, as well as their influences on software systems, and maps these on formalized architectural templates. The catalog contains the patterns API gateway, client-side load balancing, service discovery and circuit breaker. As another contribution, he extends parts of SimuLizar, a simulator for self-adaptive systems, see Section 2.9.3, to allow for customized behavior specification interpretation. He conducts a case study to evaluate how the developed architectural templates help software architects. The results showed that analysis with architectural template instances provide accurate prediction results, but there is still potential for further enhancement [Rom17].

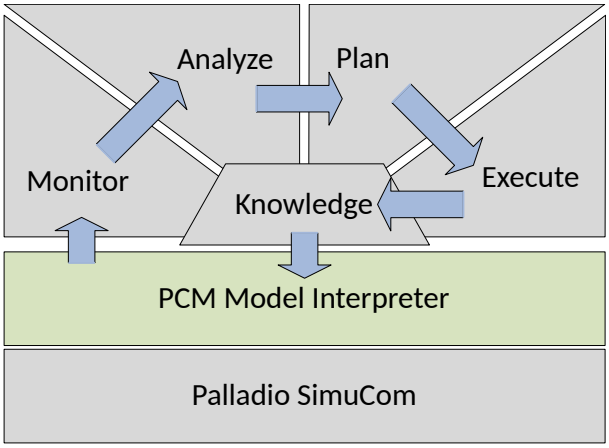


Figure 2.11.: SimuLizar architecture, adopted from[BBM13].

2.9.3. Simulation of Cloud-based Systems

Palladio Component Model

Palladio is a software component model for business information systems to enable model-driven predictions on throughput, response time and resource utilization. Component models provide many advantages over object-oriented development approaches, e.g., higher usability, quality and better test potential [BSG+17].

SimuLizar is an extension for Palladio, which was especially developed for systems that change at runtime, e.g., cloud-computing and virtualized infrastructure environments. It allows for model adaptations while the simulation is running and thereby makes it possible to simulate self-adaptive systems like microservice applications with Palladio. At that, SimuLizar is based on a MAPE-K feedback loop, as illustrated in Figure 2.11 as part of SimuLizar's architecture. The feedback loop consists of the four steps *monitor*, *analyze*, *plan* and *execute*. A *knowledge base*, containing system information, i.e., a runtime model or monitoring data, can be accessed during all steps. This allows SimuLizar to model self-adaptive systems and to predict their performance [BBM13].

The main focus of the Palladio Component Model, however, lies with performance investigation and not with the investigation of the resilience of a system.

CloudSim

Developed by the CLOUDS Laboratory of the University of Melbourne, CloudSim is a framework for modeling and simulation of cloud computing infrastructures and services. It allows modeling and simulation of large scale cloud computing data centers as

well as modeling and simulation of application containers and virtualized server hosts [Lab17]. The platform is written in Java and extensible, in fact, various extensions for the framework exist. Its architecture consists of multiple layers. The fundamental layer provides management of of applications, hosts of virtual machines, etc. while the top layer represents the basic entities for hosts and enables the generation of requests in a variation of approaches, configurations and cloud scenarios [BSG+17].

Chapter 3

Extension of Microservice Resilience Simulator

The circuit breaker pattern (Section 2.3.1) is an effective and popular resilience pattern. By increasing the resilience of system components, it reinforces an attribute that is particularly important in complex distributed systems. Therefore, the emergence of microservice applications leads to an ubiquity of the circuit breaker pattern (Section 2.4). The MiSim microservice resilience simulator is able to simulate the circuit breaker pattern. However, when the simulator was originally developed, the pattern's implementation in the simulator was not modeled after the implementation of the pattern in an actual software library [BGZ17]. This means that the simulation of microservices that implement a circuit breaker does not necessarily comply to actual metrics during the real service's productive execution. This chapter describes how the circuit breaker in the simulator is refined by adapting its implementation to the mechanics and parameters of a popular and widespread real-world implementation of the circuit breaker pattern.

3.1. Deciding on a Circuit Breaker Implementation as Reference

The most widely used library that implements the circuit breaker pattern is probably Hystrix [Hys18], a latency and fault tolerance library created by Netflix. This conclusion comes from the facts that it was difficult to find any big alternatives to Hystrix at all during the research for this thesis that do not rely in some way on Netflix's library and that most articles that talk about the circuit breaker pattern mention solely Hystrix as an example implementation [Fow14; Kum18; Ric18; Tri17]. Hystrix is a library for Java, but unofficial adaptions for other programming languages emerged, too, because of

the lack of capable alternatives. One other system that also implements the pattern is Finagle from Twitter [Twi18], although it is not just a fault tolerance library, but an entire remote procedure call system. This makes it hard to use, because a development team that wants to implement a circuit breaker for a remote operation is constraint to rely on this technology for building the whole service. Furthermore, Hystrix's popularity beats Finagles by far. Finagle's implementation of the circuit breaker works a bit different than Hystrix's. It includes two circuit breakers that are triggered by different events. First, when the connection to a host fails, the system marks the host as unavailable and does not use it until it is marked as available again. At the same time, a background process is started that repeatedly attempts to reconnect with the host; it marks the host as available upon success. Second, modules can mark themselves as unavailable once the number of observed failures reaches a certain percentage value. A module that has been marked as unavailable remains in this state for a predefined duration [Twi18].

Hystrix is far more popular than Finagle or any other implementation of the circuit breaker pattern. In addition to that, it is available outside of any frameworks as well as integrated in some of the more widespread Java-based frameworks [Spr18]. For those reasons, Hystrix is used as a reference for the refinement of the circuit breaker implementation in the microservice resilience simulator. The goal of the refinement is to increase the accuracy and realism of the simulation.

3.2. Comparision between Hystrix's and the Simulator's Implementation of the Circuit Breaker Pattern

This section discusses the differences between Hystrix's implementation and the simulator's implementation of the circuit breaker pattern and their consequences.

3.2.1. Configuration Parameters

It is possible to configure Hystrix with a range of different parameters to adjust it to different use-cases. The parameters that affect the circuit breaker are a timeout, a request volume threshold, a sleep window, an error threshold percentage and the duration of the statistical rolling window [Net18].

Timeout The timeout parameter sets the time after which the caller observes a timeout of the operation and performs fallback logic. Hystrix records this as an error. Timeouts

can optionally be turned off if desired. The default value for this parameter is 1000 milliseconds.

Request Volume Threshold This parameter sets the minimum number of requests that must be sent in a rolling window before Hystrix checks if the percentage of failed requests should trip the circuit. It exists to keep the circuit from opening on the first remote call that fails; instead, opening the circuit should be a decision made on a statistical basis. For example, if the value is 20, the circuit breaker will not open before 20 requests are received, even if all received calls fail. The default value for this parameter is 20 requests.

Sleep Window When the circuit is opened, all requests will get rejected before allowing attempts again to determine whether the circuit can be closed again. This parameter defines the amount of time for which the circuit remains in the *open* state before changing to the *half-open* state. Its default value is 5000 milliseconds.

Error Threshold Percentage This parameter sets the error percentage of executed remote calls in a rolling window at which the circuit will switch into the open state and performs fallbacks when the operation is called. Its default value is 50 percent. The error threshold percentage only becomes relevant when the request volume threshold is reached in a rolling window, because the circuit will not be tripped before that happens.

Rolling Window Duration This parameter sets the duration of the statistical rolling window. This is how long Hystrix keeps metrics for publishing, but more importantly it is how long Hystrix keeps metrics for the circuit breaker to use. Hystrix resets the number of successful requests, failures, timeouts and rejections for each rolling window to keep the recorded data relevant for the decision making if the circuit should be opened. The default value for this parameter is 10,000 milliseconds.

The circuit breaker of the microservice resilience simulator also allows for some configuration, but only with the two parameters *timeout* and *retry*. The retry parameter corresponds to the sleep window parameter of Hystrix. The lack of configuration of the simulator's circuit breaker is also reflected in its behaviour.

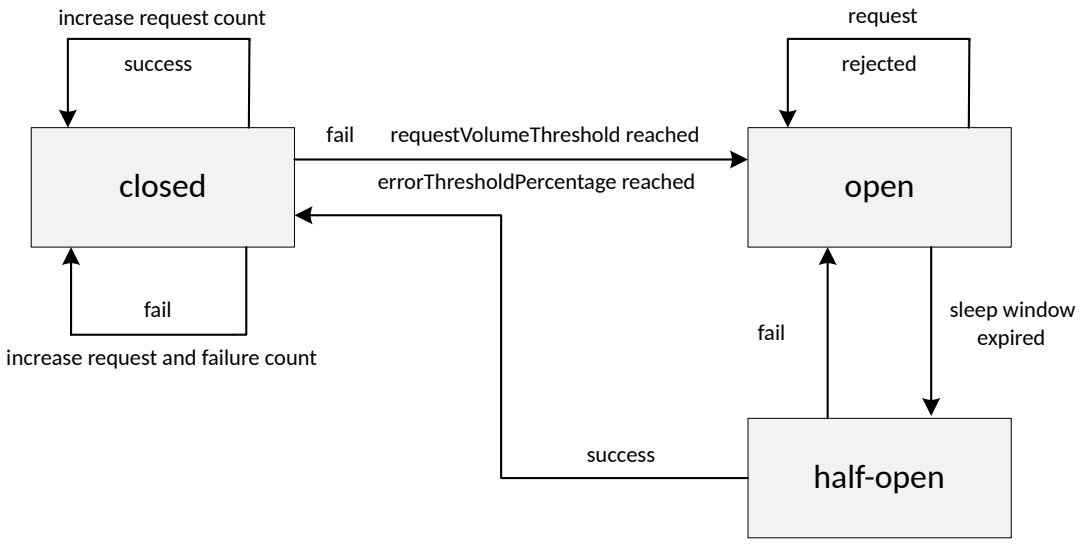


Figure 3.1.: State diagram of Hystrix's circuit breaker.

3.2.2. Behaviour

The behaviour of Hystrix's circuit breaker is described in Section 2.3.3. A state diagram of Hystrix's circuit breaker is shown in Figure 3.1. While the simulator's circuit breaker traverses through the same three states — *closed*, *open*, *half-open* — as Hystrix's, the behaviour varies quite a lot in some points.

Both circuit breakers initially start in the *closed* state. When a timeout or failure occurs in a Hystrix Command, Hystrix executes fallback behaviour and increments the error count for the current rolling window, but the circuit itself remains in the *closed* state. The circuit only switches into the *open* state, when the request volume threshold and the error threshold percentage are reached in a rolling window. If a timeout occurs in an operation that implements a circuit breaker in the simulator, the circuit breaker does not simulate the execution of a fallback mechanism; instead, it directly opens the circuit, even if this is the first request to this operation that fails. This is a relatively big difference between the two implementations that can noticeably impact the accuracy of the simulation.

Furthermore and equally impacting on its accuracy is the fact that a bug in the simulator's implementation causes only the directly called operation to get bypassed with fallback behaviour. All other dependencies that would normally be called by this operation still get simulated, even though the operation that calls them is replaced by a fallback method. The cause for this unrealistic functionality is a bug in the event scheduling and the circuit breaker implementation of the simulator. All dependencies of an operation are scheduled at the same time as the operation itself, even if the operation will get

replaced by a fallback mechanism by the circuit breaker, because the circuit breaker only removes the affected operation from the waiting queue.

Both circuit breakers transfer into the *half-open* state after a defined sleep window and let one request through to the called operation. But Hystrix throws a timeout if the request is not satisfied after a set time what causes the execution of a fallback, while the simulator keeps waiting for the execution of the operation, even if the timeout is already exceeded.

The switch from the *half-open* state into the *closed* state takes place in the same way in both circuit breaker implementations. But when the request that was let through does not complete in time, the simulator's circuit breaker remains in the *half-open* state, while Hystrix's goes back into the *open* state and resets the sleep window.

All in all, it is evident that the implementation of the circuit breaker pattern in the simulator differs in some impactful ways from Hystrix's circuit breaker implementation. This justifies a refinement of the simulator's implementation to increase the accuracy and realism of the circuit breaker simulation.

3.3. Modifications to the Input Model

The input model of the microservice resilience simulator, which was previously presented in Section 2.5, contains all the information about the simulated microservice system's architecture as well as information about the experiment.

3.3.1. Refining the Parameters of the Circuit Breaker

The parameters of the simulator's circuit breaker are specified in the input model as well, therefore, the model must be refined to fit the new circuit breaker implementation. The new model now contains the five parameters that Hystrix uses to configure its circuit breaker, in order to give an accurate representation of it in the architectural model. Listing 3.1 demonstrates how this looks like in the actual JSON file that contains the architectural model.

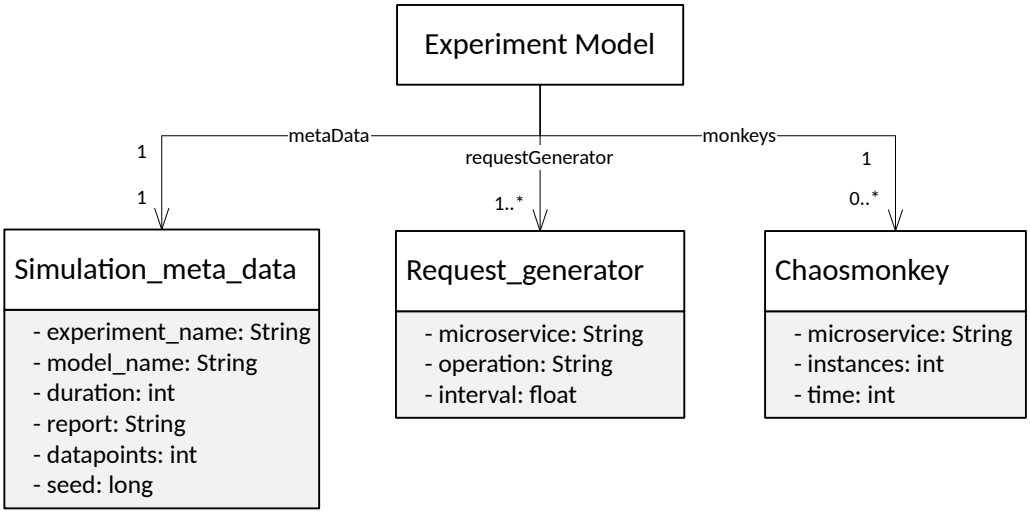
```
{
    "circuitBreaker": {
        "rollingWindow": 10,
        "requestVolumeThreshold": 20,
        "errorThresholdPercentage": 0.5,
        "sleepWindow": 5,
        "timeout": 1
    }
}
```

Listing 3.1: The representation of a circuit breaker in the architectural model.

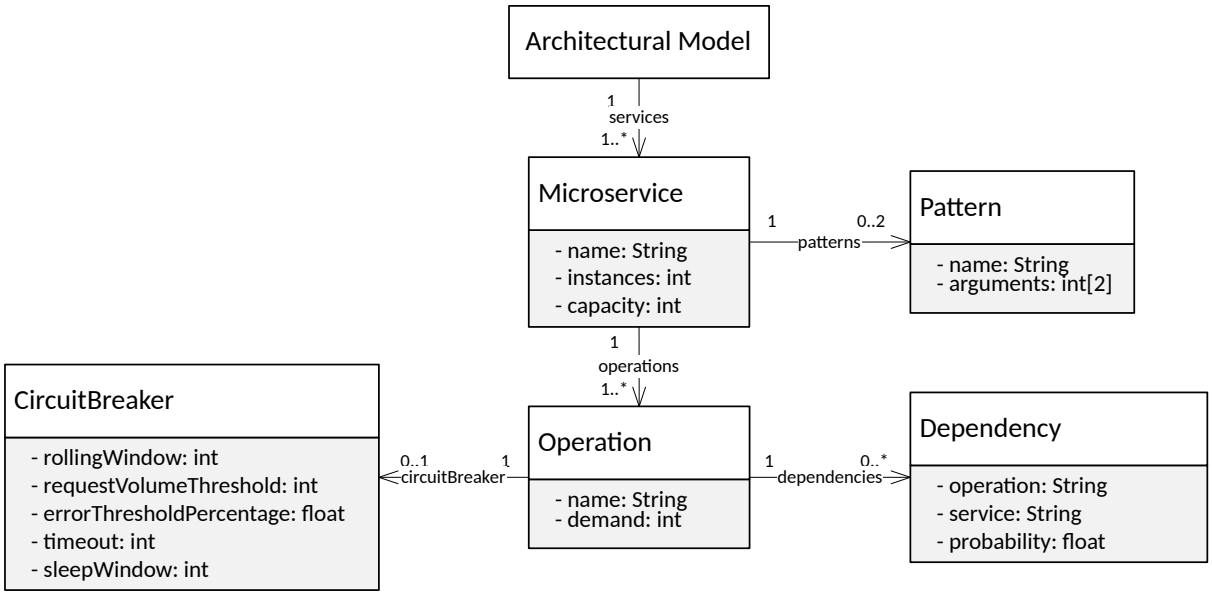
3.3.2. Splitting the Model into Architectural Model and Experiment Model

Since the simulator has only one input model that contains the information about both the architecture of the system and the experiment that is to be simulated, a new model has to be created every time that a new experiment is to be conducted if the different experiment models should be persistently saved. If the architectural information and the experiment information is saved in two separate models only the experiment model has to be saved in multiple instances. The architectural model can be reused for every experiment as long as the architecture itself is not subject to change. In addition to this, the architectural model and the experiment information come from different sources. In the presented approach, the architectural model is extracted from a microservice application, while the faultload, which is part of the experiment model, is created by the Orcas Decision Engine. If both are contained in one model, this information needs to be merged in an unnecessary intermediate step. Also, since the Orcas Decision Engine takes only the architectural model as input, it would not be possible to use the same file as input for the simulator and the Decision Engine. Because of these reasons, it makes more sense to have a separate architectural model and a separate experiment model as input for the simulator, instead of a single input model.

The new architectural model contains the microservices, their configuration, operations, dependencies, etc, while the new experiment model contains information about the experiment, like the experiment's duration, information about sampling, the request generators and chaos monkeys. Figure 3.2 shows the architectural model and the experiment model with all their parameters visualized as UML class diagrams.



(a) Experiment model



(b) Architectural model

Figure 3.2.: The experiment model and architectural model visualized as a UML class diagram.

3.4. Refinement of the Circuit Breaker Implementation

This section presents an overview over how the current implementation of the circuit breaker in the simulator works, how the implementation was refined to improve its accuracy and how the refined implementation of the circuit breaker works.

3.4.1. Current Implementation of the Circuit Breaker

Whenever the simulator schedules an operation, it checks if one of the circuit breakers of the service, which executes that operation, should change into another state. First, it processes which of the operation threads already exists for the longest time. Then, it checks if that time exceeds the timeout of the circuit breaker. If this is not the case, the circuit remains closed. If the thread life time exceeds the timeout time, the circuit transfers to the state *open*. The traversal between the different states is also illustrated in Figure 3.3.

If the circuit is already opened, the simulator checks if the sleep window is expired. If this is the case, the circuit switches to the *half-open* state, otherwise it remains in the *open* state. The first new thread that is added to the CPU when the circuit breaker is in the *half-open* state is executed instead of handled with a fallback mechanism and marked. When the circuit breakers are checked the next time, the simulator checks if

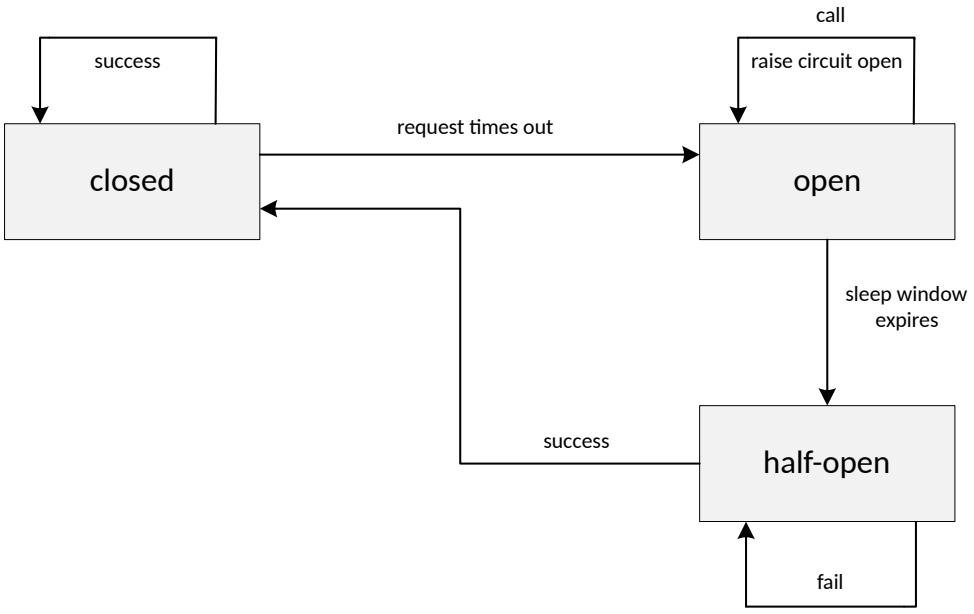


Figure 3.3.: State diagram of the old circuit breaker implementation.

the request has been completed. If that is the case, it checks if the completion was faster than the specified timeout time and puts the circuit back into the *closed* state if this requirement is full filled. Otherwise the circuit remains in the *half-open* state.

A few other things to remark are that the simulator always assumes that a circuit breaker is implemented by an operation, not a single dependency The circuit opens when one of the operation's dependencies times out, without regard to the other dependencies that an operation may contain. Furthermore, due to a bug in the implementation, all operations that implement a circuit breaker and that are executed on the same microservice instance share the same circuit breaker — even though the different operations should not effect each other. This will also be changed in the refinement of the circuit breaker implementation. And last, the simulator does not schedule actual fallback operations when the circuit is open. Instead, it takes the operations that would get replaced by fallback behaviour out of the event queue and assumes that a fallback operation would perform instantly without resource demand. This happens for simplicity, but deviates from real behaviour and could therefore be subject of future work.

3.4.2. Refined Circuit Breaker Implementation

The first thing that was changed in the refined version of the circuit breaker implementation is that the circuit state is no longer shared by all operations, which implement a circuit breaker, in one microservice instance. Instead, the circuit state of every operation of that instance is saved separately. This means that a single open circuit does not lead to all operations with a circuit breaker being dealt with by a fallback method anymore. Now only the operations, whose circuit is actually open, are being handled with a fallback, while the other operations are executed as normal.

Furthermore, the refined circuit breaker uses the same statistical calculations as Hystrix's circuit breaker in the decision making whether to open a circuit, instead of opening the circuit as soon as a single operation times out. This is made possible by the added parameters that were adopted from Hystrix. The simulator now checks for every thread that runs an operation with a circuit breaker if the thread life time exceeds the timeout. The old version of the circuit breaker implementation only checked this for the thread which existed for the longest time. If the circuit status of that operation is *closed* and the thread timed out, the simulator increments the error count in the current statistical rolling window of this operation's circuit breaker and executes a fallback for this operation. Note, that this does not lead to the circuit being opened anymore, like it did in the old circuit breaker implementation.

Already scheduled dependent operations are not impacted when an operation is falling back to a standard mechanism. This is, because a remote call cannot be undone once it was sent in a real system either.

After the simulator iterated over all threads of this service instance and every circuit breaker's error count was updated, it iterates over all circuit breaker objects from the operations of this instance and checks if the requirements for opening a circuit are fulfilled. These requirements are the same as in Hystrix; a circuit transfers to the *open* state if the request volume threshold is reached in the current rolling window and the error percentage in this rolling window is higher or equal to the error threshold percentage.

The simulator then checks for all circuit breakers of this service instance that are in the *open* state if the sleep window is passed. The circuit breakers for which this is the case are put into the *half-open* state.

The next step is to check if the circuit breakers in the *half-open* state have already let a request pass to the called operation. If this is the case, the simulator checks if the respective request takes already longer than the specified timeout. This is new, because this was previously only checked when the dependent operation already completed. By checking on the request before that the circuit breaker can switch its state earlier without having to wait for the completion of a request that may take very long if the respective service is unresponsive. The refined circuit breaker switches back into the *open* state if the request time out, where it stays until the sleep window is passed again If the request did not time out yet, the simulator checks if it was completed yet. If this is the case the circuit switches into the *closed* state again to accept requests again.

The last step of this routine is to check if the rolling window of any circuit breaker is exceeded. The error count and request count of the circuit breaker are reset if that is the case and the rolling window starts anew.

If the circuit of an operation is in the *open* state on a service instance, the simulator does not schedule any more of these operations on this instance. This behaviour is supposed to simulate the fallback mechanism of Hystrix, since concrete fallback operations are not supported by the simulator yet. But this behaviour also prevents operations from being scheduled that are being called by an operation which is bypassed by a fallback mechanism due to an open circuit.

All in all, these refinements made to the circuit breaker implementation of the simulator should make for a far more realistic and accurate simulation of the circuit breaker pattern.

3.5. Comparison between the Old and the Refined Circuit Breaker Implementation

In this section, the difference between the old and the new circuit breaker are demonstrated based on a few example simulations.

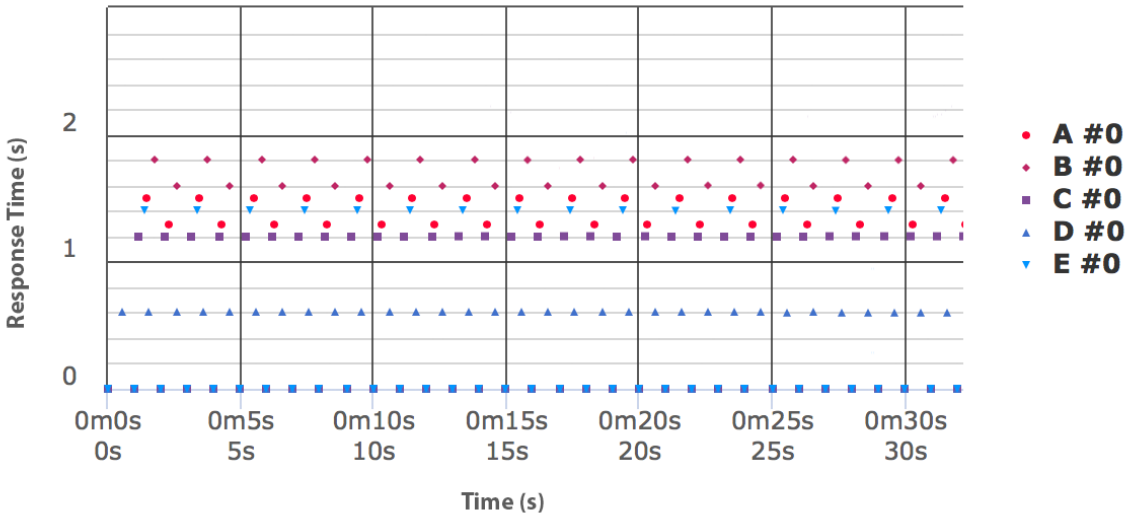
The microservice architecture used for this system is the same as in the example in Section 2.4 and consists of five services. Service A has two operations with multiple dependencies to other services in the system. Its operation a1 is getting one request per second in the first experiment and has a circuit breaker with the default Hystrix configuration, except for the request volume threshold, which is four requests per rolling window. The experiment is designed so that the operations called by a1 take longer than the specified timeout, therefore the circuit should get opened eventually and requests should be handled by the fallback method.

Figure 3.4a shows the response time in seconds for each operation with the old circuit breaker implementation, where the red dots represent microservice A.

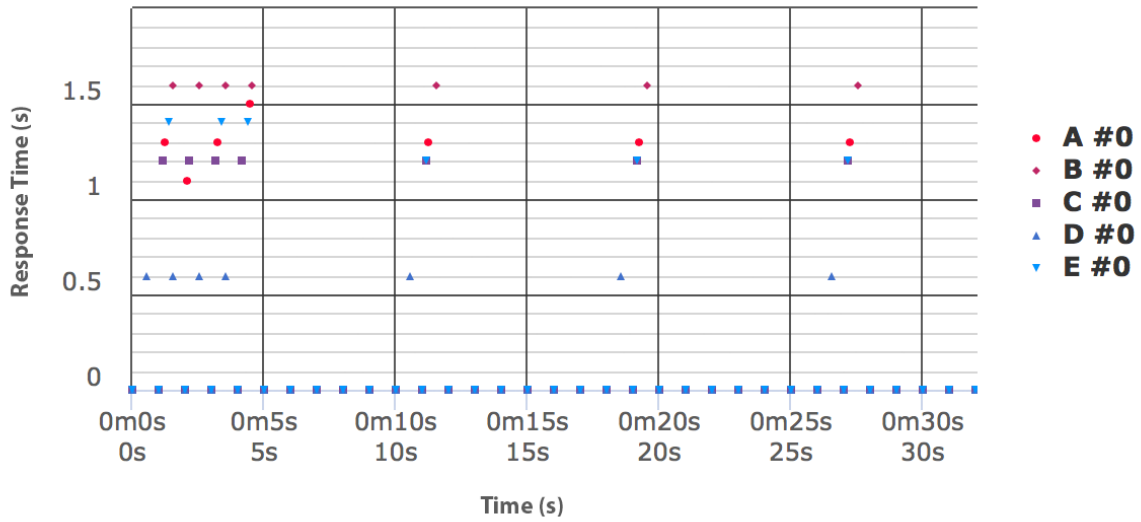
It can be seen that the response time of a1 is slightly higher than one second, yet the old circuit breaker implementation does not open the circuit. This is because the old implementation only counts the time in which a thread is active as its life time, the time in which a thread is inactive and waits for a response from a called operation is disregarded. In this case, the response time of the operation exceeds one second, while the time in which the thread of the operation is active is shorter, because the thread is inactively waiting for responses of calls to other services most of the time.

In comparison, Figure 3.4b shows the response time in seconds for each operation with the new circuit breaker implementation, which was modeled after Hystrix's circuit breaker. Microservice A is again represented by the red dots.

The response time of a1 again exceeds the timeout of one second in the first four calls, after which the request volume threshold of four requests is reached. The error percentage at this point is 100%, because all four requests timed out, since the new implementation uses the whole life time of a thread for its computations — including the time in which a thread is waiting for the response of a remote call. This exceeds the error threshold percentage of 50%, therefore the two requirements for opening the circuit are met and the circuit is opened. This is visible in the diagram, because the response time of a1 falls to zero seconds — since the simulator assumes that the fallback executes instantly. The default sleep window of the circuit breaker is five seconds; the diagram shows that one request was executed after that window to test if the called operation now responds faster than the timeout. But this request also exceeds the timeout, because the system was configured in such a way that the circuit would constantly be opened in



(a) Old circuit breaker implementation.

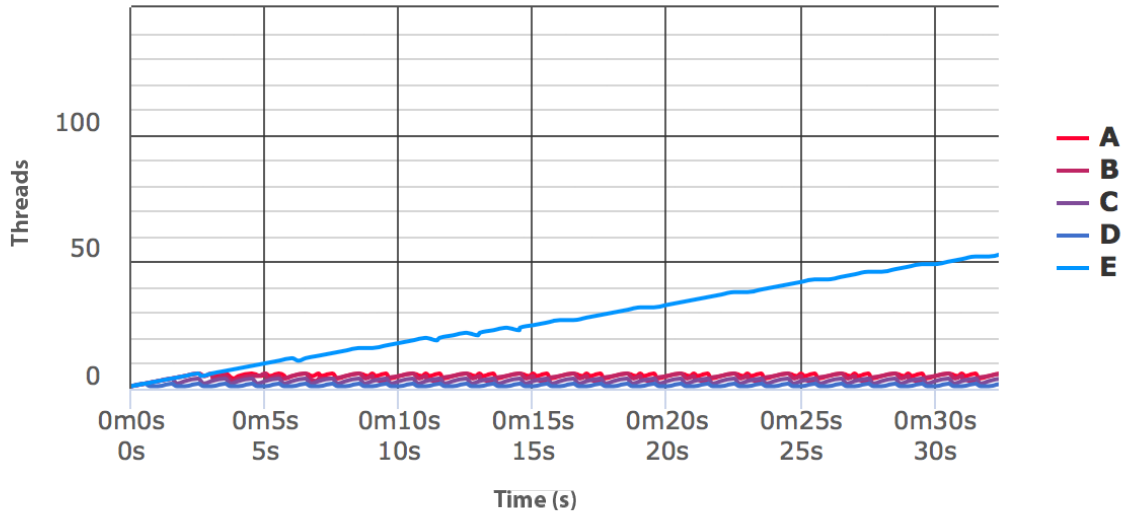


(b) New circuit breaker implementation.

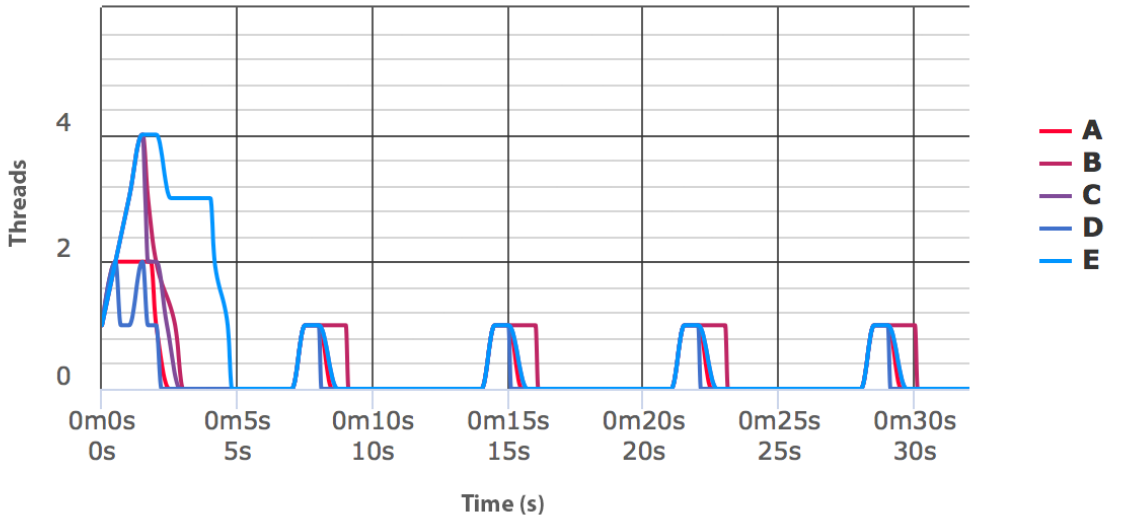
Figure 3.4.: Response time of each operation.

this experiment. Therefore, the circuit opens again after the request times out and starts handling requests with a fallback mechanism anew.

In the second experiment, operation a1 of service A receives two requests per second, putting even more load on the system. Otherwise, the architecture configuration remains unchanged.



(a) Old circuit breaker implementation.



(b) New circuit breaker implementation.

Figure 3.5.: Number of threads of each service instance.

This time the old circuit breaker opens the circuit, too, when the response time of a1 surpasses the two second mark. But the bug in the old implementation that caused the simulator too keep scheduling remote calls from operations that were instead handled by a standard fallback is clearly visible in Figure 3.5a, which shows the number of threads in each service instance.

Service E, which is represented by the blue triangles, gets a lot more requests than it can handle, since the circuit breaker does not stop service A from calling its operations. As a consequence, service E becomes even more unresponsive and keeps creating threads for incoming requests, as can be seen in the diagram.

This bug was fixed in the new circuit breaker implementation, as can be seen in Figure 3.5b.

No more new requests are send to service E as soon as the circuit of operation a1 opens, the service's operations are only called when the circuit breaker is in the half-open state to check whether the circuit could be closed again.

It can be seen that the new implementation behaves more realistic than the old circuit breaker. A comparison with the behaviour of Hystrix's circuit breaker is presented in Chapter 6.

Chapter 4

Architectural Model Extraction Approach

Both the microservice resilience simulator and the Orcas Decision Engine require the architectural model of a microservice application as input in order to simulate microservice systems and create the faultloads needed for it. Manually creating such a model per hand is a tedious and error-prone task and models of complex systems can quickly grow vast and unclear. Therefore, extracting them directly from the system is a more efficient solution that at the same time takes care of a common source of errors [BHW+15].

On the highest level of abstraction, the two kinds of architectural model extraction approaches are static system analysis and dynamic system analysis. While static analysis approaches focus on analyzing static data, like source files, configuration files or code repositories, dynamic approaches rely on performance monitoring, logging and tracing in order to achieve their goals, see Section 2.7. Often both methods are combined in order to get an accurate architectural model, as in the approach presented by Granchelli et al. in 2017 [GCD+17].

Section 4.1 presents the extraction approach that will be developed in this chapter and states the reasons for selecting that approach. Thereafter, the steps of the preparatory microservice application instrumentation are explained in Section 4.2 and the extraction of architectural information from the collected traces is detailed in Section 4.3.

4.1. Deciding on an Extraction Approach

Extracting the architecture of a microservice system is not an easy task, because they are often vast and complex. Services often have dependencies to other services, what leads to a confusing network of different microservices working together [GCD+17]. Developing an architectural model extraction approach for microservice applications can probably be a big enough task to justify a thesis for itself. Yet it is only one part

of this thesis and, therefore, we decided to keep the approach as simple as possible. By largely preparing the microservice system before the actual extraction it is possible to keep the extraction approach itself straight forward and easy. For this reason we chose to perform dynamic analysis on the microservice applications, to be specific via tracing (Section 2.8.2), because the main effort of this approach would be to instrument the analyzed system with a distributed tracing tool. Afterward, the actual extraction approach must only process the recorded traces and construct the architectural model from that data.

Why OpenTracing?

We decided to use a distributed tracing system that implements the OpenTracing standard (Section 2.8.2) for the extraction, because it is growing to be the de facto standard for distributed tracing. At the moment it is quickly being adopted by big vendors and small organizations alike and on its way to become the universal tracing standard where more and more software libraries come with built-in support for OpenTracing instrumentation to enable distributed tracing without vendor limitations [Šab18a]. This also means that it will be less effort to instrument microservice applications for tracing in order to extract the architectural model from them that is required for the simulator and the Decision Engine.

Why Jaeger?

Besides Jaeger (Section 2.8.3) there are a few other tracing systems out there that support the OpenTracing standard, like Appdash, LightStep, Hawkular and Instana [OT18]. While Jaeger is still fairly new it already accumulated a high number of adopters, has a low overhead, supports dynamic sampling and is easily scalable, according to Šabić [Šab18b]. Furthermore, it allows for span tags, a benefit that will prove valuable in the developed architectural model extraction approach. And even more, Jaeger offers client libraries for six different programming languages [Šab18b], which means it can be used to instrument pretty much every microservice regardless of the employed technology stack and, therefore, most microservice applications can be instrumented with it in preparation for our architecture extraction approach. These advantages let to the decision to make Jaeger the tracing tool of our choice, but in the end it comes down to personal preferences, since it would have been possible to achieve similar results with other tracing implementations of OpenTracing as well.

4.2. Preparatory Service Instrumentation

The developed architectural model extraction approach presented in this chapter requires the respective microservice system to be instrumented with a distributed tracing tool that implements the OpenTracing standard as preparation for the actual extraction. As explained above we decided to use Jaeger for this purpose. This section gives an explanation of how a service should be instrumented to create the tracing data necessary for the architecture extraction. Exemplary the following explanation is for services that have been developed with SpringBoot, a framework for Java web applications. But the process does not vary much for services implemented in another programming language or framework, as Jaeger and OpenTracing offer client libraries for a multitude of them.

4.2.1. Creating a Tracer with Jaeger

The first thing to do when instrumenting a service with OpenTracing is to create the tracer object that will record the traces and their spans and send them to the jaeger-agent service. In SpringBoot this must be done in the *SpringBootApplication* class in a method that creates the tracer as a Spring bean. The Jaeger client library allows a tracer to be configured with various parameters, some of them are required for the tracer object to be instantiated successfully. It is necessary to specify a service name, otherwise Jaeger will throw an exception during runtime. For the extraction, this service name should be identical with the actual name of the microservice, because it will later be used in the architectural model. Further, the host and the port on which the jaeger-agent runs should be specified in the tracer, if the service and jaeger are not running on the same host. In that case the tracer also needs the jaeger sampler manager host and port to successfully obtain the sampling strategy from the jaeger-agent. As shown in Listing 4.1, it is also possible to read the tracer configuration from environment variables, which allows for the creation of tracers that are not hard-coded into the service. This is especially helpful when running services and the jaeger-agent in Docker containers, because the container that is running the jaeger-agent must be declared in the configuration of the tracer object as the host of the jaeger-agent, since the default host would otherwise be localhost.

```
@Bean
public Tracer jaegerTracer() {
   Configuration config = Configuration.fromEnv();
   return config.getTracer();
}
```

Listing 4.1: Creation of a tracer bean in SpringBoot.

```
public ResponseEntity<String> fallback() {
   jaegerTracer.buildSpan("fallbackMethod")
            .asChildOf(jaegerTracer.activeSpan())
            .startActive(true);
   ResponseEntity<String> response = new ResponseEntity<String>("Fallback",
       HttpStatus.OK);
   jaegerTracer.activeSpan().finish();
   return response;
}
```

Listing 4.2: Instrumenting the fallback method of a circuit breaker with Jaeger.

Once the tracer object has been created, Jaeger will record traces for any remote calls that are annotated with the *RequestMapping* annotation from Spring, without any further instrumentation of the code. It is possible to create spans for other methods as well and inject them into the trace, if it is desired to include methods that are not annotated as *RequestMapping* or that do not perform network calls.

4.2.2. Recording Circuit Breaker Implementations with Jaeger

If some or all of a services' remote calls implement a circuit breaker, i.e., by integrating the Hystrix library and wrapping the remote calls in Hystrix Commands, more instrumentation is required to record the pattern in the traces, what is essential to including the circuit breaker in the extracted architectural model of the application.

By default, the created tracer object cannot record the implementation of the circuit breaker pattern. Even when the circuit is opened and the fallback is executed, there will not be an occurrence of this in the trace. One possible solution to this could be to manually create a span in the fallback method and inject it into the trace, as shown in Listing 4.2.

But, because of the way that Hystrix triggers the fallback method, Jaeger creates two traces in this case. One trace contains the spans from the remote call, which was executed and then caused the circuit to open, because Hystrix only throws an *InterruptedException* in this case and has no further way of stopping the execution of a remote call that has already sent its request. The other trace only contains the span that was manually created in the fallback method and any other spans that could have been created as a consequence of the fallback. Jaeger does not inject these spans into the first trace, because it does not know that they are executed in the context of the original method that caused the circuit breaker to open. To record an extra fallback trace is not very

```
@RequestMapping(value = "/a", method = GET)
@HystrixCommand(fallbackMethod = "fallback")
public ResponseEntity<String> a() {
   jaegerTracer.activeSpan().setTag("pattern.circuitBreaker", true);
   jaegerTracer.activeSpan().setTag("pattern.circuitBreaker.fallback", true);
   restTemplate.getForObject("http://b:8080/c", String.class);
   restTemplate.getForObject("http://c:8080/e", String.class);
   jaegerTracer.activeSpan().setTag("pattern.circuitBreaker.fallback", false);
   return new ResponseEntity<String>("Operation a executed successfully.", HttpStatus.OK);
}
```

Listing 4.3: Adding span tags to record circuit breaker pattern implementations in the trace.

helpful in the extraction of the architectural model, because we want to extract which operations implement a circuit breaker. This data only gives us the data that an operation of a certain service implements the pattern, therefore, another approach is needed to include the necessary information in the traces.

Since it is not possible to connect the spans from the original operation and its fallback, it is not needed to manually create a span in the fallback method. Instead we take advantage of the fact that Jaeger supports an OpenTracing feature called *span tags*. A span tag represents contextual metadata and consists of a sequence of key-value pairs, where keys are strings and values can be strings, numbers, booleans or dates [Šab18a]. It is possible to manually inject tags into a span to add information to it that is relevant for a specific request. In a workaround to include the circuit breaker data, the second thing to do, when instrumenting a service in preparation for the architectural model extraction, is to add span tags to the active span in every operation that implements a circuit breaker with the key *pattern.circuitBreaker* and the boolean value *true*. If it is also desired to record if the circuit breaker fallback method was executed, one span tag should be added at the start of the method with the key *pattern.circuitBreaker.fallback* and again the boolean value *true* and another at the very end of the method with the same key and the boolean value *false*, as seen in Listing 4.3.

This way, the span tag will save the value *false* if the method executes successfully and *true* if the execution gets interrupted by the circuit breaker falling back on the fallback operation. It would also be desirable to be able to include different parameters of the circuit breaker configuration in this way, but no apparent way of accessing them outside of the method's *HystrixCommand* annotation exists in SpringBoot. Therefore, the parameters cannot be extracted automatically at the moment, but it would certainly

be a target of future work to include the circuit breaker configuration in the extraction process.

The microservice system is prepared for the architecture extraction once every service of the application has been instrumented in this way with Jaeger. In the next step every user-facing operation of the application must be executed at least once for Jaeger to record the traces that will later be used in the extraction process. It does not matter if operations are called multiple times, but it is crucial that every operation is invoked at least once, because it is otherwise missing in the traces and, therefore, cannot be extracted into the architectural model.

4.3. Extracting Architectural Information from Jaeger Traces

When every service of the application is instrumented and traces for every possible transaction in the system have been recorded, all the data that is needed for the extraction is gathered. You can find an exemplary trace in Listing 4.4 that demonstrates which data can be read from them, as also explained in Section 2.8.2. Please note that some insignificant information contained in the trace is not shown in the listing to save space. Jaeger comes with an API that can be consumed to get information about the instrumented system and traces for the instrumented services. As part of this thesis we created an extraction tool, written in Java, that processes this data to build the architectural model that can be used as input for the microservice resilience simulator and the Orcas Decision Engine. The tool cannot extract all architectural data that is needed by the simulator, but it extracts the services, the operations of every service, the circuit breaker for each operation and the dependencies between all the operations and services. Other parameters of the architectural model, like the number of service instances, resource demand, available resources in each service instance, etc, cannot be extracted with this approach for now. Future work on the developed extraction approach could certainly be to expand its capabilities in this regard to make it even more comfortable to extract architectural models without having to manually insert these parameters. They are filled with default values in the current state.

The tool first requests the names of all services from the Jaeger API. Listing 4.5 is an example of the response by the API that gets processed by the extraction tool to get the service names. As illustrated, the JSON response also contains the jaeger-query service that is filtered out.

It will use this data in the architectural model, but also to send further requests to the API. The API can also return the names of all the operations that it has saved for one

```
{
  "traceID": "8f833961ebe3c8b7",
  "spans": [
    {
      "traceID": "8f833961ebe3c8b7",
      "spanID": "2b52e0ef905d327c",
      "operationName": "a1",
      "references": [
        {
          "refType": "CHILD_OF",
          "traceID": "8f833961ebe3c8b7",
          "spanID": "8f833961ebe3c8b7"
        }
      ],
      "startTime": 1532943589380000,
      "duration": 2656,
      "tags": [
        {
            "key": "pattern.circuitBreaker",
            "type": "bool",
            "value": true
        }
      ],
      "logs": [],
      "processID": "p1"
    }
  ],
  "processes": {
    "p1": {
  "serviceName": "A",
      "tags": [
        {
          "key": "hostname",
          "type": "string",
          "value": "3aa65e8e05b7"
        },
        {
          "key": "ip",
          "type": "string",
          "value": "172.23.0.7"
        }
    ]}}
```

Listing 4.4: Examplary Jaeger trace.

```
{
    "data": ["jaeger-query","A","D","B","E","C"],
    "total": 6,
    "limit": 0,
    "offset": 0,
    "errors": null
}
```

Listing 4.5: Response from the Jaeger API that contains the names of all services.

service. The extraction tool takes advantage of this functionality in the next step, where it sends requests for every microservice to Jaeger to retrieve their operations. With this data it is already possible to build a model of the architecture that contains the different services and their operations. What is still missing is the information if an operation implements the circuit breaker pattern and the dependencies between operations. The Jaeger API can return the dependencies between a system's services, what is also a useful feature, but does not give us the required inter-operation dependencies. It is necessary to process the recoreded traces to obtain these.

It is possible to use the API to search for the traces of a specific service. The extraction tool, again, does this for every service before iterating through the traces and reading the spans contained in each trace. Each span contains the name of the operations that it records and can, hence, be mapped to the operations of the different microservice. Next, the span tags are read to determine if the operation has a circuit breaker. If that is the case, the circuit breaker pattern will be added to this operation in the architectural model with the default parameters from the configuration of Hystrix. After that, it is checked whether the span contains a reference to a parent span. When this is the case the tool looks for the parent span and the operation that maps to it and then adds a dependency to the child operation to it. The tool iterates over all traces of every service and over all spans of every trace and, thus, adds every dependency to the architectural model that was called when traces were recorded. A dependency is only added once to an operation, dependencies that were recorded in multiple spans do not lead to multiple occurrences of the respective dependency in the model. The tool also counts the number of hosts on which a service is running. That number is used to define the number of instances for every service. Table 4.1 presents an overview over how the model attributes are extracted from the information contained in the traces.

After all these steps are successfully completed the extraction tool creates a JSON file that contains the architectural model of the instrumented microservice system. When

Trace	Architectural Model
operationName	operation.name
reference	dependency
pattern.circuitBreaker	operation.circuitBreaker
hostname	service.instances

Table 4.1.: Transformation of trace information to architectural model attributes.

required, it can also save the data that was gathered from Jaeger in a directory on the hard drive and later read those files again to extract the architectural model anew. As aforementioned, the model contains only default values for the parameters that cannot be extracted directly from the system for now, but it can already be used as input for the simulator. Though it makes certainly sense in most cases to adapt these parameters to model the real-life conditions of the system. Nonetheless, the created model already contains all the information that is necessary for the Orcas Decision Engine to create resilience experiments for the system and can directly be parsed to fit the input model of the Decision Engine. That process is described in detail in Chapter 5.

4.4. Summary and Discussion

As presented in this chapter, tracing can be used effectively to extract simple architectural models from microservice applications, but it also comes with limitations. While it is possible to extract services, operations, patterns and dependencies with this approach, the preparation before the architecture extraction itself, i.e., the instrumentation of the system's services, can be time consuming if the system is large and composed by a grand amount of services. On the other hand, tracing — and the OpenTracing standard — is getting adopted by more and more development teams and organizations, especially when they work on microservices. If a system is already instrumented for tracing only a small configuration change would be necessary to enable the extraction of architectural models from such a system. Therefore, this problem could become negligible over time. Another shortcoming of the approach is the fact that the created models do only contain default values for the parameters that are not extracted for now. This information must be added manually, though the information that gets extracted into models with this approach is the most critical and already meets the requirements of the Orcas Decision Engine. Still, it would be desirable to extend the architectural model extraction in future work to save manual steps in the model creation if the model is used as input for the microservice resilience simulator.

Chapter 5

Connection of Simulator and Orcas Decision Engine

This thesis is created in the context of the Orcas project [Orc18], which has the goal of effective resilience benchmarking of microservice architectures. Resilience benchmarking [VMSK12] aims to assess failure tolerance mechanisms, e.g., via fault injection [NCM16]. It is not only conducted in development and staging environments but also during a system's production use [BBR+16]. Netflix's Simian Army [Net16] is a popular example of a resilience tool that is used in production. But one of the disadvantages of failure injection is that resilience experiments take a lot of time and effort [NCM16]. Especially the decision making for the selection of experiments remains a challenge; the state of the art is to simply inject failures randomly into the system [BBR+16]. By incorporating architectural knowledge and knowledge about the relationships between (anti) patterns and suitable failure injections, Orcas aims to detect resilience vulnerabilities more efficiently [HDAP18].

Figure 5.1 illustrates the approach of this thesis and how it fits into the Orcas project. The Orcas Decision Engine [HDAP18] takes architectural information about a microservice application and uses it to compute resilience experiments, i.e., it suggests operations, based on probability calculations, for which it is most likely that a failure injection leads to an impact on the system's non-functional attributes. The decision engine includes the results of those experiments into further iterations of its computations to create more accurate results with a higher likelihood of finding resilience weaknesses in the system. It takes a lot of effort to inject failures into a system [NCM16], that is why the presented approach includes a connection between the decision engine and the microservice resilience simulator. Simulating the initial experiments and using their results as input for further iterations of the decision engine's computations to create more promising experiments is a faster way of gathering the necessary information that is more efficient than injecting random failures into the real system. Instead,

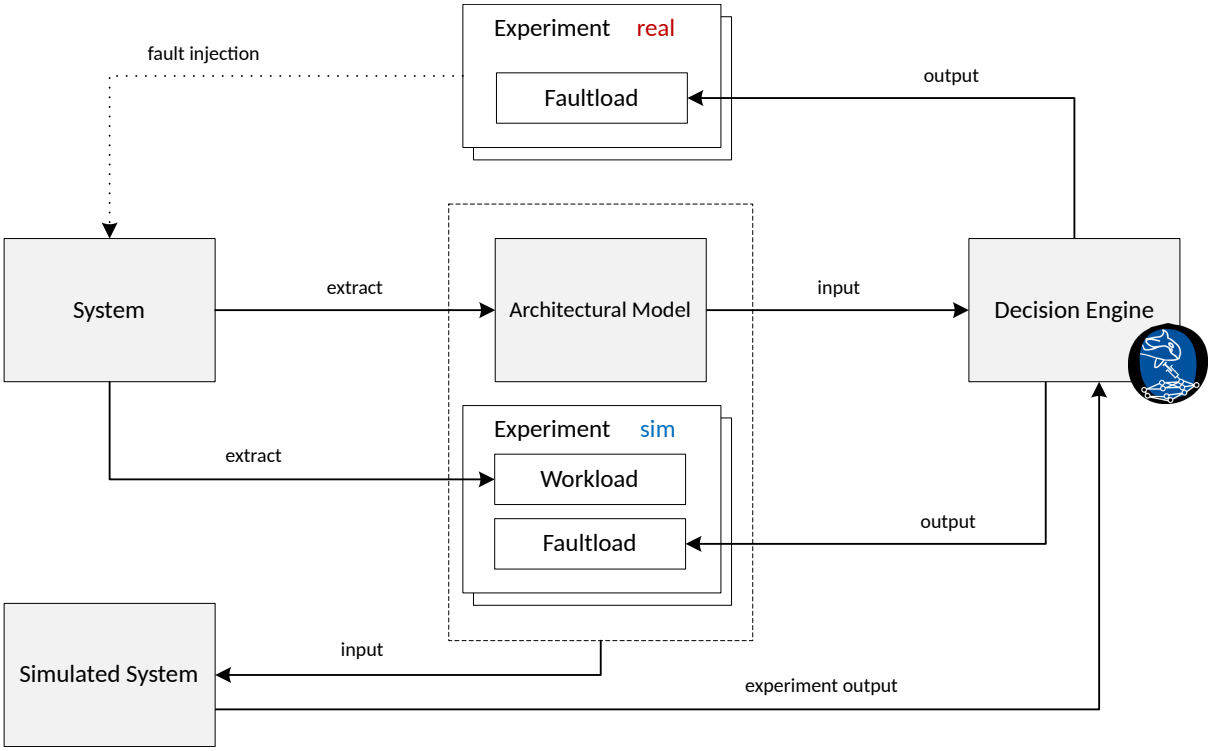


Figure 5.1.: Orcas context and solution approach.

only the resilience experiments that the decision engine deems promising after multiple computation iterations need to be run on the system, with a higher likelihood of exposing resilience vulnerabilities.

5.1. Orcas Decision Engine

The decision making of the decision engine is based on a Bayesian network [HDAP18]. An explanation of that type of statistical model will preempt the description of the decision engine's approach.

5.1.1. Bayesian Networks

A Bayesian network is a probabilistic graphical model in which different variables and conditional dependencies between them are represented via a directed acyclic graph. The nodes of that graph represent variables, e.g., observable quantities, latent variables, unknown parameters or hypothesis, while edges between them represent conditional

dependencies. Variables are conditionally independent of each other if there is no connection between them. Behind each node is a probability function. It takes a set of values for the node's parent variables as input and describes the probability of the variable that is represented by that node [Pea14].

Bayesian networks are the basis for different algorithms that perform Bayesian inference. Bayesian interference is a method of statistical inference that uses Bayes' theorem to update a hypothesis once more information is available.

$$P(A|B) = \frac{P(B|A) \cdot P(B)}{P(A)} \tag{5.1}$$

Equation 5.1 states Bayes' theorem, where $P(A|B)$ is a conditional probability — the likelihood of event A occurring after B is observed, also called the *posterior probability* — $P(B|A)$ is also a conditional probability and $P(A)$ and $P(B)$ are the probabilities of observing A and B independently of each other. $P(A)$ is also called the *prior probability*, because it is the estimate of the probability of A prior to the occurrence of B and $P(B)$ is called marginal probability. The theorem describes the probability that an event occurs or that a hypothesis is true, based on prior knowledge of conditions or variables that might be related to it [NJ09].

5.1.2. Decision Making

The decision engine's approach is request-based. That means each request is tagged as either successful or failed. The input to the algorithm consists of the following constituents [HDAP18].

- Architectural information
  - c_i: the criticality of an operation i
  - d_i: the number of other operations depending on operation i
  - $pa_{i,j,k}$: equal to 1 if pattern k is implemented in the interaction between operations i and j, 0 otherwise
- Faultload
  - $e^t_{f,i}$: equal to 1 if fault f is injected into operation i at time step t in the current experiment

The output is the probability that that the hypothesis for metric m is rejected ($h_m = 1$), given a certain combination of the four input parameters. Equation 5.2 expresses this formally, where $n(h_m = 1 \wedge c_i \wedge d_i \wedge pa \wedge e^t_{f,i})$ is the number of instances with a parameter setting that resulted in a rejection of the null hypothesis and $n(c_i \wedge d_i \wedge pa \wedge e^t_{f,i})$ is the total number of instances with that parameter setting.

$$P(h_m = 1 | c_i, d_i, pa, e^t_{f,i}) = \frac{n(h_m = 1 \wedge c_i \wedge d_i \wedge pa \wedge e^t_{f,i})}{n(c_i \wedge d_i \wedge pa \wedge e^t_{f,i})} \tag{5.2}$$

The next experiment is selected through a roulette wheel selection strategy that is based on the conditional probabilities calculated by equation 5.2. The aim of the experiment is to reject the hypothesis that the failure injection does not influence the specified metric, i.e., find faults in the system that impact the metric. The observation on the hypothesis is recorded, but more than one measurement is required to calculate the conditional probability. The random fault injection is repeated until a sufficient number of hypotheses has been recorded. After the initial experiments with random fault injections, the next experiment is determined by the calculated conditional probabilities. This ensures that parameter value combinations that are more likely to reveal vulnerabilities are selected more often for experiments [HDAP18].

5.2. Connecting the Simulator and the Decision Engine

To connect the Orcas Decision Engine with the developed architectural model extraction approach and the microservice resilience simulator, which will simulate the experiments suggested by the decision engine, it is necessary to transform the extracted architecture model into the model that the decision engine understands and to map the output of the decision engine on the operations of the respective microservice system.

5.2.1. Transformation of Architectural Information for Decision Engine

The decision engine requires architectural information in order to suggest the next experiment. We defined an architectural model in Section 3.3 that is able to describe the architecture of a microservice application by representing its services, operations, resilience patterns and dependencies between operations. But the decision engine does not need all information that is contained in this architectural model. More importantly, it also requires some architectural information that is not directly saved in the developed model. The decision engine takes d_i as input, the number of operations that depend

```
{
    microservices: [
        {
            "name": "A",
            "operations": [
                {
                    "name": "a1",
                    "dependencies": [
                        "service": B,
                        "operation": "b1"
            ]}]
        },
        {
            "name": "B",
            "operations": [
                {
                    "name": "b1",
                    "circuitBreaker": true
            }]
        }
    ]
}
```

Listing 5.1: Simplified architectural model instance describing operation b1.

on an operation i, but the architectural model only states how many dependencies an operation has to other operations. It is possible to compute d_i from the architectural model by checking how many operations have a dependency to operation i. This makes it possible to transform an extracted architectural model into the model that the decision engine takes as input. That model also contains a parameter c_i, called the criticality of an operation i. As for now, it is not possible to obtain this parameter from the architectural model and we will ignore it in our approach.

As an example we will take operation b1 from microservice B of the microservice system presented in Section 2.4. Listing 5.1 shows a simplified version of an instance of the architectural model that describes operation b1 and operation a1, which has an dependency to b1.

	d	pa	h
0	1	1	0

Table 5.1.: Operation b1 represented as interaction for the decision engine.

Table 5.1 shows the respective representation of operation b1 in the decision model's input model for the first iteration. The first column contains an index, column d contains the number of operations that depend on b1, column pa states if operation b1 is implementing a pattern and column h states if the hypothesis for this interaction was rejected or not — the value is zero, because no experiments were conducted before the first iteration.

The decision engine performs these model transformations when it is presented with an extracted architectural model as input [HDAP18]. This concludes the connection between the developed architectural model extraction approach and the decision engine.

5.2.2. Simulation of Suggested Experiments

The result of every iteration of the decision engine algorithm is a new experiment that should be run to gather more information about a hypothesis for the next iteration. An example for that output is shown in Listing 5.2.

```
['dependencies:1;patterns:0']
```

Listing 5.2: Result of one iteration of the decision engine algorithm.

This output means that the next experiment should be a failure injection into an operation that has one other operation depending on it and does not implement a resilience pattern.

To connect the decision engine with the microservice resilience simulator, a simple tool was implemented that outputs the system's operations that fulfill the specifications of the next experiment. The program takes the architectural model and the next experiment that was created by the decision engine as input. The program then looks for all operations that fit the parameters of the experiment and returns them. The experiment models needed by the simulator to run those experiments are not yet automatically created; instead, it is necessary to manually adapt the experiment model by adding a chaos monkey for one of the returned operations. A goal of future work could be to

automatically create the chaos monkeys in the experiment model, too, in order to extend the grade of automation of the Orcas approach.

5.3. Example

To illustrate the created approach as a whole we present an example based on the microservice system illustrated in Section 2.4. The first step is to extract the architectural model from the system, as it serves as input for the decision engine, which will suggest an experiment to be simulated, and the simulator as well. The system's services each need to be instrumented with the Jaeger distributed tracing system. After this is done, we need to record traces by executing every operation from every service at least once, in order to create the tracing data from which the architectural model will be extracted. Once all the necessary traces have been recorded, we execute the architectural model extraction program that was developed as part of this thesis.

An excerpt of the created architectural model can be seen in Listing 5.3. To save space, the excerpt does not contain the parameters of the circuit breakers. It contains all five services and all their operations, except for one. It is not possible to execute operation b2 of service B, because it cannot be called the user, neither is it called by any other operation. Therefore, there is no way to record it in the traces and extract it into the architectural model. But this should not be a problem for our experiments, since this operation is never called during the execution of the actual service anyways.

Once we extracted the architectural model it can be transformed by the decision engine into the model shown in Table 5.2.

When we run the decision-making algorithm with this model as input, we get the experiment shown in Listing 5.4 as a result.

	d	pa	h
0	0	1	0
1	0	1	0
2	1	0	0
3	1	0	0
4	1	0	0
5	1	0	0
6	1	0	0
7	1	0	0

Table 5.2.: Transformed architectural model.

```
{
  "microservices": [
    {
      "name": "A", "instances": 1, "patterns": [],
      "capacity": 1000,
      "operations": [
        {
          "name": "a1",
          "demand": 100,
          "circuitBreaker": true,
          "dependencies": [
            {
              "service": "B", "operation": "b1", "probability": 1.0
            },{
              "service": "C", "operation": "c1", "probability": 1.0
            }]},
        {
          "name": "a2",
          "demand": 100,
          "circuitBreaker": true,
          "dependencies": [
            {
              "service": "C", "operation": "c2", "probability": 1.0
            }]}]},
    {
      "name": "B", "instances": 1, "patterns": [],
      "capacity": 1000,
      "operations": [
        {
          "name": "b1",
          "demand": 100,
          "circuitBreaker": null,
          "dependencies": [
            {
              "service": "D", "operation": "d1", "probability": 1.0
            },{
              "service": "E", "operation": "e1", "probability": 1.0
            }
        ...
```

Listing 5.3: Extracted architectural model.

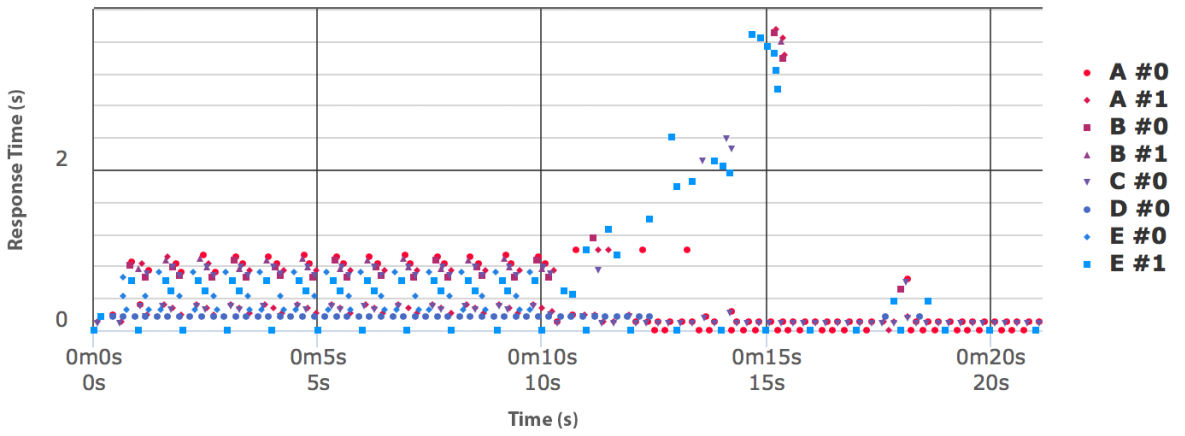


Figure 5.2.: Response times of the fault injection experiment reported by the simulator.

```
['dependencies:1;patterns:0']
```

Listing 5.4: Suggested next experiment.

The result tells us that the next experiment should be conducted on an operation that has 1 other operation depending on it and that does not implement a resilience pattern. Since this is the result of the first iteration of the algorithm, the suggested fault injection was selected randomly.

In order to find out which operations fit this requirements, we execute the developed tool, with the architectural model and the suggested experiment as input. The tool outputs the operations which correspond to the required specifications, in this case that are the operations b1 of service B, c1 and c2 of service C, d1 of service D, and e1 and e2 of service E.

From these operations, operation e1 is selected for a fault injection in this example. The following hypothesis is formulated:

- H_0: The response time of the system does not change after a fault is injected in operation e1 of service E.

The fault injection experiment is simulated on the microservice resilience simulator, which creates the report of the response times of the system that is illustrated in Figure 5.2. It is obvious that the circuit breakers implemented in operations a1 and a2 of service A prevent the injected fault from impacting the response time of the system. Therefore, hypothesis H_0 is accepted, what is recorded in the observations for the next iteration of the decision-making algorithm.

Chapter 6

Evaluation

This chapter describes the conducted evaluation of the developed approach for simulation-based evaluation of resilience antipatterns in microservice architectures. Section 6.1 states the goals of the evaluation and Section 6.2 explains the methodology that will be followed in the evaluation. After that, Section 6.3 describes the two microservice applications that are used in the evaluation, while Section 6.4 describes the settings of the experiments that are conducted in the evaluation. The results of those experiments are described in Section 6.5 and subsequently discussed in Section 6.6, which also presents possible threats to the validity of the evaluation.

6.1. Evaluation Goals

The goal of the evaluation is to investigate the research questions RQ3, RQ4, RQ4.1 and RQ4.2 that have been stated in Section 1.3. The goal of investigating these questions is to get an understanding of how accurate the architectural models created with the developed architectural extraction approach represent the respective system. Furthermore, to understand how accurate the results of the simulations of the extended microservice resilience simulator are compared with the real execution of the regarding system.

RQ3 Do the extracted architectural models accurately represent the architecture of the corresponding application?

In this evaluation, an architectural model represents a microservice architecture accurately if the model contains exactly the microservices that make up the application, the number of their instances, all their operations, every implementation of the circuit breaker pattern in an operation, and all dependencies between operations and services.

RQ4 How accurately does the simulation of a microservice system through the microservice resilience simulator correspond to the execution of the real system?

This research question is split into the following two sub-questions to give a better assessment over the simulators accuracy and realism in its simulation behavior.

RQ4.1 Does a simulated microservice system show the same behavior as the corresponding real microservice system after a fault has been injected into one of its services?

The goal is to investigate if both the simulated system and the executed system show the same changes in their response times.

RQ4.2 Does the simulation of the circuit breaker resilience pattern through the microservice resilience simulator correspond to the actual behavior of the real implementation of a circuit breaker?

The simulator's circuit breaker was implemented after Hystrix's circuit breaker implementation. The goal is to evaluate if the simulated circuit breaker shows the same behavior as Hystrix's, i.e., if the state transitions of the circuit breakers correspond to each other.

6.2. Evaluation Methodology

The evaluation is conducted on the basis of two microservice systems. To investigate RQ3, the developed architectural model extraction approach is used to extract an architectural model from each system. These models are then checked for completeness and correctness by comparing them with already existing architectural models of the systems, which were created manually. Subjects of the comparison are the represented microservices, the number of their instances, their operations, the implementations of the circuit breaker pattern in the operations, and the dependencies between services and their operations.

Furthermore, a number of experiments is conducted in a lab setting in order to investigate the remaining research questions RQ4, RQ4.1 and RQ4.2. The microservice applications are executed in a Docker environment, as well as simulated through the microservice resilience simulator, which was extended as part of this thesis. The architectural models of the systems that serve as input for the simulations are created by hand to ensure they represent the systems correctly. The system is monitored with Jaeger during each

	Generated Microservice System	Library Management System
RQ3	1.1	1.2
RQ4	2.1	2.2
RQ4.1	3.1	3.2
RQ4.2	4.1	4.2

Table 6.1.: Overview over the experiments conducted in the evaluation.

experiment since the services are already instrumented with the tracing system for the evaluation of RQ3. The recorded tracing data is compared with the data that the simulator creates during the simulation of the same experiment. More precisely, the response time of the operations is compared between the system and the simulation. Two experiments are carried out for each research question, i.e., one experiment on each exemplary system for each research question. Apache JMeter is used as a load driver to generate HTTP requests to the services during the experiments. During the experiments that investigate RQ4.1 and RQ4.2, faults are injected into the systems. The only fault that can be simulated by the microservice resilience simulator in its current state is the shutdown of an entire service instance. This is achieved by letting JMeter execute a shell script at a specified time point during the experiment that shuts one Docker container down in which the instance of a designated service is running. The CPU resources of the Docker containers that run the services are limited to make it easier to model the available resources in the simulation.

After an experiment is concluded, the response times of the different operations of the executed system, which are saved in the recorded traces, are compared to the response times of the operations in the simulation by investigating the differences between them. Because the response time is the metric that is the most simple to measure and the most relevant in the setting of this thesis, the evaluation will rely purely on the comparison of this one metric.

Table 6.1 gives on overview over the experiments conducted in the evaluation, explaining which research question they investigate and on which microservice system they are conducted.

6.3. Evaluation Setup

The first microservice application used in the evaluation is a system that has been generated using the model-based microservice architecture generation tool developed by Düllmann et al. [Arc18; DH17]. It is the same system that has been used as an example

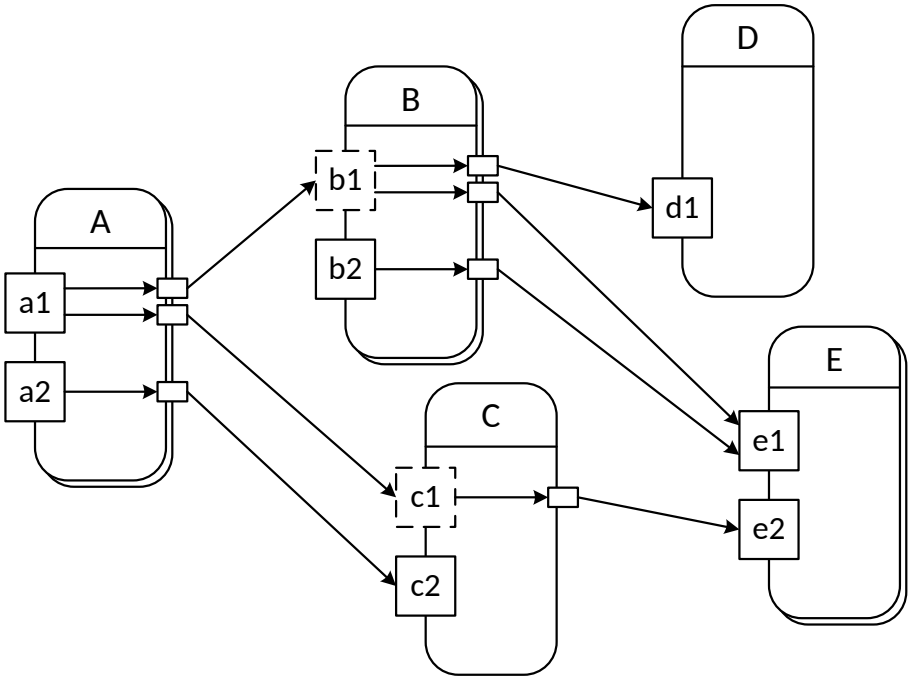


Figure 6.1.: Architecture of the generated microservice system that is used in the evaluation.

in other chapters of the thesis. The architecture of the generated microservice system is illustrated in Figure 6.1.

The system is made up of five services. The operations a1 and a2 of service A can be called by the user and have multiple dependencies to different operations from other services. In order to investigate RQ3 and RQ4.2, operation b1 of service B and c1 of service C implement a circuit breaker with the Hystrix library. This is indicated in Figure 6.1 by the dashed line around those operations. All services are created with the Spring Boot framework and implemented in Java and are run with docker-compose in Docker containers. The services have no real functionality besides calling their dependencies since they were only generated for the use in resilience benchmarking, but that should be sufficient for the conducted experiments. Small delays are incorporated in the operations to give the impression of an actual workload. This microservice system is used in the evaluation, because the model-based microservice architecture generator was created to allow for fast and simple generation of architectures based on the instances of a metamodel. While the resulting system is not a complex microservice application, it provides the building blocks for experimentation with different resilience and instrumentation mechanics. Furthermore, it provides a controlled environment for the experiments which will be conducted in this evaluation.

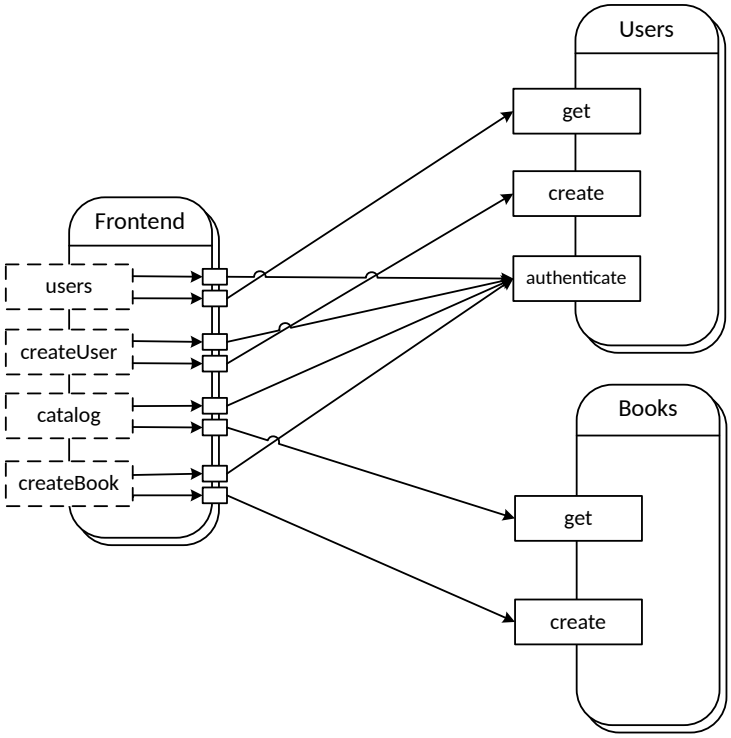


Figure 6.2.: Architecture of the library management system that is used in the evaluation.

The second microservice application used in the evaluation is a small library management system that is realized through three microservices. The architecture of the system is illustrated in Figure 6.2. The frontend service has four operations which all have a dependency to the authentication operation of the user service. In addition, each frontend operation also has another dependency to an operation from either the user service or the book service. Users can call the frontend operations to see the catalog of books; administrators can also create new book entries, see the list of registered users and create new users. In order to investigate RQ3 and RQ4.2, all frontend operations implement a circuit breaker with Hystrix. The user and book service are implemented in Node.js with the feathers.js[1] framework, while the frontend service was created with Spring Boot. This microservice system is used for the evaluation, because it is a real system that is used productively. Nevertheless, it is not critical to any business function and, therefore, provides a playground for experimentation. In addition to that, the system is has a simple structure and is easy to understand. Because of that it is not difficult to investigate the various research questions through experiments conducted with this application. Beyond this, it gives an opportunity to test the viability of the

[1] https://feathersjs.com/

developed architectural extraction approach on services that are implemented in another programming language than Java.

6.4. Experiment Settings

This section presents a description of the settings of the various experiments that are conducted in the course of this evaluation. To gain general, technology independent insights on the research questions stated in Section 1.3, experiments are conducted on two different exemplary microservice applications. Subsection 6.4.1 presents the experiments that are conducted in order to evaluate the accuracy of the developed architectural model extraction approach. Subsection 6.4.2 describes the experiments that investigate the accuracy of the microservice resilience simulator. Subsequently, Section 6.4.3 gives an overview of the experiments that evaluate the accuracy of the simulation of fault injections. Concluding, Section 6.4.4 presents the experiments which are conducted in order to evaluate the accuracy of the circuit breaker simulator by the simulator.

6.4.1. Accuracy of Architectural Model Extraction Approach

An experiment is conducted on each microservice application to investigate the correctness and completeness of the developed architectural model extraction approach.

Experiment 1.1

The goal of this experiment is to investigate RQ3 and to test the developed architectural model extraction approach for completeness and correctness. The generated microservice system is instrumented with the Jaeger tracing tool as described in Section 4.2. Operation b1 of service B and operation c1 of service C implement a circuit breaker with the Hystrix library in this setting. The system consists of two service instances of the services A, B, and E each and of one service instance of the services C and D each. That information should also be extracted into the architectural model. The two user-facing operations a1 and a2 of service A are both called three times each in order to generate traces as the basis for the extraction. Afterward, the developed extraction tool is executed to extract the architectural model from the recorded traces. After the model is created it is compared to the architectural model of the system that is presented in Section 6.4. In the comparison, the completeness and correctness of the services and their operations, the implementations of the circuit breaker pattern and the

dependencies between operations are checked in particular. The results are presented as a graphical visualization of the extracted architectural model similar to the illustration of the architectural model in Section 6.3.

Experiment 1.2

The goal of this experiment is to examine RQ3 and thus the correctness and completeness of the architectural extraction approach under different circumstances than in Experiment 1.1. During development, the approach was constantly tested on a microservice application whose services were implemented with Spring Boot. The purpose of this experiment is to investigate how well the architectural model extraction works for services that are implemented in a different programming language than Java.

The services of the library management system are again all instrumented with Jaeger as described in Section 4.2. All operations of the frontend service implement a circuit breaker using the Hystrix library. The frontend service runs in two instances, the users service in three and the books service only in one service instance. That information should be extracted into the architectural model. To create the traces that are the basis of the extraction process, each of the four frontend operations is called three times. Afterward, the developed extraction tool is executed to extract the actual architectural model from the collected traces. Once the the model is created, it is compared, as the model in Experiment 1.1, to the architectural model of the system that is presented in Section 6.3. Again, the completeness and correctness of the services and their operations, the implementations of the circuit breaker pattern and the dependencies between operations are checked in the comparison of the models. As in Experiment 1.1, the extracted model is presented through a graphical visualization similar to the illustrations in Section 6.3.

6.4.2. Accuracy of Microservice Resilience Simulator

An experiment is conducted on each microservice application to investigate the accuracy of the microservice resilience simulator.

Experiment 2.1

The goal of this experiment is to investigate RQ4, i.e., the accuracy of the simulation of a microservice system through the expanded simulator without any failure injections. For this, the system is executed and monitored with Jaeger to record the response times of its

Service	Number of Instances	CPU	Operations with Circuit Breaker
A	2	1 GHz	-
B	2	1 GHz	-
C	1	1 GHz	-
D	1	1 GHz	-
E	2	1 GHz	-

Table 6.2.: System configuration in Experiment 2.1.

operations. The services of the application run in Docker containers via docker-compose. The scale parameter of docker-compose is used to specify the number of instances of each service. There are two instances of the services A,B and E each and one instance of the services C and D each. The services' operations do not implement the circuit breaker pattern in this experiment setting, because it is supposed to represent normal productive execution without any anomalies. The *cpus* parameter provided by docker-compose is used to assign each service instance 0.357 CPUs of the four CPUs assigned to Docker. This should correspond to a CPU with the clock speed of roughly 1 GHz, since Docker has four CPUs with 2.8 GHz each assigned to it. The boost speed of the host machine's Intel Core i7 processor is not considered in the evaluation. The configuration of the system is also described in Table 6.2.

Apache JMeter[2], an open source load driver tool, is used to generate HTTP GET requests to the operations a1 and a2 of service A. 200 threads send one request to each operation over a ramp-up duration of 50 seconds. This results in four requests send to both operations per second over a duration of 50 seconds.

After the response times of the system's operations are recorded, the same experiment is simulated on the microservice resilience simulator. The system setting and experiment setting correspond to the experiment conducted on the real application. The simulator creates a report after the simulation, which contains, among other metrics, the response time of each simulated thread. The previously recorded and the simulated response times are compared at the end of this experiment to give an impression of the overall accuracy of the simulator in a simulation without anomaly injections into the system. The focus lies on comparing the changes of the response times over time and to check whether they correspond to each other.

[2]https://jmeter.apache.org

Service	Number of Instances	CPU	Operations with Circuit Breaker
Frontend	2	1 GHz	-
Users	3	2 GHz	-
Books	1	1 GHz	-

Table 6.3.: System configuration in Experiment 2.2.

Experiment 2.2

The goal of this experiment is to investigate RQ4, i.e., the accuracy of the simulation of a microservice system through the expanded simulator without any failure injections. The difference to Experiment 1.2 is that the experiment is conducted with a system that was not only generated for resilience experiments, but that is actually used productively. This leads to a higher workload than in Experiment 1.2, because the services do not solely call dependencies, but also have to make different computations.

The services of the library management system also run in Docker containers via docker-compose. In the setting for this experiment there are two instances of the frontend service, three instances of the user service and one instance of the book service. The user service has the most instances, because every operation of the frontend service is dependent on the authentication operation of the user service, which needs relatively much computational power compared with the other operations. In this setting, none of the operations implements the circuit breaker pattern. The *cpus* parameter provided by docker-compose is again used to limit the processor resources of every service instance. Each instance of the frontend and book service has 0.357 CPUs assigned to it. That corresponds to a CPU with the clock speed of approximately 1 GHz. Every instance of the user service has 0.714 CPUs assigned to it, was corresponds to a CPU with about 2 GHz. The configuration of the system is also described in Table 6.3.

As in the other experiments, Apache JMeter is used as the load driver in the experiment. It generates HTTP GET requests to the operations users, createUser, catalog and createBook of the frontend service. 30 threads send a request to the catalog, createBook and createUser operation each over a ramp-up period of 50 seconds. That amounts to one request sent to each of those operations every 1.67 seconds. At the same time, 20 threads send a request to the users operation over a ramp-up period of 20 seconds, what results in one request sent to the operation every 2.5 seconds.

The same experiment is simulated on the microservice resilience simulator and the recorded response times from the real execution and the simulation are compared to determine how accurate the simulation is.

```
{
    "service": "E",
    "instances": 1,
    "time": 10
}
```

Listing 6.1: Specification of a chaos monkey in the experiment model of the simulation.

6.4.3. Accuracy of Fault Injection Simulation

An experiment is conducted on each microservice application to investigate the accuracy and realism of the behavior of the microservice resilience simulator during fault injections into the simulated system.

Experiment 3.1

The goal of this experiment is to investigate RQ4.1, i.e., if the behavior of the simulator after fault injections is realistic and corresponds with the behavior of the real system. The experiment setting is the same as for Experiment 1.2, save for the failure injection. The injected failure is the shutdown of one instance of service E after 10 seconds in the experiment. Service E is chosen for the failure injection, because service E receives a high workload since multiple dependencies of the called operation a1 of service A are depending on it. The shutdown of one of service E's instances should cause an increase in the overall response times, because one service instance should be overwhelmed by the number of incoming requests.

The service instance is shutdown through a script that is executed from a thread that is started at the tenth second of the experiment by JMeter. The script shuts down one of the Docker containers that runs a service instance of service E. The response times of the systems operations are again recorded with Jaeger.

For the simulation, a chaos monkey is specified in the experiment model, as shown in Listing 6.1, that shuts one service instance of service E down after 10 seconds.

Again, the measured and simulated response times are compared. The main focus is to test, whether the failure injection leads to an increase of the response times in the simulation, as it does in the real execution of the system. Furthermore, the purpose of this experiment is also to probe if the increase of the response times after the failure

injection starts at the same time and happens at the same rate during the experiment in both the simulation and the real execution.

Experiment 3.2

The goal of this experiment is to investigate RQ4.1, i.e., if the behavior of the simulator after a fault injection corresponds to the behavior of the real system. Its purpose is once again to investigate the accuracy of the simulator when the simulated system is not only a generated application without actual workload.

The experiment setting is the same as for Experiment 2.2, except for the failure injection. The injected failure is again the shutdown of a service instance. After 10 seconds, one instance of the user service, which receives the heaviest load, is shut down. This should have an effect on the response time of every operation of the frontend service, since they all depend on the authentication operation of the user service.

As in Experiment 3.1, the service instance is shut down through a shell script that is executed by JMeter. A chaos monkey is specified in the simulation's experiment model again to achieve the same in the simulator.

The purpose of this experiment is also to examine if the increase of the response time of the operations after the failure injection in the simulation corresponds to that during real execution.

6.4.4. Accuracy of Circuit Breaker Simulation

An experiment is conducted on each microservice application to investigate the accuracy and realism of the circuit breaker implementation in the microservice resilience simulator.

Experiment 4.1

The goal of this experiment is to investigate RQ4.2, i.e., the behavior of the refined circuit breaker implementation in the simulator. Subject of the examination is especially the state transitions of the circuit breaker and the question if they occur corresponding to the implementation of the circuit breaker in the Hystrix library.

The experiment setting is for the most part unchanged from Experiment 3.1, except that two of the system's operations now implement a circuit breaker, as shown in Table 6.4. Operation b1 of service B and operation c1 of service C both depend on

Service	Number of Instances	CPU	Operations with Circuit Breaker
A	2	1 GHz	-
B	2	1 GHz	b1
C	1	1 GHz	c1
D	1	1 GHz	-
E	2	1 GHz	-

Table 6.4.: System configuration in Experiment 4.1.

Parameter	Value
rolling window	10000ms
request volume threshold	4
error threshold percentage	0.5
sleep window	5000ms
timeout	1000ms

Table 6.5.: Configuration of the circuit breakers implemented in operations b1 and c1.

operations of service E. When one instance of that service is shut down through a failure injection, the remaining service instance cannot handle the load of the incoming requests from operations b1 and c1. That should lead to increased response times of the two operations. Implementing a circuit breaker in those operations should lead to a timeout in that scenario, what means that the circuit breaker falls back to a standard response. If too many requests to the operations time out the circuit is opened and all requests are handled by the fallback mechanisms, i.e, keeping response times low. The circuit breakers of both operations are configured with the same parameters, which are described in Table 6.5.

The circuit breakers are also added to the architectural model that is used as input for the simulation in this experiment. The focus of the comparison of the measured and simulated response times is to see if the circuit breaker implementation in the simulator changes the circuits' states in the same manner and at the same time as Hystrix's circuit breakers. Also, to investigate if the simulator's circuit breaker has the same effect on the response times of the operations and the overall distributed transactions as the one in Hystrix.

Service	Number of Instances	CPU	Operations with Circuit Breaker
Frontend	2	1 GHz	users, createUser, catalog, createBook
Users	3	2 GHz	-
Books	1	1 GHz	-

Table 6.6.: System configuration in Experiment 4.2.

Parameter	users, catalog, createBook	createUser
rolling window	10000ms	10000ms
request volume threshold	3	3
error threshold percentage	0.5	0.5
sleep window	5000ms	5000ms
timeout	2000ms	4000ms

Table 6.7.: Configuration of the circuit breakers implemented in the operations users, catalog, createBook and createUser.

Experiment 4.2

The goal of this experiment is to investigate RQ4.2, i.e., the behavior of the refined circuit breaker implementation in the simulator. Especially, if the circuit state transitions also correspond to those of the real circuit breaker in a system that has an actual workload.

Save for the added circuit breakers the experiment settings remains unchanged from Experiment 3.2. Table 6.6 shows the system configuration used in this experiment, including which operations implement the circuit breaker pattern. The configuration of the different circuit breakers can be seen in Table 6.7.

The focus of the comparison of the measured and simulated response times is to see if the circuit breaker implementation in the simulator changes the circuits' states at the same time as Hystrix's circuit breakers, in this system, too.

6.4.5. Evaluation Environment

All experiments are conducted in a Docker (v18.06.0-ce-mac70) environment that got assigned four CPUs and 8196MB of RAM in total. The system on which Docker is running is a MacBook Pro 15" 2015 model with a quad-core 2.8 GHz Intel Core i7 processor with 16GB of RAM. The machine is running macOS High Sierra v10.13.3. The Java services of the microservice systems are instrumented with the Jaeger java client library v0.27.0. The Node.js services of the library management system are instrumented with

the Jaeger node client library v3.11.0. Spring Cloud Starter Hystrix v1.4.5 is used for the implementation of the circuit breaker pattern. The load driver used in the experiments is Apache JMeter 4.0.

6.5. Description of Results

Subsection 6.5.1 describes the results of the experiments made to investigate RQ3. Subsequently, Subsection 6.5.2 presents the results of the experiments made to investigate RQ4. Subsection 6.5.3 details the results of the experiments conducted in order to investigate RQ4.1 and Subsection 6.5.4 describes the results of the experiments carried out to investigate RQ4.2. The dataset containing all metrics that have been recorded in the course of the evaluation is publicly available at [Bec18].

6.5.1. Accuracy of Architectural Model Extraction Approach

Figure 6.3 shows a graphical visualization of the architectural model that was extracted from the generated microservice system during Experiment 1.1, using the approach presented in Chapter 4.

The model contains all five services of the application and the correct number of service instances for each service. All operations have been extracted, save for operation b2 of service B. Except for the one dependency of that operation, all dependencies between operations have been extracted correctly, too. The dashed lines around operation b1 of service B and operation of service C indicate that the operations implement a circuit breaker. So the information about the implemented patterns has been extracted correctly, as well.

Figure 6.4 shows the graphical visualization of the extracted architectural model of the library management system that was obtained in Experiment 1.2.

This model also contains all services of the application and the correct number of service instances for each service. While all operations of the frontend service have been extracted, the operations that the model extracted for the users and the books service deviate from their operations that are specified in the architectural model presented in Section 6.3. Instead of the operations get and create, the books service is represented with a single operation, named books. The dependencies of the operations catalog and createBook of the frontend service point to those two operations point to the books operation in the extracted architectural model. While the users service is represented with the authentication operation, its get and the create operations are also replaced by

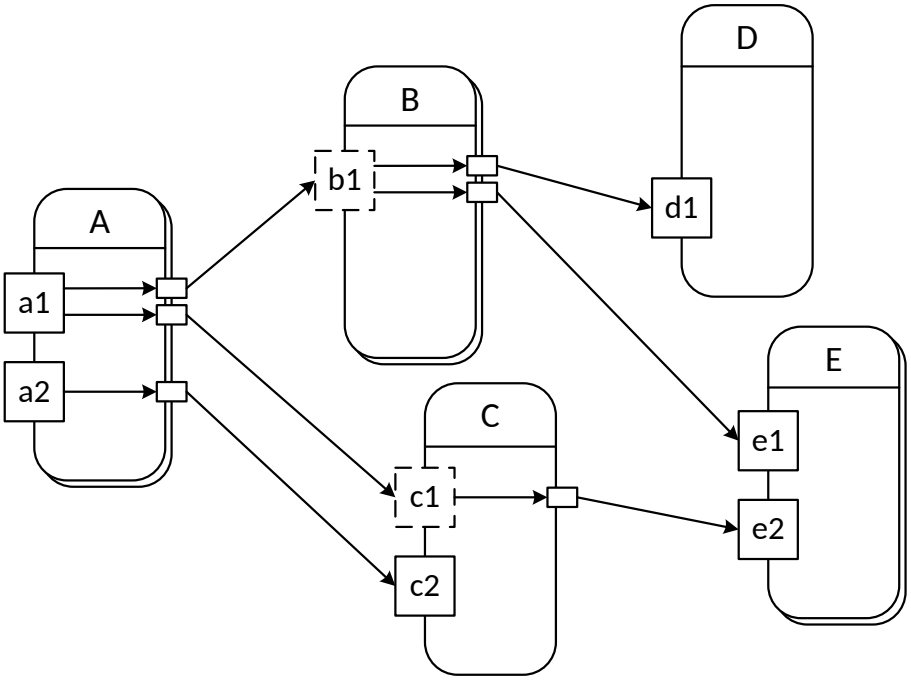


Figure 6.3.: Extracted architectural model of the generated microservice system visualized as a graph.

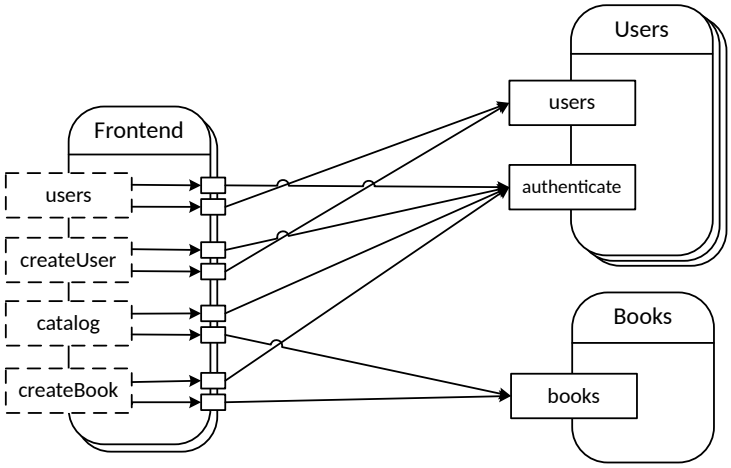


Figure 6.4.: Extracted architectural model of the library management system visualized as a graph.

one operation, called users. The former dependencies to get and create now point to the users operation. The circuit breakers of the frontend operations have been correctly extracted into the architectural model.

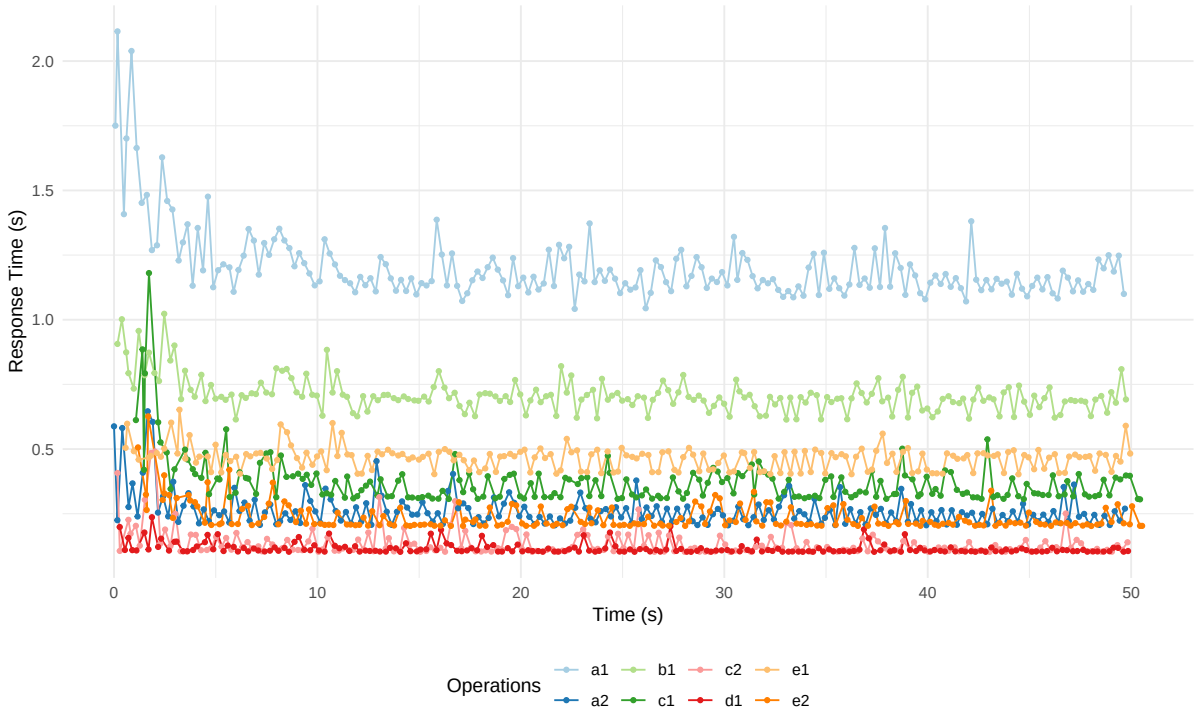
6.5.2. Accuracy of Microservice Resilience Simulator

The response times that have been measured and simulated during Experiment 2.1 are visualized in Figure 6.5. Figure 6.5a shows the response time of each operation in seconds that has been measured on the real system during the experiment over a duration of 50 seconds. The microservice resilience simulator does not display the response time of each operation in its report. Instead, it displays the response time in seconds of each thread of every microservice instance. This is visualized in Figure 6.5b.

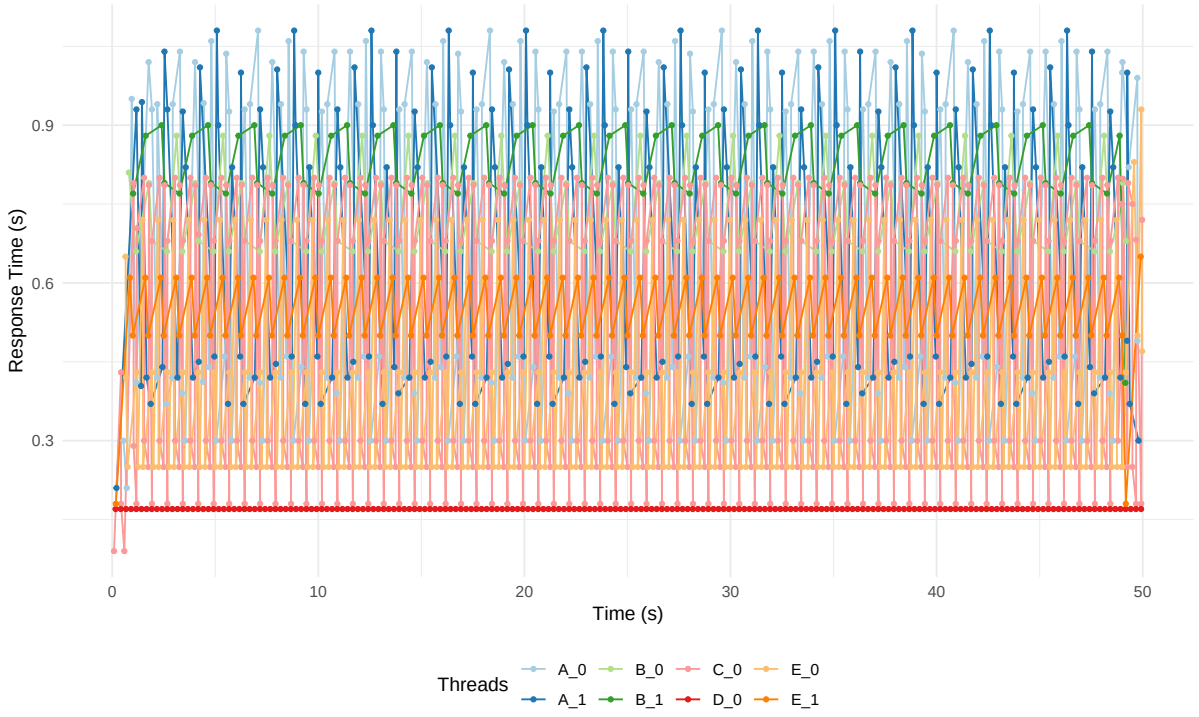
As can be seen in Figure 6.5a, all operations of the real system experience a spike in their response times during the first seconds. This is most likely due to some initialization procedures of the services. The response times decline until approximately the third second from where on they stay on the same level for the rest of the experiment with some minor variations. For example, the response time of operation a1 varies between 1.1 and 1.4 seconds in this period, that of operation a2 corresponds to 0.2 to 0.4 seconds. The different presentation of the response time by the simulator makes it a bit difficult to compare its values with the response times of the real operations, but a closer look on Figure 6.5b allows for some differentiation of the values of different service instances. The response times of the threads executed on the instances from service A can be grouped into two areas. Some threads have a response time between 0.8 to 1.1 seconds while others take on response times between 0.3 to 0.5 seconds. Those two areas likely correspond to the different response times of operations a1 and operation a2. In comparison with the measured response times of all operations from the real execution of the system the simulated response times are consistently off by a few hundred milliseconds. Since not all thread response times of Figure 6.5b allow a conclusion on the operation response times, we refrain from a precise comparison of the gathered values of the real execution and the simulation of the system. Instead, we will take a look at the development of the response times over time.

While the values of the real system have some slight, irregular variation, the variation of the values from the simulation is recurring and builds a pattern. The response times of both the real system and the simulation stay consistent throughout the whole experiment duration and experience neither a drastic increase or decrease. Even though the simulated response times are not completely accurate, they correspond to the development of the real response times in that matter.

The findings from Experiment 2.2 match those from Experiment 2.1 despite the different workload in the library management system. The response times reported by the simulator in Experiment 2.2 also do not accurately match those recorded in the real system, but the evolution of the response times in the simulator does correspond to that of the real response times in this experiment as well. The related plots can be found in Appendix A



(a) Measured response times.



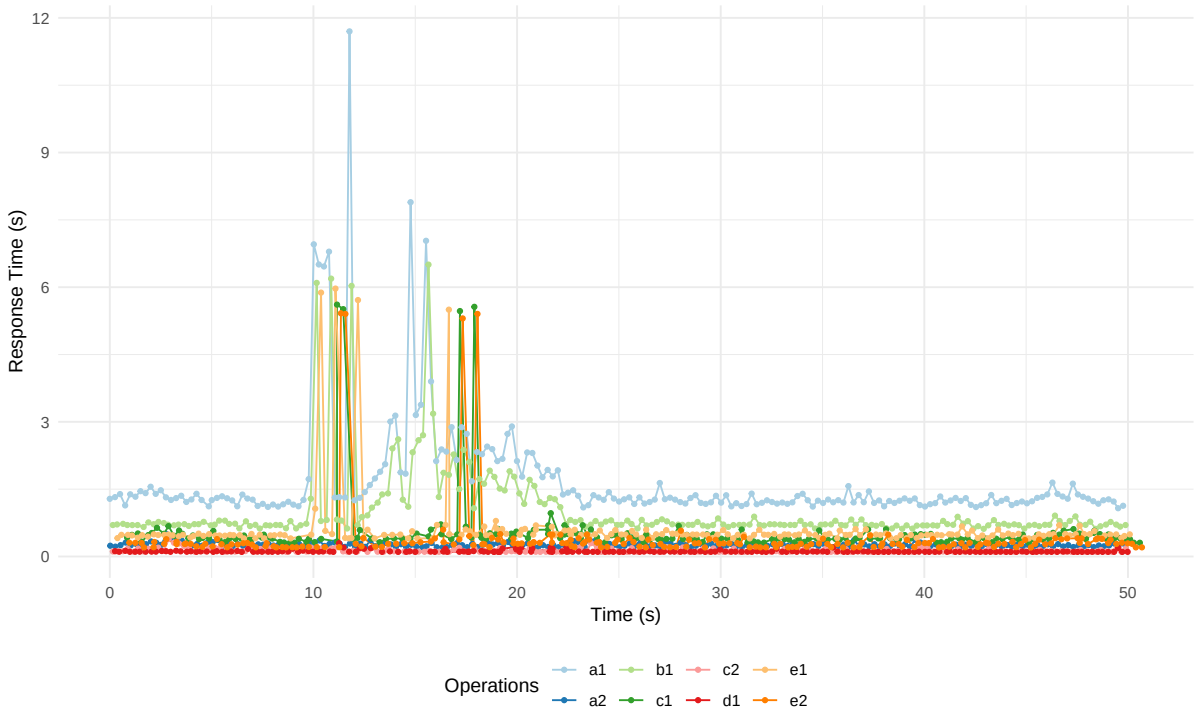
(b) Simulated response times.

Figure 6.5.: Response times recorded during Experiment 2.1.

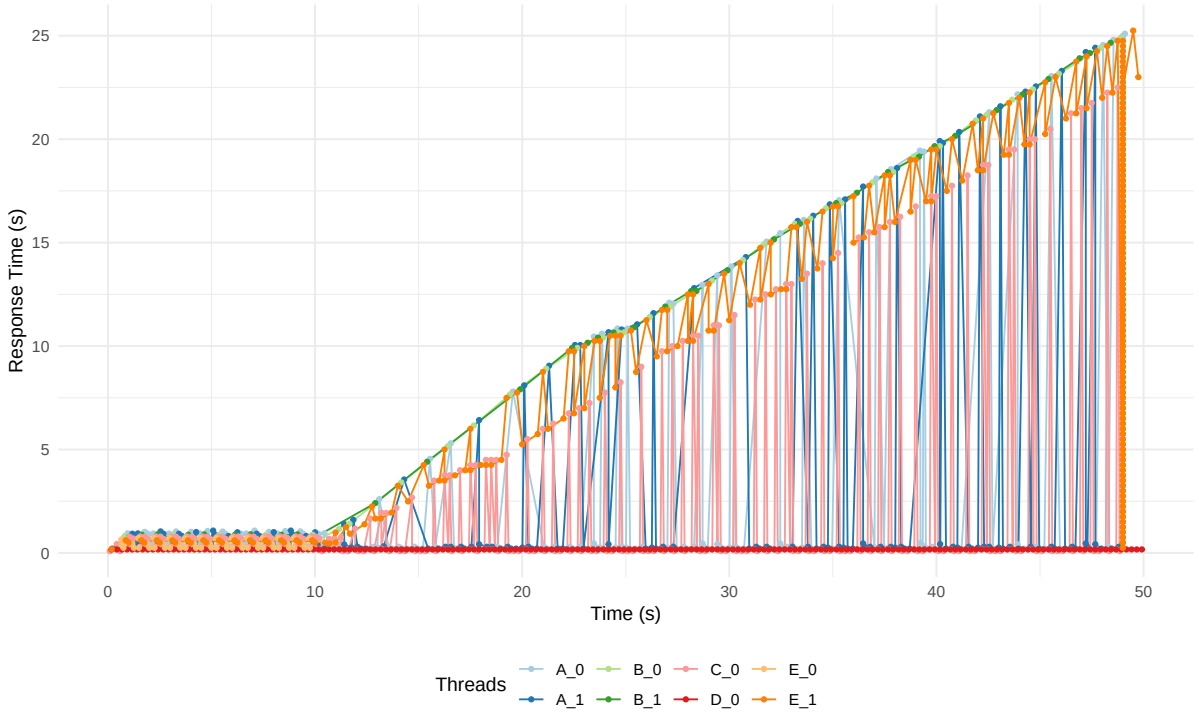
6.5.3. Accuracy of Fault Injection Simulation

Figure 6.6 presents the response times that have been measured and simulated during Experiment 3.1. After ten seconds, one instance of service E was shut down in this experiment. It is visible in Figure 6.6a that the response times of the two operations of service E increases severely as a consequence. This propagates to the other operations that depend on service E. It can be observed that the response times of operations c1, b1 and a1 soars as well. Astonishingly, the response times fall again around second 13 before they rise again a few seconds later. Around 22 seconds into the experiment and twelve seconds after the service instance was shut down, the response times of the aforementioned operations go back to the values before the failure. The expected behavior would have been for the response times to keep rising, because the single instance of service E should not have been able to handle the load of incoming requests by itself. We have not found a satisfactory explanation as to why the operation times decrease again after some seconds; the same findings were recreated when the same experiment was conducted again.

As shown in Figure 6.6b, the simulated response times of operation e1, e2, c1, b1 and a1 rise as well after one instance of service E was shut down by a simulated chaos monkey. Before that, the response times in the simulation again correspond approximately to those of the real system. The increase of the response times after the service instance shutdown during the first seconds is slower than it is on the real system, but it is still distinct. Unlike in the real execution, the response times of the mentioned operations keep rising after the failure. While this is the expected behavior, it does not correspond to the unusual behavior of the real system. Hence, no meaningful findings on the accuracy of the simulation of fault injections trough the microservice resilience simulator can be made in this experiment.



(a) Measured response times.



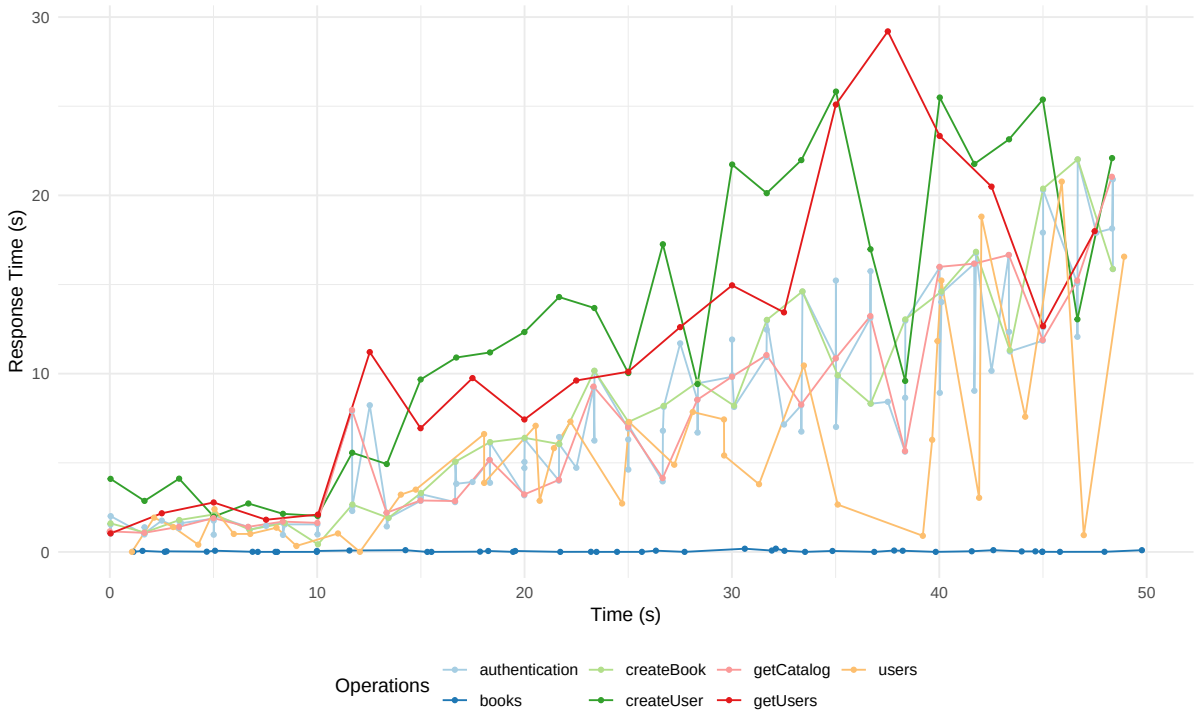
(b) Simulated response times.

Figure 6.6.: Response times recorded during Experiment 3.1.

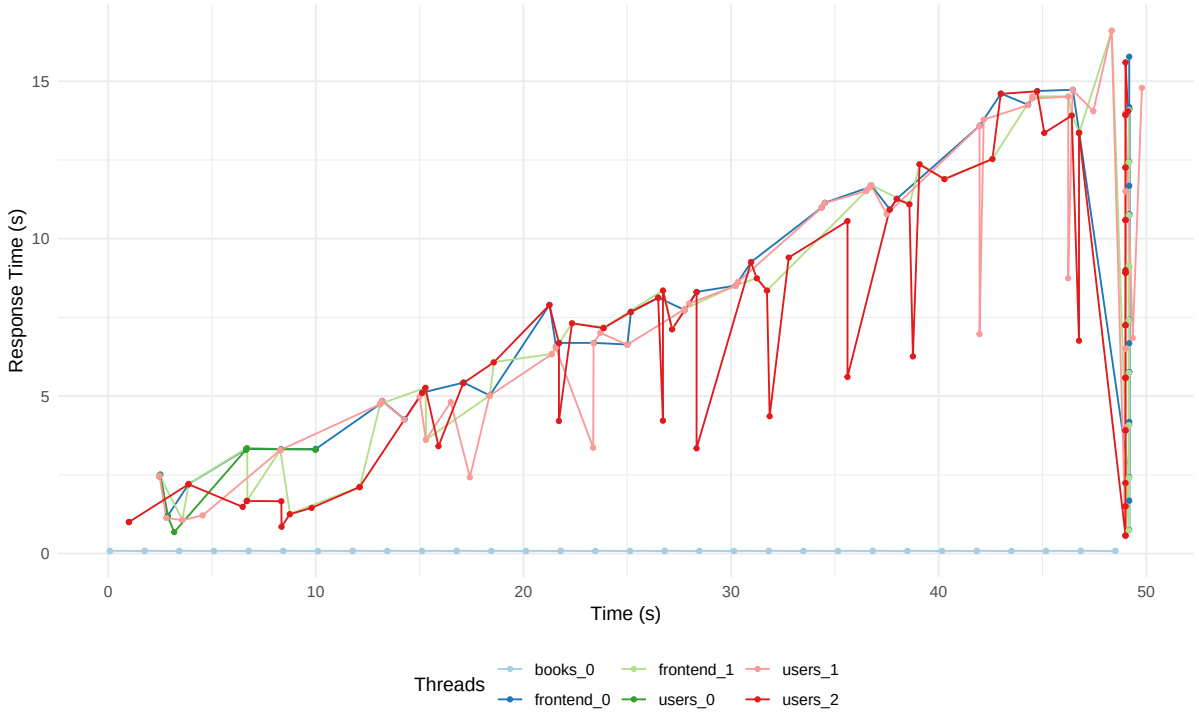
The response times that have been measured and simulated during Experiment 3.2 can be seen in Figure 6.7a. Again after ten seconds, a service instance was shut down. This time an instance of the users service, which has many dependencies from the frontend service on it. The response times of the getUsers and createUser operations of the frontend service experience a severe increase right after the failure. The response time of the getUsers operation even soars higher than 10 seconds. In the simulation, the rise of the corresponding response times is slower. Shortly after the failure, the affected operations' response time is still around five seconds.

The response time of the books operation of the books service is unaffected by the service instance shutdown since it has no dependencies to the users service. This is accurately simulated in the simulator.

Wile the affected response times rise in both the real system and the simulation, the increase of the real response times is steeper than in the simulation. 30 seconds into the experiment, the real response time of the createUser operation is about 22 seconds and the one of the getUsers operation around 15 seconds. The response time of the authentication operation is around twelve seconds at this point. In the simulation, the response time of the frontend threads is around eight seconds and the one of the users service's threads is around the same value. The values for the users operation of the users service correspond to each other since its real response time is also round five to eight seconds. But in total, this means that the discrepancy between the simulation's response times and the real response times is roughly seven to 14 seconds. While the simulated response times do not accurately reflect the measured response times of the system's operations, an increase can be seen in both recordings. The increase does not happen at the same rate, but the simulation reflects the general behavior of the real system after the service instance shutdown.



(a) Measured response times.



(b) Simulated response times.

Figure 6.7.: Response times recorded during Experiment 3.2.

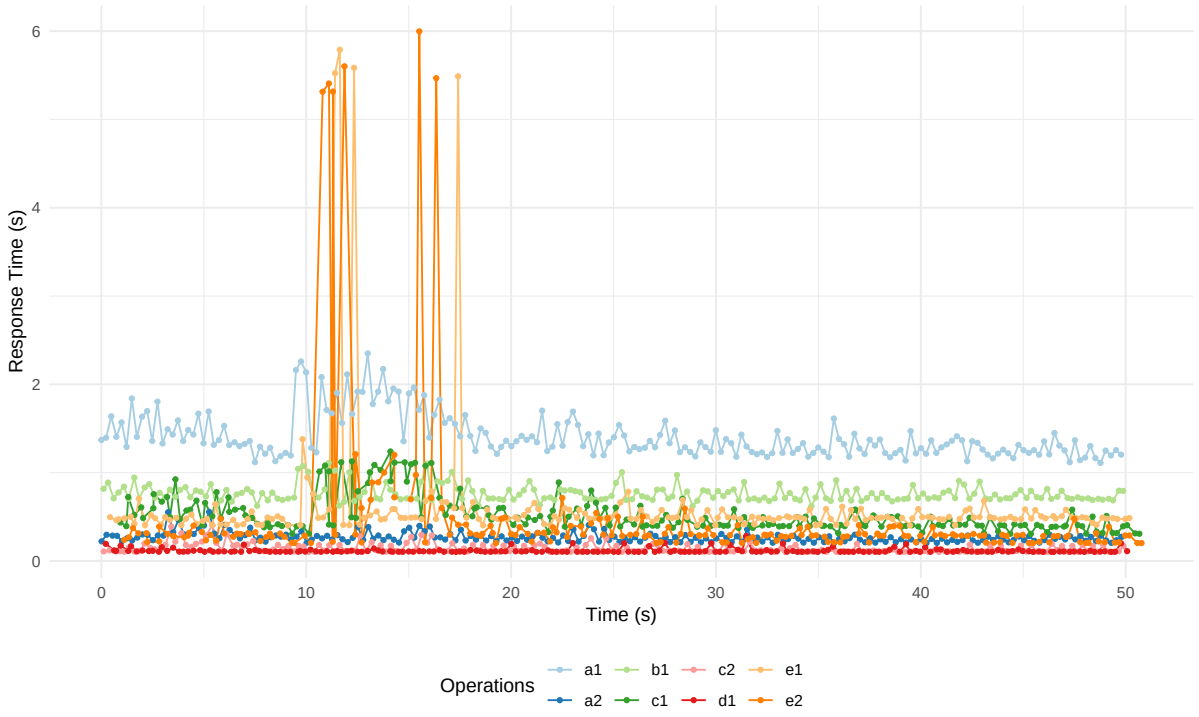
6.5.4. Accuracy of Circuit Breaker Simulation

In Experiment 4.1, a circuit breaker is implemented in operations b1 and c1 to keep the response time of operation a1 at an acceptable level. The configurations of the circuit breakers and the experiment setting are described in Section 6.4.4. Figure 6.8 visualizes the measured and simulated response times during the experiment.

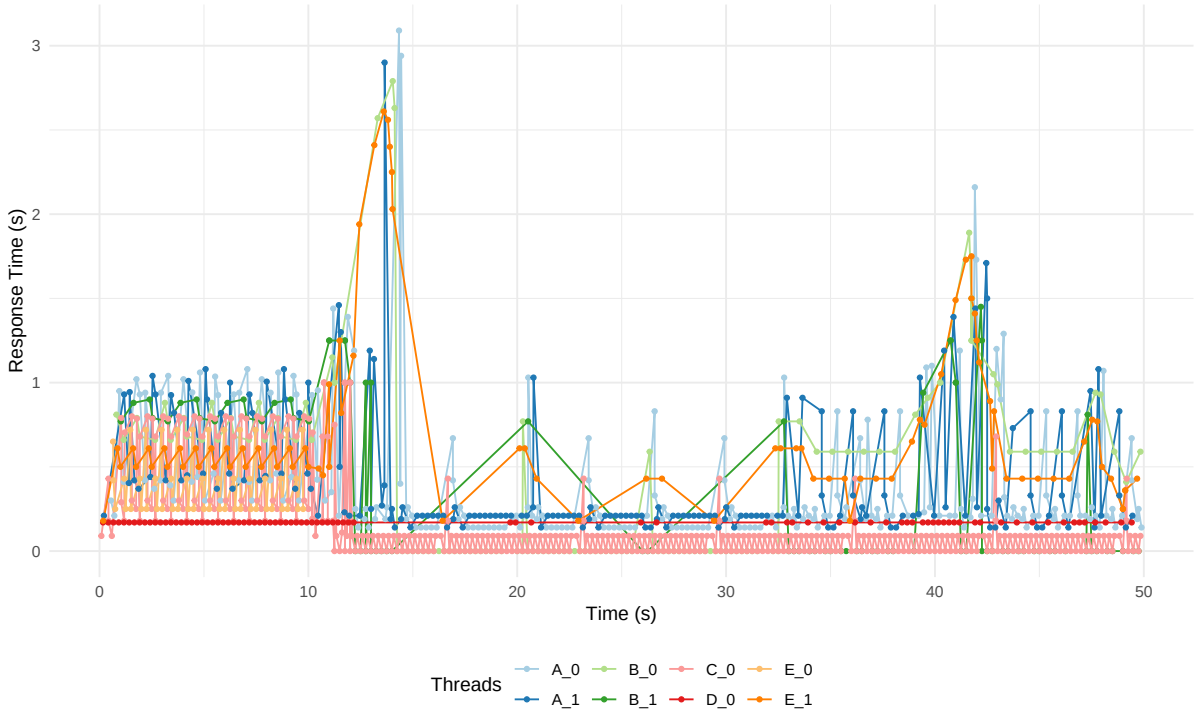
Figure 6.8a shows that the response time of operation e1 and e2 rises drastically to almost 6 seconds after the failure of the microservice instance. In contrast to Experiment 3.1, where this caused an increase in response time for operations c1, b1 and a1 as well, the response times of those operations only experience an increase of a few hundred milliseconds. This is caused by the circuit breaker implemented in operation b1 and c1 which executes the specified fallback mechanism of the operations after the timeout of one second. As observed in Experiment 3.1, the response time of the affected operations goes back to normal again some seconds after the service instance shutdown, but the effect of the circuit breaker could be observed during the few seconds in which the response times spiked.

The response times in the simulation are presented in Figure 6.8b. As in the real system, a spike in the response time of service E's operations can be seen directly after the service instance is shut down. However, some of the threads of service A and service B also have a response time of around three seconds, which should not be the case because of the circuit breaker in operation b1 and c1. It can be observed, that the circuits of those circuit breakers change to the open state at around the time point of 15 seconds because no more requests are sent to service E in the next five seconds and the response times of the threads of all services are low. At around 20 seconds, the circuits change into the half-open state and a trial request is sent. This trail seems to fail since the response time of most threads remains low and only a few requests are sent to service E. At 33 seconds, instance 0 of service B opens its circuit again. This can be observed based on the increase in response times of services A, B, and E. The remaining instance of service E seems to be able to handle the requests sent by only one instance of service B. As soon as the second instance of service B starts to send requests again around second 39, the response times go up again until that instance of service B opens its circuit in operation b1 again.

In conclusion, the behavior of the simulated circuit breaker is not very accurate, as some threads still experience high response times even though their requests should experience a timeout. Later, the simulated circuit breaker stops a cascading of the failure of a service instance of service E to other services, by opening its circuit. But the findings of the experiment are inconclusive due to the inexplicable decrease of the response times in the real system. Because of that, no full comparison between the real and the simulated circuit breaker could be undergone.



(a) Measured response times.



(b) Simulated response times.

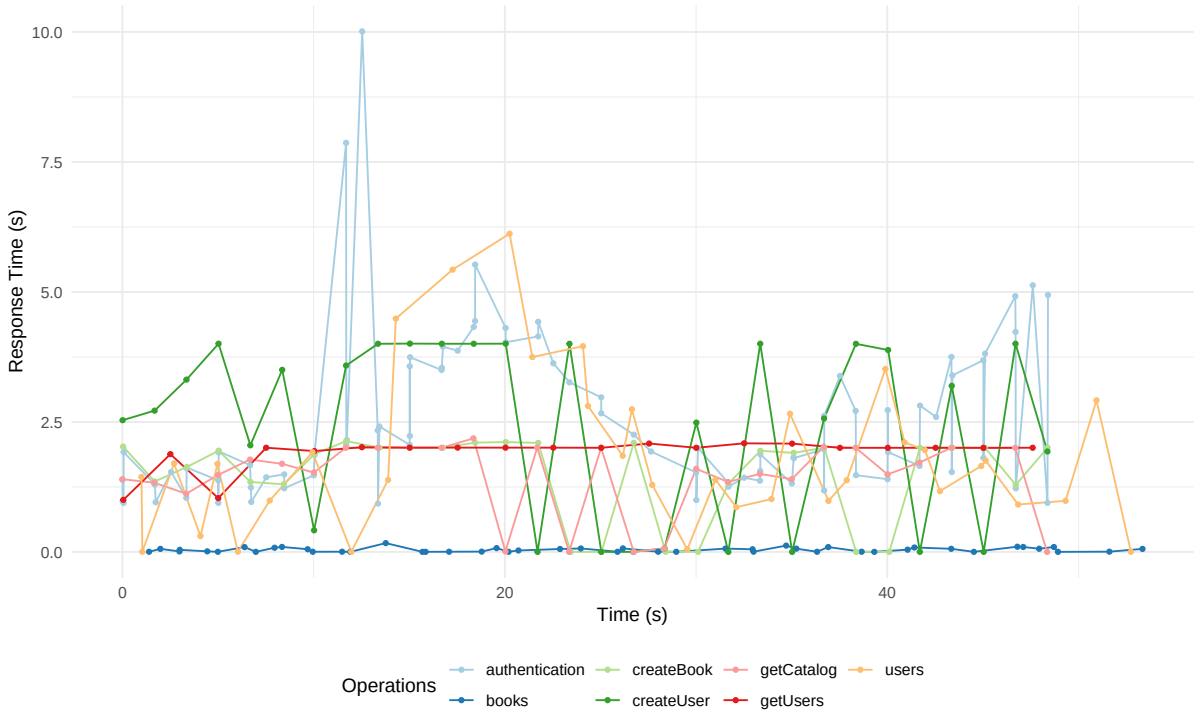
Figure 6.8.: Response times recorded during Experiment 4.1.

Figure 6.9 visualizes the measured and simulated response times during Experiment 4.2. The setting of the fault injection and the circuit breaker configuration is described in Section 6.4.4

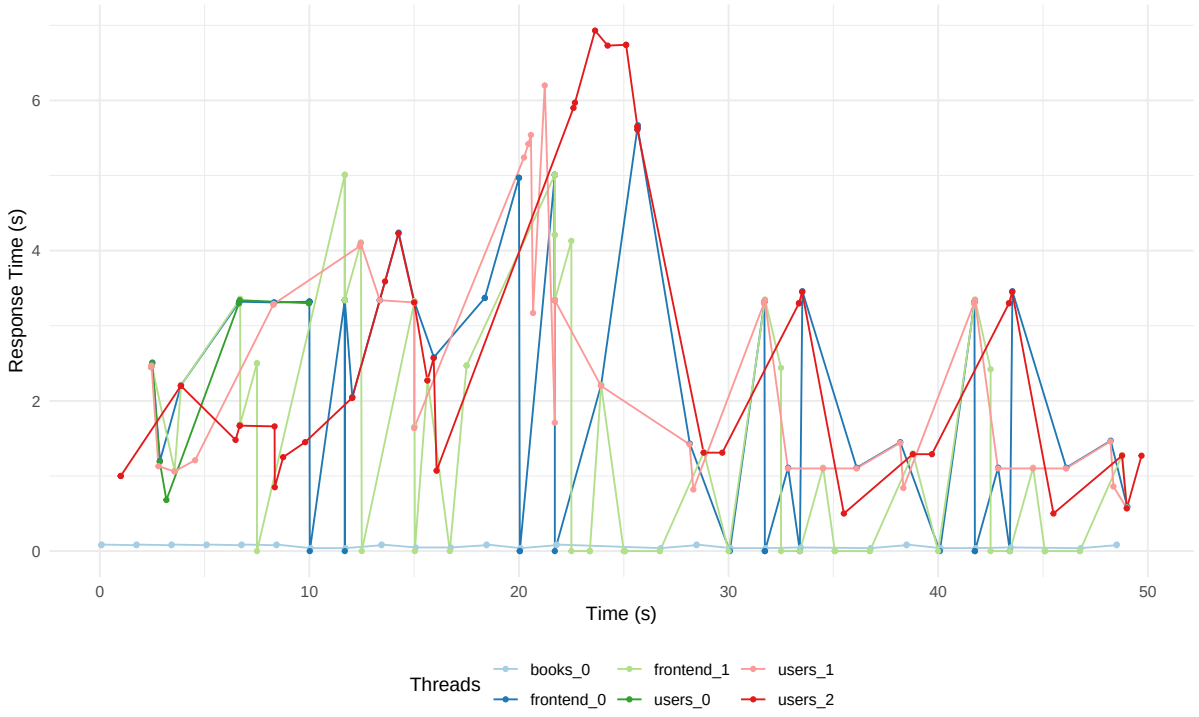
Figure 6.9a shows that the authentication operation of the users service experiences a drastic increase in its response time of up to ten seconds. The response time of the users operation also rises to approximately six seconds over the course of the next ten seconds. The response time of the createUser operation does not pass the four second mark, as per specified timeout of its circuit breaker. After several timed out requests, one of the two frontend instances opens the circuit of the createUser operation at around 20 seconds. This is visible by the response times of the operation jumping between almost zero seconds and four seconds. It can be observed that the response times of the authentication and users operations sink while the circuit is opened. The getUsers operation times out during every request as well and can be seen in the figure with a constant response time of two seconds. This is because even obtaining the fallback response of the operation takes that amount of time. The createBook and getCatalog operations of the frontend open their circuit on once service instance at respectively 19 and 22 seconds, as the createUser operation does. This can also be deducted by their jumping response times.

Figure 6.9b shows the response times of the service threads in the simulation. After the service instance of the users service is shut down at ten seconds, the users threads do not experience a rise in their response times as drastic as the authentication operation does in the real system. Their response times are about four to five seconds. Over the next seconds those response times rise as in the real system to around six to seven seconds, although that happens a bit slower than it does in the real system. It can again be noted that some of the frontend threads have a higher response time than the two or respectively four seconds of the timeout specified in the circuit breakers. While the circuits of the frontend operations open at around 20 seconds, the same happens in the simulation at about 25 seconds in the experiment. This can be seen by the decline of the response times of the frontend operations as well as of the operations of the users service. Some threads have a response time of only a bit more than zero seconds while others have response time of around one or three seconds, depending on the operation they execute. This indicates again that only the circuits of one frontend instance have been opened.

Overall, while the accuracy of the simulated response times is not spot on, the behavior of the simulated circuit breaker corresponds to that of the real circuit breaker. The response times may not be exactly the same and circuit state transitions happen delayed from the real circuit breaker, but they offer a sufficient perception of how the circuit breaker acts during real system execution.



(a) Measured response times.



(b) Simulated response times.

Figure 6.9.: Response times recorded during Experiment 4.2.

6.6. Discussion of Results

This section provides a discussion of the previously described findings of the conducted experiments.

6.6.1. Accuracy of Architectural Model Extraction Approach

Based on the results of Experiment 1.1 and Experiment 1.2, the developed architectural model extraction approach works well for services implemented with Spring Boot. Except for once operation, the architecture of the system was completely and correctly extracted. Since collected traces are the basis of the extraction approach, it is not possible to extract information about operations that were not executed and, therefore, not included in the traces. It could be difficult to call every operation in a big and complex microservice application with possibly hundreds of services. Thus, this case exposes a not insignificant weakness of the developed approach. Since it is build on the basis of dynamic analysis this weakness cannot be revised — to extract an operation it must be called first during the preparation phase for the extraction.

Furthermore, Experiment 1.2 uncovered another weakness of the approach. Due to the way that feathers.js works [Fea18], CRUD operations (create, read, update, delete) are always handled by one single operation. This might be the case in other frameworks, too. The only way to differentiate which operation is called in the background is by examining the HTTP method of the request. This could be possible future work in order to improve the approach when used to extract architectural models from services implemented with feathers.js.

The developed approach could also be extended to extract more of the architectural information that is needed by the microservice resilience simulator, like the resources available to a service instance or the resource demand of an operation.

6.6.2. Accuracy of Microservice Resilience Simulator

Based on the results of the experiments that investigated RQ4, RQ4.1 and RQ4.2, the general accuracy of the simulator is rather bad, as simulated response times deviate from real response times by several hundreds of milliseconds to up to multiple seconds. These differences are bigger and more noticeable after fault injections, when the simulated response times rise at a different rate than their real counterparts.

Due to the simulator reporting response times for finished threads instead for operations, it was difficult to precisely compare the response times of the real system to

those of the simulated system. When it was possible to identify the response times of individual operations in the thread response times, the aforementioned findings can be concluded.

The evaluation was made more difficult by the unexpected development of the measured response times in Experiment 3.1 and Experiment 4.1. For an unknown reason, the response times of the operations affected by the fault injection settled down again after some seconds, what is not the previously expected behavior. The fact that all of the services of the system used in the experiments were running on one machine made it difficult to limit the amount of resources available to the Docker containers. An unexpected resource distribution could have led to more resources than specified being allocated to the remaining service instance of service E and, thus, be responsible for the inconclusive findings of these experiments.

It was shown that the simulated circuit breaker does not consistently handle requests, which time out while the circuit is closed, with the specified fallback mechanism, what leads to an unrealistic high response time of some threads. This behavior does not correspond to the implementation of the circuit breaker in Hystrix.

These findings indicate poor accuracy of the simulations, what is not surprising, as the simulator and the architectural model disregard some aspects of real microservice environments, such as network latency, and the estimation of the operations resource demands is difficult and imprecise. All of these aspects present opportunities for future work to improve the simulation accuracy. Nevertheless, while the response times were not exactly precise, they never contradicted the behavior and the development of response times in the real systems. During the first two experiments without fault injections, response times stayed at the same level, as they should have. If the inexplicable findings from Experiment 3.1 and Experiment 4.1 are disregarded, the effects of fault injections were clearly observable in a distinct increase in response times. While this increase did not happen at the same rate or to the same extend as in the real system, it was still distinct and illustrated the existence of an antipattern in the architecture, namely slow responses and cascading failures. And even though the simulation of the circuit breaker pattern did not exactly mirror the circuit breaker of the Hystrix library, because time outs did not lead to fallbacks in every case and circuit state changes happened slightly delayed from the real system, it still demonstrated the general effects that the implementation of such a resilience pattern has on change of response times after a failure. The findings of Experiment 4.2 show that the simulated circuit breaker went through the same state transitions as the real one. In addition, when the circuit of an operation opened only on one service instance, the simulator reflected that behavior correctly in the simulation.

In conclusion, the simulator does not offer the best accuracy during the simulation of microservice architectures. But its behavior corresponds to that of the real respective

system. This holds true for the behavior of a system after fault injections and for the behavior of the simulated circuit breaker pattern. Thereby, the simulator is not able to precisely predict the metrics of system execution through simulations. But that is also not the focus of the approach. The focus lies on the evaluation of resilience antipatterns in microservice architectures and the findings of the evaluation indicate that the simulator fulfills these requirements. The simulator can be used to efficiently assess the potency of fault injections and, therefore, achieves its goals.

6.6.3. Threats to Validity

As this evaluation was conducted in a lab experiment, there are several threats to validity that are discussed hereafter.

Internal

Both microservice applications used in the evaluation have a rather simple structure and are small in size compared with real microservice systems. Due to those limitations, the structure of the systems might not be representative of a real microservice system. All of the system's services were executed on the same machine, what does not represent a real microservice environment, which usually runs on a cloud platform. Furthermore, the estimation of the operations' resource demands in the architectural model is not precise. This could lead to inaccuracies of the presented results. Since the Docker containers, which ran the microservices, were all executed on a real machine, it was a difficult task to limit their access to the hosts computing resources. The attempt made through the docker-compose configuration file does not guarantee precise resource limitations. This could be observed in Experiment 3.1 and Experiment 4.1. The microservices of the generated microservice system did not have a real workload, therefore, they might not be representative to the behavior of real microservices.

External

In terms of the applicability of the results of this evaluation on the real world, the evaluation could only represent parts of real microservice environments. The scale, complexity, and structure of the used microservice applications is not representative of actual microservice systems. The execution of the services in a Docker environment is also not representative to the execution of real microservice systems on multiple hosts or in the cloud. Since the generated load and the injected faults were chosen because they demonstrate the effect of successful fault injections and the circuit breaker pattern

quite clearly, they might not be representative to general fault injections or the effects that the circuit breaker pattern has on a system.

Chapter 7

Conclusion

In the conclusion, the thesis is summarized and the overall outcome is discussed. Additionally, possible future work is presented.

7.1. Summary

This thesis proposes an approach for effective simulation-based evaluation of resilience antipatterns in microservice architectures in the context of the Orcas project. The implementation of the circuit breaker pattern in the microservice resilience simulator was refined to correspond to the implementation of the circuit breaker of the popular Hystrix library. In connection to that, the input model of the simulator was split into two separate models. An architectural model that contains information about the system that is simulated and an experiment model that contains meta information about the experiment to be simulated on the system. Parameters have been added to the model of the circuit breaker to give a realistic representation of the configuration of Hystrix's circuit breaker. Furthermore, an architectural model extraction approach was developed, which is based on service instrumentation via Jaeger and OpenTracing. The traces that are collected during system execution are used to extract architectural information into a model. That information compromises the microservices of the systems, the number of their service instances, and their operations. Furthermore, it extracts the information if an operation implements the circuit breaker pattern and the dependencies between services and their operations. Concluding, the work of this thesis was connected with the Orcas Decision Engine to enable the efficient assessment of the potency of fault injections recommended by the decision engine.

The supplemental material for this thesis is publicly available at [Bec18].

7.2. Discussion

The results of the evaluation showed that the developed architectural model extraction approach is promising, although it highlighted some of its weaknesses as well. Because the extraction is based on the collected Jaeger traces, operations must be executed in order to be represented in the architectural model. In vast and complex applications this might prove difficult. Operations that are not executed do not appear in the extracted model. Another finding of the evaluation was that the approach does not take the used HTTP method of requests into account. This can lead to incomplete representation of a service's operations if they are only differentiated by the used HTTP method, but not by the called operation.

The findings of the evaluation indicated that the accuracy of the microservice resilience simulator is rather imprecise. Differences in response times off multiple hudreds of milliseconds to seconds were revealed between the simulation and the real system execution. But the results of the evaluation also showed that the overall behavior of the simulator represents real microservice systems sufficiently. Fault injections induce the real increase of response times that corresponds vaguely to that measured in the real system. The simulated behavior of the circuit breaker pattern, while not being completely accurate, matches the behavior of the real circuit breaker implementation in Hystrix. Both the simulated and the real circuit breaker traversed the same state transitions in the evaluation. Thereby, the developed approach can be used to evaluate resilience antipatterns in microservice architectures and to efficiently asses the effectiveness of fault injections recommended by the Orcas Decision Engine, in order to select faults for the injection in the real system.

7.3. Future Work

This section presents potential future work that could be pursued to improve the developed approach.

7.3.1. Microservice Resilience Simulator

Many potential aspects came up during the thesis that could be pursued in future work in order to improve the microservice resilience simulator. The simulator's accuracy could be improved by including more detailed properties for the modeling of a service's available resources or the resource demand of an operation. Latency could be added to the simulation to make it more realistic. Furthermore, automatic scaling of service

instances could be integrated into the simulator to give a more realistic environment that corresponds more to the cloud environments in which microservice systems are hosted. Service instances that have been shut down by a simulated chaos monkey could recover after some time, to represent a system with self-healing services. To add to this, other failure modes besides the failure of en entire service instance could be added to the simulator. One example for such a failure mode could be latency that is injected into the connection between two services.

7.3.2. Architectural Model Extraction Approach

The focus of future work could be put on extending the attributes that the developed approach extracts from microservice applications. Such attributes could be the resource configuration of a service instance or the automated recognition and extraction of the resource demand of an operation. At the moment, this information must be manually added to an extracted model before it can be used as input for the simulator. In addition to that, static code analysis could be included in the approach to be able to also represent operations in the architectural model that were not recorded in collected Jaeger traces. Such an extension of the approach would result in multiple new challenges and almost represent the development of a second architectural model extraction approach. Another focus of the developed approach could be put on analyzing the HTTP methods of requests to an operation to offer a more accurate extraction of operations from services implemented with feathers.js or similar frameworks.

Chapter 7

Bibliography

[Ale77] C. Alexander. *A pattern language: towns, buildings, construction*. Oxford university press, 1977 (cit. on p. 10).

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing." In: *IEEE transactions on dependable and secure computing* 1.1 (2004), pp. 11–33 (cit. on pp. 7–9).

[Arc18] *Model-driven Generation of Microservice Architectures for Performance and Resilience Benchmarking*. URL: https://github.com/orcas-elite/arch-gen (visited on 08/07/2018) (cit. on p. 67).

[BBM13] M. Becker, S. Becker, J. Meyer. "SimuLizar: Design-Time Modeling and Performance Analysis of Self-Adaptive Systems." In: *Software Engineering* 213 (2013), pp. 71–84 (cit. on p. 28).

[BBR+16] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, C. Rosenthal. "Chaos Engineering." In: *IEEE Software* 33.3 (2016), pp. 35–41 (cit. on pp. 1, 19, 20, 55).

[Bec18] S. Beck. *Simulation-based Evaluation of Resilience Antipatterns in Microservice Architectures*. 2018. URL: https://doi.org/10.5281/zenodo.1341684 (cit. on pp. 4, 78, 95).

[BGZ17] S. Beck, J. Günthör, C. Zorn. "Simulation-based Resilience Prediction of Microservice Architectures." Specialist Research Software Engineering. University of Stuttgart, 2017 (cit. on pp. 2, 3, 15, 16, 31).

[BHJ16] A. Balalaie, A. Heydarnoori, P. Jamshidi. "Microservices architecture enables devops: Migration to a cloud-native architecture." In: *IEEE Software* 33.3 (2016), pp. 42–52 (cit. on p. 13).

[BHK11] F. Brosig, N. Huber, S. Kounev. “Automated Extraction of Architecture-level Performance Models of Distributed Component-based Systems.” In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 183–192 (cit. on p. 20).

[BHW+15] A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, et al. “Performance-oriented DevOps: A research agenda.” In: *arXiv preprint arXiv:1508.04752* (2015) (cit. on pp. 20, 45).

[BL94] T. Ball, J. R. Larus. “Optimally profiling and tracing programs.” In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.4 (1994), pp. 1319–1360 (cit. on p. 22).

[BMMM98] W. H. Brown, R. C. Malveau, H. W. " McCormick, T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. 1st. New York, NY, USA: John Wiley & Sons, Inc., 1998 (cit. on p. 9).

[BSG+17] J. Byrne, S. Svorobej, K. Giannoutakis, D. Tzovaras, P. Byrne, P.-O. Östberg, A. Gourinovitch, T. Lynn. “A Review of Cloud Computing Simulation Platforms and Related Environments.” In: *CLOSER*. 2017, pp. 679–691 (cit. on pp. 28, 29).

[Com18] Docker, Inc. *Overview of Docker Compose*. URL: https://docs.docker.com/compose/overview/ (visited on 08/12/2018) (cit. on p. 21).

[Con68] M. E. Conway. “How do committees invent.” In: *Datamation* 14.4 (1968), pp. 28–31 (cit. on p. 14).

[Dav95] A. M. Davis. “Tracing: A Simple Necessity Neglected.” In: *IEEE Software* 12 (Sept. 1995), pp. 6–7 (cit. on p. 21).

[DH17] T. F. Düllmann, A. van Hoorn. “Model-driven Generation of Microservice Architectures for Benchmarking Performance and Resilience Engineering Approaches.” In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ACM. 2017, pp. 171–172 (cit. on p. 67).

[Doc18] Docker, Inc. *What is a Container*. URL: https://www.docker.com/resources/what-container (visited on 08/12/2018) (cit. on p. 21).

[Fea18] *Feathers*. URL: https://docs.feathersjs.com/ (visited on 08/13/2018) (cit. on p. 90).

[FL14] M. Fowler, J. Lewis. *Microservices: A definition of this new architectural term*. 2014. URL: https://martinfowler.com/articles/microservices.html (visited on 04/04/2018) (cit. on pp. 1, 13–15, 21).

[Fow14] M. Fowler. *CircuitBreaker*. 2014. URL: https://martinfowler.com/bliki/CircuitBreaker.html (visited on 04/05/2018) (cit. on pp. 11, 31).

[Fri15] U. Friedrichsen. *Eine kurze Einführung in Resilient Software Design*. 2015. URL: https://jaxenter.de/unkaputtbar-einfuehrung-resilient-software-design-15119 (visited on 01/22/2018) (cit. on p. 8).

[Fri16] U. Friedrichsen. *Resilient Software Design - Robuste Software Entwickeln*. 2016. URL: https://www.informatik-aktuell.de/entwicklung/methoden/resilient-software-design-robuste-software-entwickeln.html (visited on 02/28/2018) (cit. on p. 9).

[Gam95] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995 (cit. on p. 1).

[GCD+17] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, A. Di Salle. "Towards recovering the software architecture of microservice-based systems." In: *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*. IEEE. 2017, pp. 46–53 (cit. on pp. 26, 27, 45).

[Goe02] B. Goetz. *Thread pools and work queues*. 2002. URL: https://www.ibm.com/developerworks/java/library/j-jtp0730/index.html (visited on 08/12/2018) (cit. on p. 12).

[Hai09] Y. Y. Haimes. "On the definition of resilience in systems." In: *Risk Analysis: An International Journal* 29.4 (2009), pp. 498–501 (cit. on p. 8).

[HDAP18] A. van Hoorn, T. F. Düllmann, A. Aleti, T. Pitakrat. "Efficient Chaos: Pattern-based Fault Injection in Microservices." In: In preparation. 2018 (cit. on pp. 2, 20, 55–58, 60).

[HE17] S. Hukerikar, C. Engelmann. *Resilience Design Patterns: A Structured Approach to Resilience at Extreme Scale (Version 1.2)*. Tech. rep. ORNL/TM-2017/745. Oak Ridge, TN, USA: Oak Ridge National Laboratory, 2017 (cit. on p. 8).

[Hys18] *Hystrix*. URL: https://github.com/Netflix/Hystrix (visited on 08/11/2018) (cit. on pp. 11, 31).

[Ja18] *Jaeger: open source, end-to-end distributed tracing*. URL: https://www.jaegertracing.io (visited on 07/14/2018) (cit. on p. 24).

[Kum18] A. Kumar. *Circuit Breaker Pattern*. 2018. URL: https://dzone.com/articles/circuit-breaker-pattern (visited on 07/20/2018) (cit. on p. 31).

[Lab17] C. Laboratory. *Cloud Sim*. 2017. URL: http://www.cloudbus.org/cloudsim/ (visited on 04/10/2018) (cit. on p. 29).

[Lyu07] M. R. Lyu. “Software reliability engineering: A roadmap.” In: *Future of Software Engineering, 2007. FOSE'07*. IEEE. 2007, pp. 153–170 (cit. on p. 8).

[Nak15] H. Nakama. *Inside Azure Search: Chaos Engineering*. 2015. URL: https://azure.microsoft.com/de-de/blog/inside-azure-search-chaos-engineering/ (visited on 08/12/2018) (cit. on p. 20).

[NCM16] R. Natella, D. Cotroneo, H. S. Madeira. “Assessing Dependability with Software Fault Injection: A Survey.” In: *ACM Comput. Surv.* 48.3 (2016), 44:1–44:55 (cit. on pp. 1, 19, 55).

[Net11] Netflix, Inc. *The Netflix Simian Army*. 2011. URL: https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116 (visited on 04/04/2018) (cit. on pp. 15, 20).

[Net16] Netflix, Inc. *Netflix Simian Army*. 2016. URL: https://github.com/Netflix/SimianArmy (visited on 07/25/2018) (cit. on pp. 20, 55).

[Net18] Netflix, Inc. *Hystrix Wiki*. URL: https://github.com/Netflix/Hystrix/wiki (visited on 02/14/2018) (cit. on pp. 12, 13, 32).

[New15] S. Newman. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015 (cit. on pp. 1, 13, 14, 22).

[NJ09] T. D. Nielsen, F. V. Jensen. *Bayesian networks and decision graphs*. Springer Science & Business Media, 2009 (cit. on p. 57).

[Nyg07] M. T. Nygard. *Release It!: Design and Deploy Production-Ready Software*. The Pragmatic programmers, 2007 (cit. on pp. 1, 9–12).

[Orc18] *ORCAS: Efficient Resilience Benchmarking of Microservice Architectures*. URL: https://github.com/orcas-elite (visited on 07/26/2018) (cit. on pp. 1, 55).

[OT18] *OpenTracing Documentation*. URL: http://opentracing.io/documentation/ (visited on 07/12/2018) (cit. on pp. 21, 22, 46).

[OTS1818] *The OpenTracing Semantic Specification*. Version 1.1. 2018. URL: https://github.com/opentracing/specification/blob/master/specification.md (visited on 07/13/2018) (cit. on p. 22).

[Pea14] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier, 2014 (cit. on p. 57).

[Raj16] R. Rajesh. *Spring Microservices*. Packt Publishing Ltd, 2016 (cit. on pp. 14, 15).

[RHB+17] C. Rosenthal, L. Hochstein, A. Blohowiak, N. Jones, A. Basiri. *Chaos Engineering: Building Confidence in System Behavior through Experiments*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2017 (cit. on p. 1).

[Ric18] C. Richardson. *Pattern: Circuit Breaker*. URL: http://microservices.io/patterns/reliability/circuit-breaker.html (visited on 07/20/2018) (cit. on p. 31).

[Rom17] M. Rombach. "Enabling Architectural Performability Analyses for Microservices via Design Pattern Completions." Master's Thesis. Department of Informatics, Karlsruhe Institute of Technology, 2017 (cit. on p. 27).

[Šab18a] N. Šabić. *OpenTracing: Distributed Tracing's Emerging Industry Standard*. 2018. URL: https://sematext.com/blog/opentracing-distributed-tracing-emerging-industry-standard/ (visited on 07/14/2018) (cit. on pp. 46, 49).

[Šab18b] N. Šabić. *OpenTracing: Jaeger as Distributed Tracer*. 2018. URL: https://sematext.com/blog/opentracing-jaeger-as-distributed-tracer/ (visited on 07/14/2018) (cit. on pp. 25, 46).

[SBB+10] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, C. Shanbhag. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010 (cit. on p. 21).

[Shk17] Y. Shkuro. *Evolving Disributed Tracing at Uber Engineering*. 2017. URL: https://eng.uber.com/distributed-tracing/ (visited on 07/14/2018) (cit. on p. 25).

[Sig16] B. Sigelman. *Towards Turnkey Distributed Tracing*. 2016. URL: https://medium.com/opentracing/towards-turnkey-distributed-tracing-5f4297d1736 (visited on 07/12/2018) (cit. on pp. 21, 22, 24).

[Spr18] *Getting Started: Circuit Breaker*. URL: https://spring.io/guides/gs/circuit-breaker/ (visited on 07/20/2018) (cit. on p. 32).

[SW03] C. U. Smith, L. G. Williams. "More new software performance antipatterns: Even more ways to shoot yourself in the foot." In: *Computer Measurement Group Conference*. 2003, pp. 717–725 (cit. on pp. 1, 10).

[Tri17] D. Trivedi. *Circuit Breakers and Microservices Architecture*. 2017. URL: http://techblog.constantcontact.com/software-development/circuit-breakers-and-microservices/ (visited on 07/20/2018) (cit. on p. 31).

[Twi18] Twitter, Inc. *Finagle*. URL: https://twitter.github.io/finagle/guide/index.html (visited on 07/20/2018) (cit. on p. 32).

[Uni17] D. o. C. S. University of Hamburg. *Desmo-J: Quick Overview*. 2017. URL: http://desmoj.sourceforge.net/overview.html (visited on 04/05/2018) (cit. on p. 16).

[VMSK12] M. Vieira, H. Madeira, K. Sachs, S. Kounev. “Resilience benchmarking.” In: *Resilience Assessment and Evaluation of Computing Systems*. Springer, 2012, pp. 283–301 (cit. on p. 55).

All links were last followed on August 14, 2018.

Appendix A

Additional Metrics

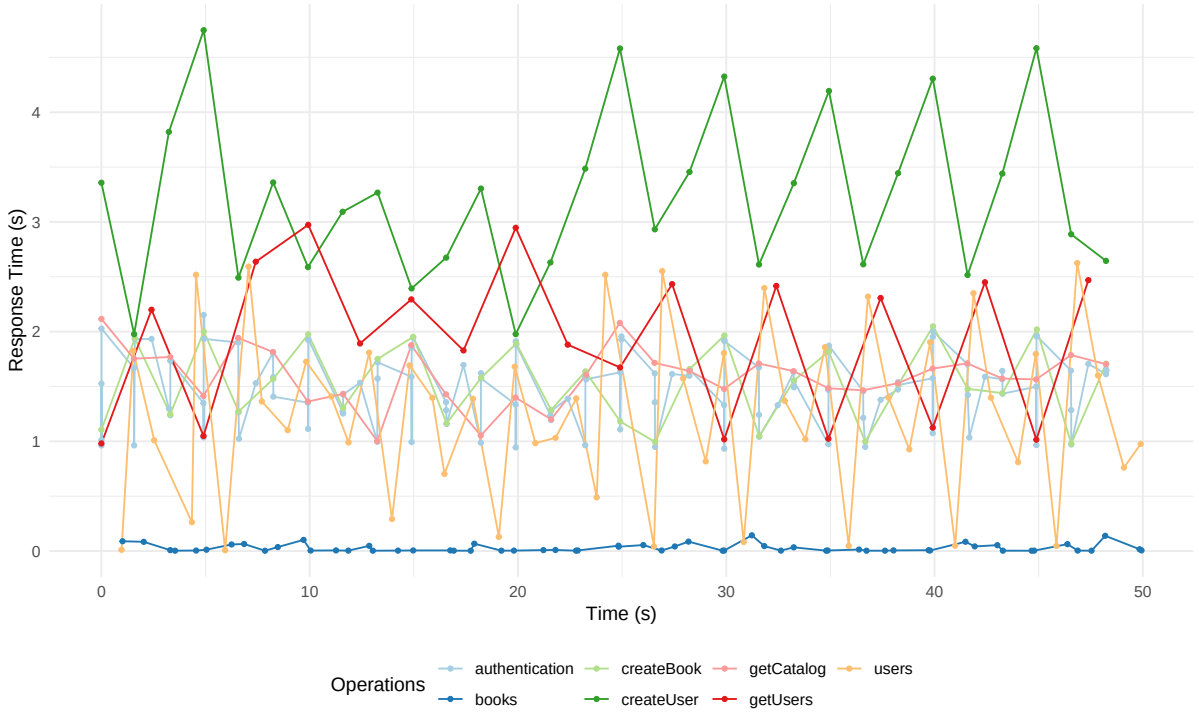


Figure A.1.: Measured response times recorded during Experiment 2.2.

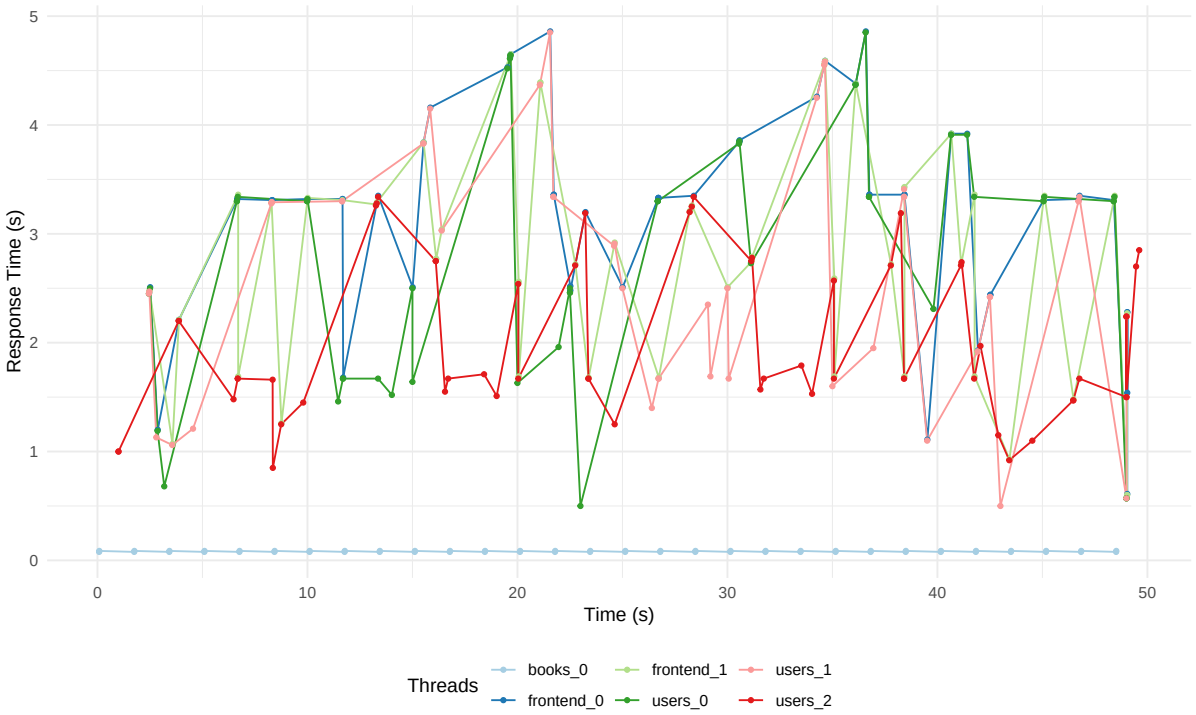


Figure A.2.: Simulated response times created during Experiment 2.2.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature