Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Simulating Scenario-based Chaos Experiments for Microservice Architectures

Lion Wagner

Course of Study:	Softwaretechnik
Examiner:	Dr.-Ing. André van Hoorn
Supervisor:	Dr.-Ing. André van Hoorn Sebastian Frank, M.Sc. Mir Alireza Hakamian, M.Sc.
Commenced:	December 1, 2020
Completed:	June 1, 2021

Abstract

Context. With the growing popularity of microservice-based architectures, the need for effective resilience testing of such architectures occurred. In a preceding case study, we showed that transforming resilience scenarios to formalized scenario-based chaos tests, and executing those is a feasible way to do so.

Problem. While producing very representative results, chaos testing can require a not insignificant expenditure of time and stresses the system under test. Simulating such experiments reduces these problems. Unfortunately, there are currently no simulators available that fulfill the requirements for simulating such scenarios to an acceptable level.

Objective. Therefore, this thesis examines which simulators are suitable for which types of scenarios. Furthermore, the most promising of these simulators is extended to support a common scenario description and other features.

Method. To properly elicit the requirements for such a simulator, stakeholders conduct a requirements analysis. Existing simulators are searched and evaluated based on these requirements. The simulator that looks the most promising is then extended. To verify the accuracy of the simulator, the scenarios from the preceding case study are utilized. They are transposed to models and simulated. The result data of the simulation is compared to the results of the case study.

Result. This thesis presents five microservice simulators and which scenarios they currently potentially support best in a structured overview. Further, a re-engineering of the MiSim simulator results in better support of scenario-based chaos experiments and others of the aforementioned requirements. Conclusion. MiSim 3.0 is evaluated as a simulator that is capable of accurately simulating scenario-based chaos tests. Specifically, the newly implemented resilience patterns and chaos injections behave as expected. However, an inaccurate calibration may harm its accuracy.

Conclusion. Previously existing microservice simulators could not simulate all types of scenario-based chaos experiments. In the context of this thesis *MiSim* 3.0 is created and evaluated as a simulator capable of correctly simulating many types of scenario-based chaos tests. In particular, the newly implemented resilience patterns and chaos injections behave as expected. However, inaccurate calibration can significantly affect its accuracy.

Kurzfassung

Kontext. Mit der wachsenden Popularität von Microservice-basierten Architekturen ist der Bedarf an effektiven Resilienz-Tests für solche Architekturen entstanden. In einer vorangegangenen Fallstudie haben wir gezeigt, dass die Umwandlung von Resilience-Szenarien in formalisierte szenario-basierte Chaostests und deren Ausführung ein praktikabler Weg ist, dies zu tun.

Problemstellung. Obwohl Chaostests sehr repräsentative Ergebnisse liefern, erfordern sie einen nicht unerheblichen Zeitaufwand und belastet zudem das zu testende System. Das Simulieren solcher Experimente reduziert diese Probleme. Leider gibt es derzeit keine Simulatoren, die die Anforderungen für die Simulation solcher Szenarien auf einem akzeptablen Niveau erfüllen.

Zielsetzung. Daher wird in dieser Arbeit untersucht, welche Simulatoren für welche Arten von Szenarien geeignet sind. Darüber hinaus wird der vielversprechendste dieser Simulatoren erweitert, um eine allgemeine Szenario Beschreibung und andere Resilienz-Features zu unterstützen.

Methode. Um die Anforderungen an einen solchen Simulator richtig zu erheben, wird Anforderungsanalyse mithilfe von Experten durchgeführt. Bestehende Simulatoren werden auf Basis dieser Anforderungen gesucht und gewertet. Der Simulator, der am vielversprechendsten erscheint, wird dann erweitert. Zur Überprüfung der Genauigkeit des Simulators, werden die Szenarien aus der vorangegangenen Fallstudie herangezogen. Sie werden in Modelle übertragen und simuliert. Die Ergebnisdaten der Simulation werden mit den Ergebnissen der Fallstudie verglichen.

Ergebnis. Diese Arbeit stellt fünf Microservice-Simulatoren und welche Szenarien sie derzeit potenziell am besten unterstützen vor. Weiterhin wird durch ein re-Engineering des *MiSim*-Simulators eine bessere Unterstützung von szenario-basierten Chaos-Experimenten und anderen der oben genannten Anforderungen erreicht.

Fazit. Bisher existierende Microservice-Simulatoren konnten nicht alle Arten von szenariobasierten Chaostests simulieren. Im Rahmen dieser Arbeit wird mit *MiSim* 3.0 ein Simulator erstellt und evaluiert, der in der Lage ist, viele Arten von szenariobasierte Chaostests korrekt zu simulieren. Insbesondere die neu implementierten Resilienz-Muster und Chaos-Injektionen verhalten sich wie erwartet. Allerdings kann eine ungenaue Kalibrierung seine Genauigkeit erheblich beeinträchtigen.

Contents

Acronyms

1 Introduction

Today, the resilience of software systems plays a crucial role in their life cycle. Specifically microservice architectures often appear in this context as they naturally lean towards maintainability and flexibility [DGL+17]. To test the resilience, usually chaos tests are utilized [Cha18]. The feasibility and usage of scenario-based chaos experiments was demonstrated in our preceding case study [KWK+20]. These scenario-based chaos tests and experiments can offer a well-defined and formal description of a system's quality requirements. This allows for using these scenarios as inputs for tools, that quantitatively evaluate a quality requirement, like simulators.

Usually, scenario-based chaos tests are run in production to create realistic experiment results [Cha18]. However, they can take a not insignificant expenditure of time. Therefore, configuration optimization and immediate testing of system resilience is not feasible. Parallel or preemptive simulation of chaos test scenarios reduces these problems, since simulations are usually cheap, repeatable and do not impact the production system. There are currently many existing simulators for distributed systems [BBM13; CRRB09; Flo; NGN17]. Most of them support some for of load testing, but to my knowledge only two of these simulators do support the simulation of some chaos toolkit like faultloads [BZG17; VDR+20]. None are present when it comes to taking a scenario-based chaos experiment description as input.

Even tho none of these simulators support scenario-based chaos experiments, they can still be used to execute some specific types of resilience scenarios. This thesis aims to establish an overview of existing microservice system simulators and presents which types of scenario-based chaos experiments they can simulate best. The most promising of these simulators, *MiSim* [BZG17] then gets extended to support a scenario-based input format. In the end, this extension should help to build a common framework for doing quantitative quality attribute software analysis with resilience scenario-based chaos test, by providing a simulator that executes such tests as inputs.

Before going into detail about the simulators, a list of requirements for a scenario-base simulation is established by leveraging expert knowledge. Based on these requirements a structured search reveals five potentially usable simulators. An evaluation of these simulators presents their respective (dis-)advantages and supported types of resilience scenarios. Lastly, the simulator *MiSim* is extended and evaluated on how good it can potentially simulate scenario-based chaos tests. The scenarios that are picked for the evaluation are selected form a catalog that we previously established during an industry case study.

The evaluation shows, that *MiSim* fulfills most of the aforementioned requirements after the re-engineering. Most notibly, it accepts a common scenario description as input. Also, it is capable of simulating all common chaos tool kit like faultloads such as delays, chaos monkeys or complex load behaviors. Additionally, more resilience patterns such as retries and autoscalers have been added. Besides these improvements, the exact calibration of the architecture model is still an open problem. The evaluation results show, that specifically varying workloads can are not simulated accurately.

Concluding, this thesis gives an overview of existing microservice architecture simulators and how good they can potentially simulate scenario-based chaos tests. The simulator *MiSim* gets re-engineering to be more feature rich and to accepted scenario-based chaos tests as simulation inputs. It is shown to accurately simulate real world scenarios and resilience patterns.

Thesis Structure

Chapter 2 – Foundations: This chapter contains explanations of the main topics that are relevant within this theses.

Chapter 3 – Related Work: This chapter introduces websites, papers, books and research projects that are closely realted to this thesis.

Chapter 4 – Requirements Analysis and Simulator Evaluation: This chapter presents the requirements analysis and analized simulators and the results of the simulator evaluation.

Chapter 5 – Improvement of the *MiSim* Simulator: This chapter offers and overview of the imporvements made during the reengineering of the *MiSim* simulator.

Chapter 6 – Evaluation: This chapter presents the evaluation results for the improved and extend simulator.

Chapter 7 – Conclusion and Future Work: This chapter finishes off the thesis by providing an outlook into potential continuing future work and gives a conclusion and summary of the thesis.

2 Foundations

The roots of this thesis lie within five fundamental concepts of software engineering, distributed systems and simulation. Each of these concepts is dedicated a section with the following order. Section 2.1 introduces the concept of microserivce architectures, a very loose approach to software architectures. Section 2.2 explains resilience engineering as a strategy to make an architecture capable of coping with and preventing failures. Further, the concept of resilience patterns is introduced. Section 2.3 continues by describing resilience scenarios as an approach to formalize the resilience requirements of a software system. Section 2.4 then presents chaos engineering as a way of testing the resilience of a system. Lastly, Section 2.5 give a general overview over discrete-event simulation and how it is applicable to microservice architectures.

2.1 Microservice Architectures

A microservice architecture is a form of a service orientated architecture (SOA), sometimes labeled as “SOA done right” [BG20]. It enforces a very loose coupling and high cohesion of services [New14]. In practice each service should be as independent as possible and only handle one specific type of tasks or business processes [DGL+17]. This is also where the name “microservice” comes from [LF14].

This loose coupling allows a microservice architecture to be more flexible, scalable, maintainable and extendable, than a more monolithic structure [New14]. Especially since the surge in PaaS providers like Amazon Webservices, Microsoft Azure or Cloud foundry, microservices have become easy to deploy and manage, due to then naturally leaning towards containerization.

In 2018 DZone conducted a survey on the popularity of microservices [Gle18]. Nearly 50% of the 732 participants answered that their company is using microservices in development or production. Additional 38.7% of participants said, that their company is at least considering a migration to microservices. Another survey in early 2020 by the O’Reilly Media, Inc. [LS20] reported that of the 1502 survey respondents, over 76% answered that their company uses microservices in some way or another. Further, over 88% of these microservice systems are older than a year. Therefore, could be considered as non-prototype or permanent installment. In both studies the main selected reasons for moving to microservices were the aforementioned flexibility, scalibility and maintainability (here: responding quickly to changing requirements). A majority of respondents rated their adoption as at least mostly successful. The results of these surveys show, that microservices are becoming more and more present in the industry. This makes development-supporting tools, like microservice system simulators, significantly more important.

Despite all their advantages, microservices are not the perfect all-in-one solution for any system. Previous adoptions of this architecture failed or had only minor success. Reasons for this are mainly their context specific applicability and immature development processes [Vuč20]. Specifically, the extremely loose coupling of services brings some unique challenges. Since microservice architectures change a lot during their run time, they have to be resilient against all sorts of communication problems and challenges. These can range from detecting and handling bad connections, over discovering each other reliably, to efficiently managing service resources.

2.2 Resilience Engineering and Patterns

The term *resilience* can be found in many domains. Hollnagel explains some of the origins of resilience on his website [Hol16]. *Resilience* first appeared in the craft of woodwork, where the term describes the response of different wood types to sudden and severe loads. This was later adopted to describe all kinds of materials besides wood. In 1973 the first formal definition of Resilience was introduced by Holling [Hol73] to describe how ecosystems cope with rapid and impacting changes in the environment without ceasing to exist. Today, there are many definitions for *resilience* as a lot of different disciplines, like mechanical engineering, social-technical engineering or psychology have adopted this concept [Bur19; Fur15; Hol16; MJ09]. Erik Hollnagel refined his definition of resilience over multiple years and his newest iteration describes “being resilient” as follows:

Definition 2.2.1 (resilience)
A system is resilient if it can adjust its functioning prior to, during, or following events (changes, disturbances, and opportunities), and thereby sustain required operations under both expected and unexpected conditions. [Hol16]

Resilience engineering is the activity of strategically modifying the resilience of a system. In the context of a software this means that a the system being engineered to withstand turbulent and unexpected conditions [MJ09].

As mentioned in Section 2.1, there common challenges for the resilience of microservice architectures. Since these are reoccurring problems, template-like solutions to most of these problems were created over time. One subset of these solutions are *resilience patterns* [Nyg07]. These are software-based solutions that aim to make a system more resilient. For example, a *Retry pattern* retries failed operations [Res] and a *Circuit Breaker* forces an alternative fallback behavior during a failure [JGC17b]. Resilience patterns often have a lot of different configuration options. A simulation of scenario-based chaos experiments can help to efficiently test and find effective configurations for resilience patterns or in general test their applicability.

As Circuit Breaker, Retrier, Loadbalancer and Autoscaler are the most prominent resilience patterns that appear in this thesis, they will be presented in detail in the following subsections.

2.2.1 Retry

The retry resilience pattern is one of most established mechanisms that helps to cope with a communication failure. It utilizes the simple strategy of resending failed requests after a short delay. To prevent network and queue saturation a retry pattern is usually configured to have a maximum number of tries per request [MACG20]. This interaction is visualized in Figure 2.1. A variety of algorithms can be employed to calculate backoff delay before the next try. The most common implementations for a backoff algorithm are the linear and exponential backoff strategies [Bro19b].

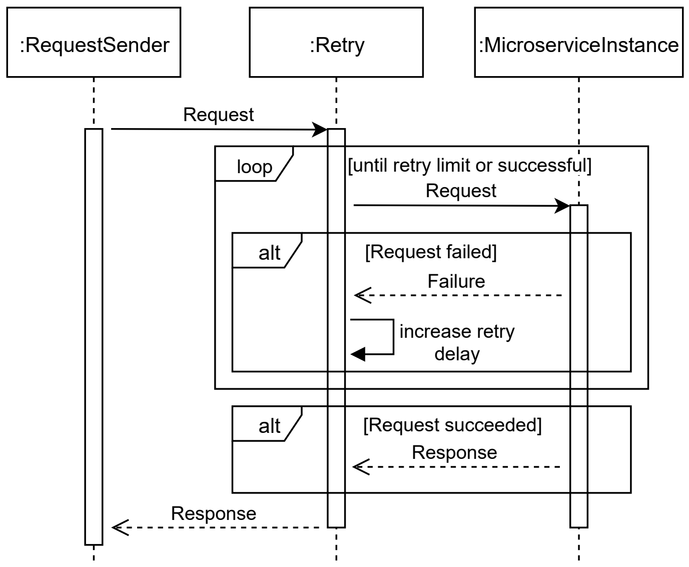

Figure 2.1: General behavior of the Retry pattern [MACG20].

The general forumla for a linear backoff is described by Equation (2.1). It uses the parameters shift s, base delay b and the number of previous tries (#tries) to calculate the delay in between tries d_{next}. Equation (2.2) displays a similar formula for the exponential backoff. It takes an additional parameter g that determines the growth rate of the backoff. Both of these implementations are capped by a maximum delay $d_{\max}$ to prevent infinite retry delays [Bro19b].

$$d_{next} = \min(d_{\max}, s + b \cdot \#\text{tries}) \tag{2.1}$$

$$d_{next} = \min(d_{\max}, s + b \cdot g^{\#\text{tries}}) \tag{2.2}$$

Both of these formulas are pure deterministic approaches. Whilst these algorithms usually yield acceptable results, they have a inherited disadvantage compared to randomized approaches. Requests that fail at a similar time will also be retried to a similar time. Therefore, load spikes can emerge and collisions would repeat. This is specifically the case if a connection fails only for a short period of time. To handle this inconvenience randomized wrappers or functions can be utilized. The slight non-deterministic delaying of messages is known as "Jitter" or "jittering". This technique is generally used to smooth out the load curve of a system [Bro19b]. Brooker [Bro15] distinguishes between three types of jittering retries. Equation (2.3) shows an implementation of a "Full Jitter" wrapper. This method takes the output of a linear or exponential backoff algorithm and returns a random value between 0 and the original deterministic value. An "Equal Jitter" implementation is displayed by Equation (2.4). This version of a non-deterministic algorithm produces more stable results than a "Full Jitter", since only one half of the next delay is randomized. Both the "Full" and "Equal" Jitter can be based on a linear or exponential strategy providing the d_{next} argument based on the previous tries. Lastly Brooker [Bro15] also presents the "Decorrelated Jitter", which is represented in Equation (2.5). This implementation returns a base delay for its first iteration and then continues to use a random value between the base and the previous delay times a constant $n \in \mathbb{R}_{>1}$. The later determines the asymptotic growth rate of the function.

(2.3) $d_{next} = \text{random}(0, d_{next})$

(2.4) $d_{next} = \frac{d_{next}}{2} + \text{random}(0, \frac{d_{next}}{2})$

(2.5) $d_{next} = \min(d_{\max}, \text{random}(b, d_{next} \cdot n))$

Which of these implementations works best is always down to the types of workloads. In general, exponential backoffs are prioritized over linear ones to give less priority to already failed requests [Bro19b]. In any case, jittering is strongly recommended [Ani21; Bro19a].

2.2.2 Circuit Breaker

A circuit breaker pattern is another technique to increase a systems resilience [JGC17a]. Circuit breakers monitor the connection between two services. Mendonca et al. [MACG20] describe the general circuit breaker behavior as the following. If enough requests of a connection fail, so that a failure threshold is reached, the circuit breaker opens. In the open state a circuit breaker blocks the client from sending new requests and lets them immediate fail. This coheres to the fail-fast principle [Sho04]. After a while, the circuit breaker goes into a half-open state. In this state it lets through a small amount of messages (usually only one) to check whether the connection is still bad. In case the requests are successful the circuit breaker closes again. Otherwise, it returns into its open state. This behavior is visualized as a sequence diagram in Figure 2.2. A practical circuit breaker implementation can take over additional responsibilities [JGC17b]. E.g. a Hystrix-based circuit breaker also takes care of caching, limiting the number of concurrent requests, activating fallback behavior and has to handle timeouts [JGC17a]. All these properties allow a service to react quickly to other service's failures and to automatically and passively check for the connection status.

2.2.3 Loadbalancer

In modern distributed system, services are designed to have multiple instances. This raises the problem of fairly distributing messages between them. A bad distribution can lead to unused or overused instances and potential SLO violations. Therefore, good and efficient load balancing is one of the key disciplines of a distributed system. It should ensure fairness and performance when it comes to distributing requests or processes. Over time, many approaches to load balancing have emerged [NLL18; SSS08; YKXY19]. When looking at architectural properties, it can be distinguished between a client-side and server-side load balancers [CCY99]. Sometimes these are also called "Instance-/Service-oriented" [NLL18] or "(de-)centralized" [KS18].

When using server-side load balancing, the clients send all requests to a central load balancer node, that has the knowledge of all existing instances. This node then decides where the request should be sent. This has the advantage, that all message can be scanned when passing and that the back end structure is hidden from the client [Sha20]. In the client-side style, a public registry of instances is held within the network. The client has to query it periodically before requests can be send. Based on the list the client itself has to choose a target instance, without knowing the decisions of other clients. This removes the central load balancer as a single point of failure and can potentially speed up communication as there is one less jump a request has to make [Sha20]. Niu

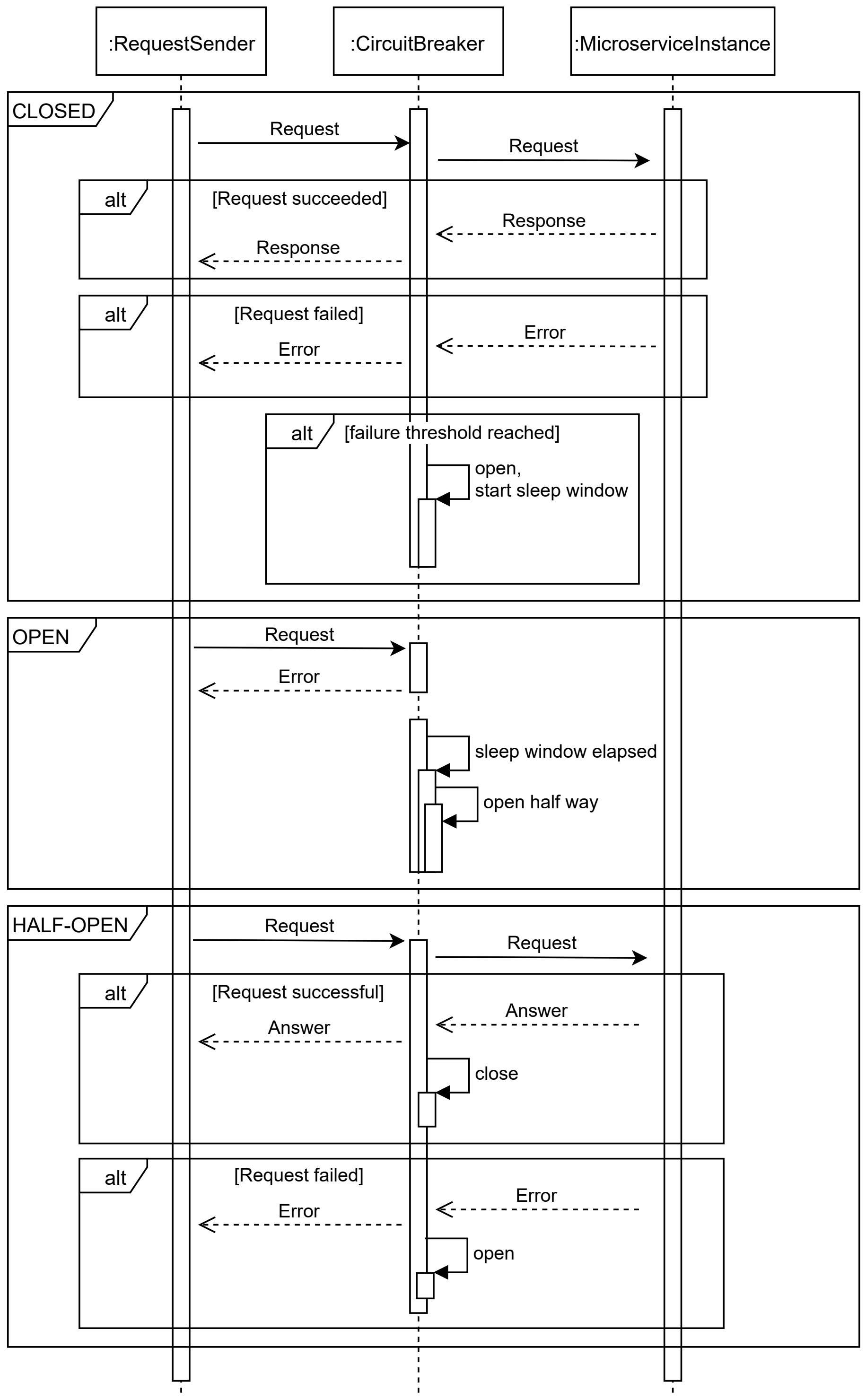

Figure 2.2: General behavior of the Circuit Breaker pattern [MACG20].

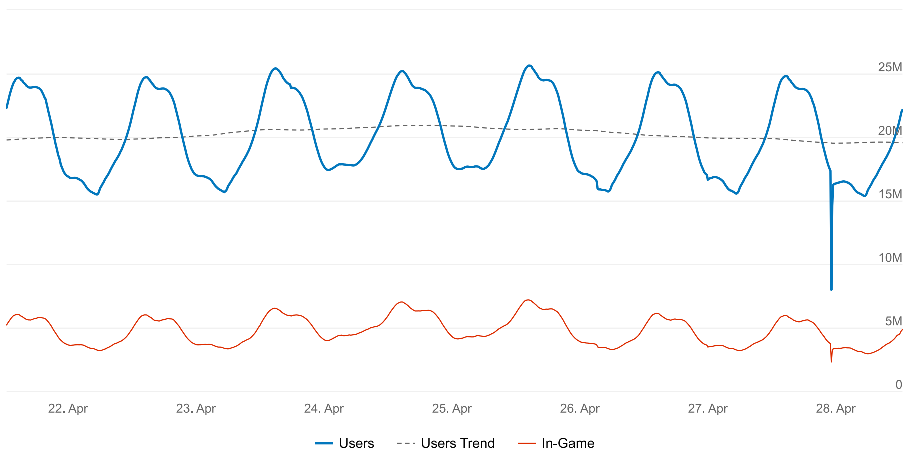

Figure 2.3: Amount of concurrent Steam users from April 22nd to April 28th in 2021 (relative to UTC+0) [DB21].

et al. [NLL18] also present microservice-orientated load balancing, where every service is assigned its own designated load balancing node. This structure combines the advantages and disadvantages of the aforementioned techniques.

2.2.4 Autoscaler

Software systems that provide services often have very varying workloads [ZQL+16]. A public and practical example for this would be the amount of concurrent *Steam* users. *Steam* is one the worlds the largest platform for distributing and managing video game libraries and used primarily in Europe and America [Val21]. Figure 2.3 displays the typical, periodic course of the number of concurrent logged-in users. Even tho the trend stays roughly consistent, the amount of active users can vary by ten million within a day. One solution to cope with such varying and partially unpredictable workloads is to scale a system's capacity relative to the current experienced workload. This can be done horizontally (spawning new servers/instances) or vertically (increasing resources of nodes. Since microservice architectures are specifically designed to scale by duplicating instances, horizontal techniques are strongly preferred with this architectural style. However, it is also possible to scale a service vertically by providing more resources (e.g. CPU Cores, RAM) to an active instances.

Autoscaling approaches this solution with different options. For periodic workload variations, like the one in Figure 2.3, a periodic scaling strategy can be applied, that automatically scales the system at a specific time. Other popular techniques include trend-based scaling [YF14] or machine-learning-based scaling [PAP+18]. In the end, each strategy tries to optimize the utilization of the current system, to prevent performance problems or the overconsumption of resources.

2.3 Software Scenarios

Scenarios in the context of software engineering are used when describing the behavior of a system in specific situations [BCK03]. They are derived from the requirements of a software and can be part of its specification. Scenarios do not contain detailed information regarding architecture or business logic. Rather, they are a short description of how a system should react to an event whilst being in a specific state. The idea behind scenarios is to improve vague QoS requirements by looking at specific impact situations and capturing their context [KABC96]. *Resilience scenarios* are a specific kind of scenarios that focus on resilience requirements.

Figure 2.4 shows the structure of a scenario. Six different concepts are part of a scenario:

Stimulation Source:	The *stimulation source* is an entity that interacts with the system.
Stimulus:	The *Stimulus* describes the event or message that is sent by the *stimulation source* and received by the *Artifact*.
Artifact:	An *Artifact* is the part of the software architecture that is stimulated by the *Stimulus* and therefore the one which has to react to it.
Environment:	The *Environment* symbolizes the situation or state in which the software is in.
Response:	How the *Artifact* handles the incoming *Stimulus* is described in the *Response*. However, intermediate steps are often neglected and only the overall reaction is described.
Response Measure:	To use scenarios as a verification method they have to be testable. For this the *Response Measure* provides a quantifiable behavior description of how the system should behave during the respective scenario.

Figure 2.5 shows an example instance for a resilience scenario. The system is in a "normal operation" environment. A heartbeat monitor is the stimulus source. It detects that a server is unresponsive and sends a "server unresponsive" message. The artifact is represented by a process that receives and accepts this message. There are two expected responses. (1) An operator should be informed and (2) the process should go back to normal operation. The response measure for this case is the downtime, which is expected to be zero.

2.4 Chaos Engineering

Definition 2.4.1
Chaos Engineering is the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production. [Cha18]

Chaos engineering, sometimes called *chaos testing* [RA20; Sma21], plays its part in testing the resilience of a software system. As Definition 2.4.1 describes, it introduces turbulent conditions, which are more informally called *chaos*, into a system to see how it behaves under unpredictable, uncommon or extreme conditions. This is done to assess the capabilities of the system in terms of its resilience. Additionally, it is usually applied in a production environment to enforce the idea of building for failure [Cha18].

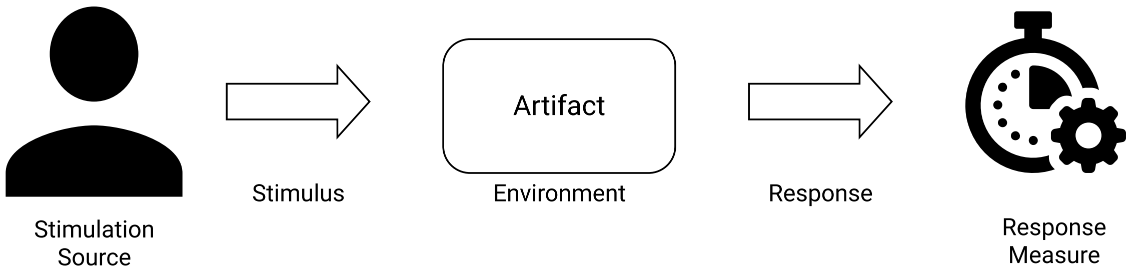

Figure 2.4: General structure of a scenario consisting out of Stimulation Source, Stimulus, Artifact, Environment, Response and Response Measure [BCK03].

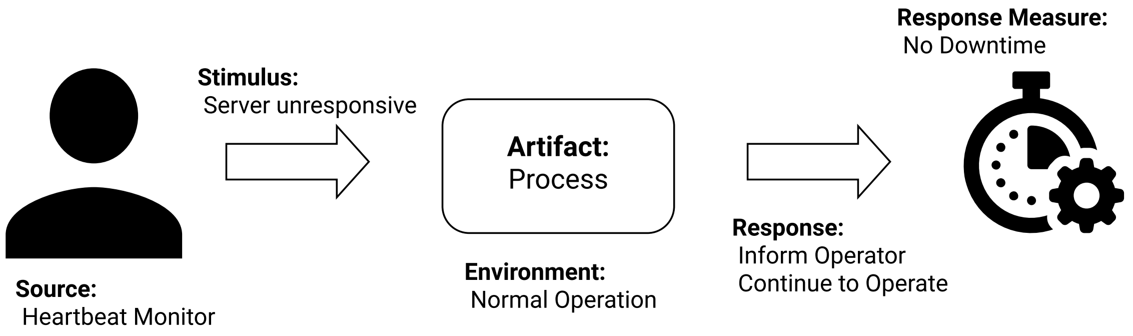

Figure 2.5: Example instance of an availability scenario [BCK03].

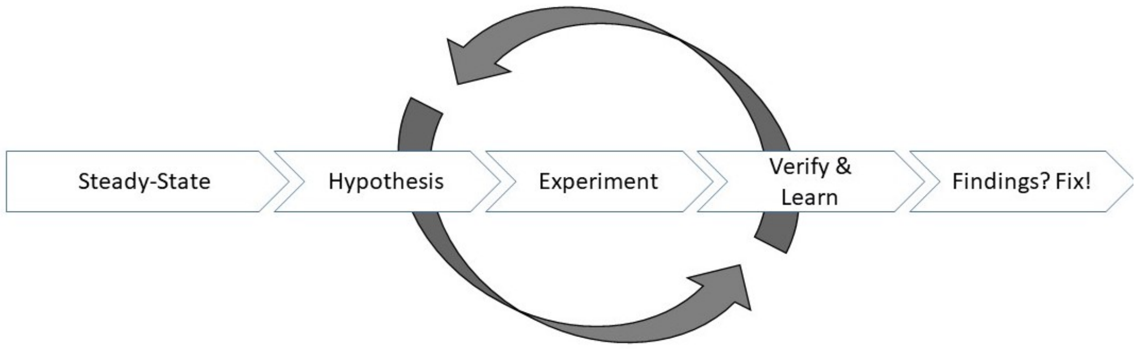

Figure 2.6: The learning loop of chaos engineering [Blo20]

The process of running a chaos experiment always has similar stages. Figure 2.6 demonstrates its typical learning loop. It starts of with describing the *steady-state* of a system. A steady-state describes the target or "normal" status. It should be quantifiable and representative for the healthiness of the system. This is done by defining regular value ranges for specific system or business metrics. Netflix for example use stream starts per second as one of their major metrics to see whether their system is healthy [BBR+16]. A more abstract example could be the body temperature of a human. In that case, the steady-state could be defined by the value range of 37 ± 0.5 °C [RHB+17]. Once the steady-state behavior is defined, the actual testing begins. First, a quantifiable hypothesis is established that describes how the system should behave under the subsequent test. Then the tests are executed. They can consist out of various disruptions such as instance failures, latency injections or traffic limiting. During the execution, data of the system is collected to verify the hypothesis. Afterwards it can be evaluated whether the hypothesis held and how confident a developer can be in the behavior of the system in its current state [BBR+16; RHB+17].

2.5 Discrete-Event System Simulation

In the context of digital systems, almost all interactions are inherently discrete due to their reliance on binary units (bits). Therefore, a simulation of such a system naturally leans towards a discrete-event simulation (DES) [BCNN10]. This can be seen in practice when looking at existing software system or queueing network simulators [BBM13; BZG17; nsn21; ZGD19].

When using a DES engine, such as DESMO-J [PL99], the state changes of the simulated system are represented as events. Each event is scheduled at specific target simulation time. Usually, some immediate initial events (target time = 0) kick off the simulation. Then, events themselves can modify the systems' state and schedule other events during the simulation. All events are held in a priority queue, sorted by their target time and scheduling priority. When simulating, the engine dequeues the next event and advances the simulation clock to its target time. Then the event gets executed. The simulation ends, when no events are present in the event queue or a certain simulation time is reached. The simulation clock can not go backwards, therefore events can not be scheduled before the current simulation time.

Like other simulation approaches, such as model solving, DES has some distinct advantages when it comes to simulating software systems [BCNN10]. Most notably is, that simulations are cheap. Running multiple system configurations through simulations is way more cost-efficient than repeatedly configuring and deploying a whole system. "What-if" scenarios can also be answered quickly without disrupting a production or development system. Further, the advancement of the simulation time can be manipulated to observe the systems behavior in slow motion.

Unfortunately, creating a simulation model can be challenging. It requires in-detail information about the system and a not negligible amount of time. Similarly simulation results can be difficult to interpret, since there might be faults in the model or unexpected randomness. Banks et al. [BCNN10] go further into details about these advantages and disadvantages and argue when DES is applicable. However, as previously established in the beginning of this section, this is not a concern within this thesis since DES is well applicable to software systems.

3 Related Work

The following sections present research topics and projects that relate closely to this work. First, Section 3.1 gives an overview of simulators that are used for distributed systems. Then, Section 3.2 takes a look at software scenarios and chaos engineering. Section 3.3 presents research projects that relate closely to this project.

3.1 Simulation of Microservice and Cloud Architectures

The simulation of cloud environments and distributed systems has become a public topic among researchers. Within the recent years many simulators have been developed in this domain (e.g. see [Cis21; CRRB09; GVGB17; KBK12; MWW12; NGN17; Pac18]). Simulators that concentrate on simulating microservice architectures are also very present and mostly do have their own specialized focus. For example, *µqsim* [ZGD19] focuses on the simulation of microservice interactions and message traces. *DRACeo* [VDR+20] approaches a deployment and scaling analysis. Otherwise, *MiSim* [BZG17] concentrates on simulating software resilience failures. However, none of these simulators are designed to fully support scenario-based chaos experiments. Specifically chaos toolkit-like faultloads, such as delays or workload injections are usually not simulated by any simulator. However, in this thesis *MiSim* will be re-engineered to better support such features and allow for a stable simulation of scenario-based chaos tests.

3.2 Scenario-based Resilience Testing

As explained in Section 2.3, Scenarios are a way to describe expected system behavior under certain conditions. The inaugural work presenting scenarios was "Scenario-Based Analysis of Software Architecture" by Kazman et al. Later, the practices around the usage of scenarios were structured and improved [BCK03; KKC00]. In this paper, I focus on the usage of resilience scenarios, which are scenarios that concentrate on resilience metrics. One possibility to explicitly let such scenarios occur in a system are chaos experiments.

In our preceding industrial case study [KWK+20] we showed, that using resilience scenarios to create chaos test is a feasible way to verify a systems' resilience. When comparing the structure of scenarios and chaos tests, similarities can be observed. For example the *hypothesis* of a chaos test and the *response measure* both describe the expected behavior of the system with business metrics. Also, a combination of Stimulus, Environment and Artifact describe what load the system is expecting during a scenario. In the case of chaos experiments this directly done by the defined system disruptions. Additionally, using scenario based architecture analysis [KKC00] can help with the definition of the *steady-state* of a system. Each environment is linked to at least one metric.

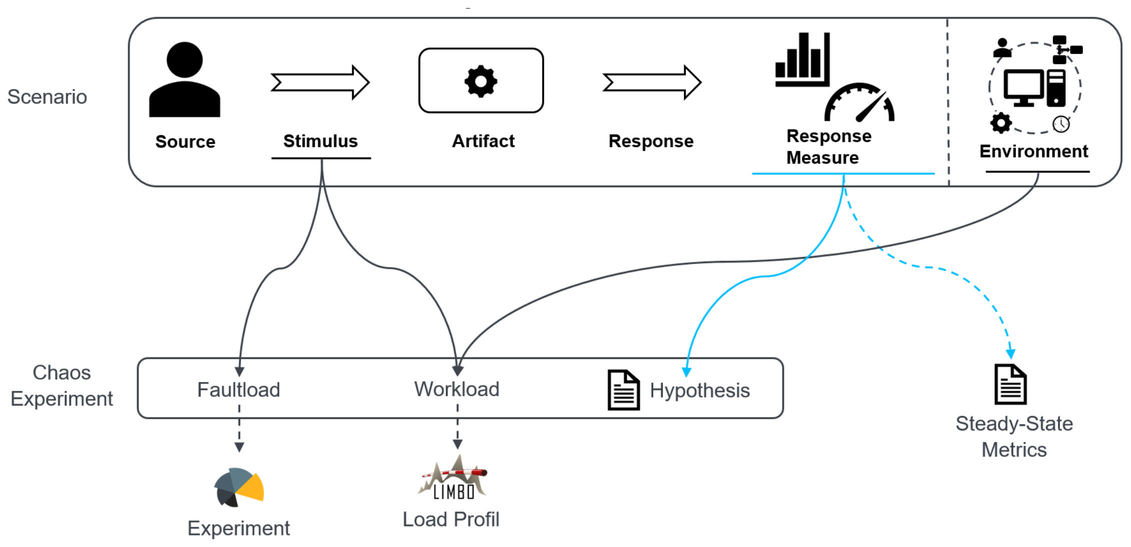

Figure 3.1: General process of transposing resilience scenarios to chaos tests. In the case of Kesim et al. [KWK+20], the different loads were then designed as CTK experiments and LIMBO [KHK14a] load profiles.

Based on this, combining the knowledge of the different possible environments with their respective response measure metrics can provide a good overview over which metrics are actually important in the system [KWK+20]. The general process of transposing resilience scenarios to chaos tests is also visualized in Figure 3.1.

Since the scenarios that were elicited during the case study are realistic, they offer the opportunity to be reused in the evaluation part of this thesis.

3.3 Preceding and Ongoing Research Projects

There are four other preceding and ongoing projects that also revolve around similar topics as this thesis.

First of, Beck et al. [BZG17] created the simulator *MiSim*, which is extended in the context of this work. Since its creation, this simulator had another iteration to support a Hystrix circuit breaker [Bec18].

Parallel to this thesis, Zorn [Zor21] is working on a project to aid with the extraction of scenario specifications from architecture descriptions.

Additionally, a students research project [HB21] is looking into creating structured and formal scenario descriptions for common incidents using the ATAM.

In general the results of this thesis will be integrated in the Cambio-Project[1].

[1] https://github.com/Cambio-Project

4 Requirements Analysis and Simulator Evaluation

To find simulators that are suitable for the approach of simulating scenario-based chaos experiments a list of requirements has to be established. Section 4.1 describes how these requirements were elicited and Section 4.2 presents the results of the requirements analysis.

4.1 Requirements Elicitation

There are many techniques to establish the requirements for a software system, e.g. prototyping, interviews or experiments [LL13]. All techniques result in the analysis of a specific state of the system (i.e. current state or target state). Since there is no prioritized simulator it was decided to create a requirements specification for the target state and evaluate existing simulators based on this.

To establish the requirements a group of three senior researches, that are experts in the field of scenario-based software resilience analysis, held a brainstorming session. This type of session was chosen, since they already had an understanding of how the requirements should look like from previous experiences. The session resulted in a list of 12 requirements. Over the course of the following weeks and during the search for existing simulators these requirements were slightly updated and finalized. To ensure the correctness the stakeholders were closely involved in this process. The actual results will be presented in the following section.

4.2 List of Requirements

The following list describes the requirements that resulted from the requirements elicitation session described in Section 4.1. The participants came to the conclusion, that the simulator ...

R1	must use discrete event simulation.
R2	must provide a headless mode for automation purposes.
R3	must be a lightweight solution (e.g. a single executable with easy to install dependencies).
R4	must enable parallel running of multiple simulations.
R5	must provide the following output metrics:
R5.1	Response Times
R5.2	Error Rates
R5.3	Throughput
R5.4	Queue lengths
R5.5	Execution/Message Traces

R6 must provide the metrics in a raw and unprocessed form (e.g. text based output).
R7 must support an architectural description that is used and supported by other tools.
R8 must support the faultloads and injections described by the case study scenarios.
R9 must support ChaosToolKit-like faultloads and injections.
R10 must support LIMBO-based workload descriptions.
R11 must support the simulation of the following techniques:
  R11.1 Microservice Instance auto-Restarting
  R11.2 Auto Scaling
  R11.3 Load Balancing
  R11.4 the following Resilience Patterns
    - Retry
    - Circuit Breaker
    - Rate Limiter
    - Caching

R12 should provide compatibility with the following tools:
  R12.1 Cambio-Project Scenario Description[1]
  R12.2 Resirio [Zor21]
  R12.3 TransVis [Bec21]

In the context of this thesis the simulator must also be able to simulate the following evaluation subjects:

- Scenarios of the preceding case study [KWK+20]
- Benchmark systems like TeaStore [KES+18] or TrainTicket [ZPX+18] (see also [RPT19])

4.3 Evaluating Microservice Simulators

First Section 4.3.1 describes the processes of how simulator candidates were found and selected. Then, Section 4.3.2 to Section 4.3.7 present the candidates in more detail. Section 4.3.8 presents how good these simulators fulfill the requirements, that were established in Section 4.2. Concluding, Section 4.3.9 explains which simulators lean towards which type of scenario and why *MiSim* was considered most suitable to support scenario-based chaos experiments.

4.3.1 Selecting the Simulators

A structured search was conducted to find microservice simulators that potential fulfill the elicited requirements. Most notably, using Google Scholar the search term "Simulator AND (Microservice OR Microservices)" lead to the finding of *DRACeo* [VDR+20], *$\mu qsim$* [ZGD19] and *MiSim* [Bec18]. The latter two, where also the most relevant results when using the same search term in CORE[2]. However, in CORE *DRACeo* did not appear within the first 50 most relevant results. Using the same search term in a normal Google search reveals two further candidates: *MuSim* [Flo] and an

[1]https://github.com/Cambio-Project/ScenarioDescriptor
[2]https://core.ac.uk/

unnamed simulator [Kur]. Both of these are quite similar, using Docker[3] containers to simulate a microservice system. However, the unnamed simulator does seem to be rather unfinished at the point of writing. It does only contain some Groovy code and incomplete documentation files. Therefore, this simulator was not considered during this work. Other related search terms like “Microservice Simulation” or “simulator for distributed systems” resulted into similar results or no relevant new findings.

Lastly, since the University of Stuttgart is closely related to the Palladio framework[4], I was also aware of the simulators SimuLizar [BBM13] and Slingshot [KB20]. Since, these simulators are also suitable to simulate a distributed system, they were also taken into consideration.

4.3.2 MiSim

MiSim [BZG17] is a microservice simulator that was developed by a team of students of the University of Stuttgart and targeted a simulation of microservices. Specifically, this tool was created to support the simulation of resilience mechanisms and chaos injections as part of the ORCAS project [HADP18]. In its original version 1.0 it had a rudimentary implementation of the circuit breaker pattern and the ability to simulate chaos-monkeys [BZG17]. Later, the circuit breaker implementation got reworked to resemble a Hystrix circuit breaker [JGC17b] for version 2.0 by Beck [Bec18]. Beck also provides an architecture extraction tool which can create an architecture input model for the simulator based on Jaeger[5] or Zipkin[6] traces.

MiSim is written in Java and contained within a single *.jar* file. It utilizes DES, based upon the DESMO-J[PL99] framework. A simulation requires two JSON files. The first one is an architecture description and the second one an experiment definition. Due to its lightweight nature the number of parallel simulations is not restricted by the simulator.

On the side of recorded metrics *MiSim* provides the raw output of its simulated response times, instance counts, CPU utilization and circuit breaker statistics.

Unfortunately, all resources are only simulated based on a single capacity value of a microservice, therefore they cannot be accurately described or simulated. Further network delay is not considered during the simulation.

4.3.3 SimuLizar

SimuLizar is the main simulator of the Palladio Component Model (PCM) [Sei21]. It was first presented in 2013 and is used to analyze the behavior of adaptive system models [BBM13]. It uses a model-driven DES and supports a variety of adaptation techniques, such as load balancing or scaling. As a part of the PCM, system models require five stages of modeling, reaching from basic component modeling over allocation assignments to usage cases. This makes these models

[3]https://www.docker.com/
[4]https://www.palladio-simulator.com/
[5]https://www.jaegertracing.io/
[6]https://www.zipkin.io/

comparably complex but also very detailed, allowing for a accurate simulation. Workload scenarios are a part of usage definition of a model [BLB13] and can be defined as open or closed workloads [LE15] and as LIMBO models [BBC+17; KHK14a].

Both the PCM and *SimuLizar* are extensions of the eclipse platform[7] or IDE. Therefore, they require this platform to execute simulations. Although, steps have been made towards the automation of this process [Mer11; RSW+20], a respective add-on is still in the incubation state [Str15].

SimuLizar is well documented, has a elaborate suite of tutorials [BBS] and is extendable by a hand full of other add-ons [Str15].

4.3.4 Slingshot

Slingshot [KB20] is another simulator for the PCM. Its aim is to create a simulator with a very extensible architecture using a DES. Like *SimuLizar* it has the property of heavily relying on the Eclipse platform and can utilize very detailed system models. At the point of writing, *Slingshot* is still in the development phase and only supports core system behaviors. Due to its early stage, there is also little to no documentation and its architecture is still potentially a subject to change.

Slingshot shows potential, but in its current state is not capable of simulating chaos scenarios. Hence, it was not considered further and does not appear in Table 4.1.

4.3.5 μqSim

μqsim [ZGD19] is a discrete-event simulator that focuses strongly on the communication between microservices. As such it offers sophisticated tools to define the properties of a microservice deployment, inter-microservice connections and request paths. Further, it can distinguish between physical servers allowing for detailed deployment models. *μqsim* supports basic linear and exponentially growing load functions. When it comes to output statistics a secondary python script is required to collect the simulators output stream. *μqsim* collects response time latencies and scheduling distribution stats. Except for load balancing, it does not consider any resilience mechanisms of an architecture.

Like, *MiSim* it compiles into a single executable and uses JSON-based inputs. A single simulation requires five files that have varying degrees of complexity and readability. Unfortunately, there is no official tool that supports the generation of architectures. During their validation of *μqsim* Zhang et al. [ZGD19] used hard coded Python scripts to generate the testing architectures.

Lastly, there is a lack of proper documentation. Neither does a manual exist, nor is the source code sufficiently documented. Occasionally, there are single line comments that circumscribe small sections of the code, but large parts are undocumented. Additionally, there are a lot commented out regions and unused “printf() debugging” statements.

[7] https://www.eclipse.org/

4.3.6 MuSim

MuSim [Flo] is a tool developed by Florio as part of this PhD thesis [Flo17] to observe the communication between microservices. There is no publication about the simulator itself. It does not rely on discrete-event simulation or model solving. Rather, *MuSim* provides a generic implementation of a service that can be deployed e.g. as a Docker container. Each instance of the service can be configured to have dependencies to other instances and a certain amount of workload. Workloads are actual synthetic mathematical calculations. *MuSim* has an integrated service discovery that requires an etcd [8] server and can output its recorded metrics to an influxdb [9].

4.3.7 DRACeo

DRACeo [VDR+20] simulates complex microservice networks with a focus on QoS requirements. It is only available as desktop application with a graphical user interface (GUI).

Similar to *µqsim*, *DRACeo* can distinguish between physical devices. Each device can be assigned a concrete amount of CPU, RAM, hard drive speed, and battery size. This allows the simulation of small devices, such as phones or laptops. Devices can host multiple microservices and are connected via different types of connections (e.g. Ethernet or 4G) with varying connection speeds. Microservices consume a configurable amount of the resources of its hosting device. As *DRACeo* focuses on the analysis of the efficiency of dynamically deployed architectures it allows microservices to duplicate, move between devices, (re)start, stop, and die. All these activities can be automatically scheduled or manually triggered. Regarding the observation of QoS *DRACeo* calculates a QoS value based on the current resource usage of a microservice or device. Additionally, the energy consumption of all devices can be observed.

DRACeo currently only comes as a graphical application without a command line interface or server like function. Meaning, architecture configurations can only be loaded (or saved) via the GUI. Therefore, automation is currently not possible. Further, the simulator goes through an intellectual property registration processes which means, it is available at all at the time of writing.

4.3.8 Evaluation Results

Table 4.1 shows how the simulators (except for *Slingshot*) stack up against the requirements presented in Section 4.2. To achieve these results the simulators where analyzed on multiple levels. Most of the information is drawn form the respective research papers, or by looking through the source code and documentation. However, in the case of *DRACeo* and while gathering information about *Slingshot* I additionally contacted the authors to fill in missing gaps.

In general the simulators share commonalities between them, e.g. all simulators use DES with the exception of *MuSim*. Most of them can be considered ’lightweight’. This means that they are single executable and do not need additional dependencies or frameworks. When it comes to output metrics, response times are automatically collected by all simulators. However, no simulator

[8] `https://etcd.io/`

[9] `https://www.influxdata.com/products/influxdb/`

	SimuLizar	*DRACeo*	*MiSim*	*μqsim*	*MuSim*
R1 Uses DES	**Y**	**Y**	**Y**	**Y**	**N**
R2 Headless Mode	**N**	**N**	**Y**	**Y**	**N**
R3 Lightweight	**N**	**N**	**Y**	**Y**	**N**
R4 Parallel Runs	**N**	**N**	**Y**	**Y**	**Y**
R5 Output Metrics					
R5.1 Response Times	**Y**	**Y**	**Y**	**Y**	**Y**
R5.2 Error Rates	**N**	**N**	**N**	**N**	**N**
R5.3 Throughput	**Y**	**N**	**N**	**Y**	**N**
R5.4 Queue lengths	**Y**	**N**	**Y**	**Y**	**N**
R5.5 Execution Traces	**N**	**N**	**Y**	**N**	**N**
R6 Raw Output	**Y**	**Y**	**Y**	**N**	**N**
R7 Common Architecture Desc.	**Y**	**N**	~[2]	**N**	**Y**[3]
R8 Case Study System Loads	**N**	~[4]	~[4]	**N**	**N**
R9 CTK faultloads	**N**	~[4]	~[4]	**N**	**N**
R10 LIMBO support	**Y**[1]	**N**	**N**	**N**	**Y**[3]
R11 Resilience Features					
R11.1 Self-healing (restart)	**N**	**Y**	**N**	**N**	**Y**[3]
R11.2 Auto Scaling	**Y**[1]	**Y**	**N**	**N**	**Y**[3]
R11.3 Load Balancing	**Y**[1]	**Y**	**N**	**Y**	**Y**
R11.4 Retry	**Y**[1]	**N**	**N**	**N**	**N**
R11.5 Circuit Breaker	**N**	**N**	**Y**	**N**	**N**
R11.6 Rate Limiter	**Y**[1]	**N**	**Y**	**N**	**N**
R11.7 Caching	**Y**[1]	**N**	**N**	**N**	**N**
R12 Compatibility					
R12.1 Cambio Scenarios	**N**	**N**	**N**	**N**	**N**
R12.2 Resirio	**N**	**N**	~	**N**	**N**
R12.3 TransVis	**N**	**N**	~	**N**	**N**

[1] Supported as part of the Palladio Component Model (PCM).
[2] Architectural description is supported by tools and extractable from Jaeger and Zipkin traces.
[3] Supported due to the usage of a PaaS.
[4] Supports instance/service/device killing.

Table 4.1: Results of the evaluation of existing simulators with respect to the requirements specification.

Simulator	Strength	Weakness
SimuLizar	Detailed system descriptions and usage profiles.	Requires heavyweight Eclipse platform
DRACeo	Detailed system descriptions, advanced deployment manipulation techniques	Currently not available for further examination. Only uses a GUI.
MiSim	Simulation of resilience patterns and CTK faultloads.	Architecture description is more on the abstract side.
μqsim	Complex system descriptions, accurate tail latency predictions	Only static architectures, bad Documentation. Unusual report system.
MuSim	Represents an actual system.	Not a DES-simulator.

Table 4.2: Overview of the advantages and disadvantages of the simulators.

supports the output of error rates. Service and instances failures can be simulated by two of the simulators (i.e. *DRACeo* and *MiSim*). Otherwise, none of the other simulators has support for faultloads. Resilience features are present in all simulators. *SimuLizar* supports by far the most (5), then *DRACeo* (3) and *MiSim* (2). Both *μqsim* and *MuSim* support only load balancing by themselves. Only *MiSim* does have some compatibility with our existing tools, since they use similar architecture descriptions.

4.3.9 Scenario Simulation Suitability

Looking at the evaluation results from Section 4.3.8 it becomes clear that none of the simulators completely fulfill the requirements set in Section 4.2. However, each of the simulators has its weaknesses and strengths, therefore leaning towards specific types of scenarios. These strengths and weaknesses are presented in Table 4.2.

Going through this list, both *SimuLizar* and *DRACeo* allow for detailed system models. It is currently not clear, what loads *DRACeo* supports, but the PCM (and therefore *SimuLizar*) can apply complex load models, e.g. LIMBO model [KHK14a]. Unfortunately, both these simulators require a GUI or other frameworks to run. Additionally, for now, *DRACeo* simulations can not be automated. *MiSim* is specifically designed to simulate CTK-like faultloads. Also it is very lightweight. On the other side, architecture descriptions of *MiSim* are not as detailed as the ones used by other simulators. Similar to *DRACeo* and *SimuLizar*, *μqsim* allows for more complex system descriptions and accurately predicts tail latencies. Unlike the previously presented simulators, it is not able to change the architecture of the system during a simulation run. Lastly, *MuSim* is a non-DES-based simulator. This means, that the simulation of a scenario involves more work, since a whole actual system has to be set up. However, simulation results can be more accurate, as natural effects that might not be modeled in DES simulators (such as operating-level scheduling) actually impact the simulation.

From these strengths, weaknesses and the other previously in Section 4.3 introduced properties of the simulators, the resilience scenario types that are suitable for the respective simulator can be derived. As a compact overview these are listed in Table 4.3.

Simulator	Suitable Resilience Scenario Types
SimuLizar	Scenarios with complex load behavior or self-adaption.
DRACeo	Scaling and Deployment Scenarios
MiSim	Failure Scenarios
μqsim	Load balancing and path-based routing
MuSim	Prototyping and behavior checks of a given PaaS.

Table 4.3: Suitability of Scenario types per Simulator

First off, *SimuLizar* obviously is currently the best choice of simulator, when a systems' development process is already revolving around the PCM. It can simulate many resilience features and has many plugins for other kinds of simulations. Since it does not support failure loads, it is most suited to run scenarios with complex load behavior or self-adaption scenarios. Modeling an existing large system for the PCM can be quite complex, due to the many involved architectural views. There are approaches that could automate architecture extraction for PCM (e.g. [BHK11; LBG+15; VHK15]), but the official documentation site on this topic is deprecated and was last updated in early 2016 [Lan].

Similarly to *SimuLizar*, *DRACeo* would be used best to simulate self-adaption or deployment scenarios. It was developed specifically for this purpose. But, since it also supports the killing of devices and services it can also be used for chaos experiments. However, as the simulator does not support resilience patterns (such as circuit breakers), systems that use these patterns for resilience cannot be simulated accurately.

MiSim is the only simulator that supports both, faultloads and resilience patterns. Therefore, resilience scenarios in which parts of the software system fail and others need to cope with them, are most suitable for *MiSim*. However, in its current state of Version 2.0 it does only use simplistic load profiles.

μqsim does support static architectures, load balancing and detailed path definitions. Hence, it is best used to analyze load balancing scenarios. One property that stands out about *μqsim* is that it allows the system to be loaded with huge workloads (e.g. > 50 kQps), which might not be possible with other simulators.

Lastly, *MuSim* represents a different simulation approach. As such it can be used to build an actual system prototype and test out the systems' behavior as well as its respective PaaS.

The choice of which simulator is suitable for an extension was quickly narrowed down. *MuSim* is not a DES simulator and very reliant on specific technologies (i.e. Docker and etcd). Therefore, it is not flexible and relatively slow to set up. It does not bring tools to hence it was not considered further. *Slingshot* is also not relevant, since it is currently in an unfinished state. *DRACeo* is also not available at the time of writing, since it goes through an intellectual property validation processes. *SimuLizar* requires an Eclipse execution environment, therefore cannot be considered as lightweight as the requirements request. Lastly, the choice between *μqsim* and *MiSim* is determined by the existence of resilience and faultload mechanisms within *MiSim*. Since the simulator extension is required to add further resilience and chaos mechanisms, the choice naturally falls in favor of *MiSim*.

5 Improvement of the *MiSim* Simulator

As presented in Section 4.3.2 and Table 4.1, the *MiSim* simulator in Version 2.0 already provides basic features to simulate chaos experiments, but lacks the ability to represent specific properties of microservice systems such as retry patterns, default network latency or service scaling. Further, a closer look at its source code reveals that it is partial poorly structured. Specifically, it is missing extensibility, separation of concerns and has some redundant code. In the context of this thesis the *MiSim* simulator is re-engineered to Version 3.0, improving the aforementioned downsides and adding new features. Section 5.1 presents the architecture *MiSim* 2.0 and the critic on it detail. Section 5.2 follows up with an explanation of the new simulation structure and how it will improve on the current state of the simulator.

5.1 Architecture Evaluation

This section goes into detail about the structure of the architecture of *MiSim* 2.0 in Section 5.1.1 and the downsides of it in Section 5.1.2.

5.1.1 The Architecture of *MiSim* 2.0

The original architecture of *MiSim* 1.0 and 2.0 is focused on a single class, i.e. the `StartEvent` class. It takes care of most of the feature interactions of the simulator. Specifically, it assigns requests to services, creates dependencies, chooses which CPU should handle a request and also watches over the different resilience patterns. Since this is all done inside the activation method of the `StartEvent`, there are many possible execution paths and up to six layers of code nesting. As a whole the part of the architecture that is responsible for the simulation of a system can be best described by a sequence diagram, since most of its code lies within a couple of classes. The sequence diagram displayed in Figure 5.1 is extracted from the source code of *MiSim* 2.0 and follows the data and control flow around a single exemplary StartEvent.

Before the actual simulation starts, the `MainModel` instantiates all objects. Then once the simulation has started, a `Generator` creates a `StartEvent`. It is scheduled immediately by the simulation. The event first collects details, mainly from the `MainModel` object and partially from the target `Microservice` object. It then checks whether the circuit that relates to the current request is open. If yes, it is noted in the circuit breaker statistics and triggers a `StopEvent`. This finalizes the request as if it would be completed successfully. If the circuit was closed, the `StartEvent` has to differentiate between two cases. In the first case, the request has no dependencies and is submitted directly to its handling `CPU` as a `Thread` object. In the second case, when there are dependencies, the `StartEvent` creates a `DependencyNode` object for each dependency, containing the relationship information.

Then, it schedules a new `StartEvent` for each dependency, which represent the requesting for dependencies. Each dependency can have a probability value that determine how likely it is required for the request to complete. All dependencies are collected in parallel. Once a request is submitted to a `CPU` object as a thread it is fixed quantum round-robin scheduled until its completion. In the interest of readability, this is shown only very simplistically in Figure 5.1. Upon completion a `StopEvent` is created and immediate scheduled. This event then checks the `DependencyNode` object of the respective request and if its parent does not have any uncompleted dependencies it is submitted to its original target `CPU`. Otherwise, the `StopEvent` collects the statistics for the current request.

When it comes to the general execution path of the *MiSim* program, the `MainModel` class needs to be analyzed. It is the entry point for the program. As such it parses the command line arguments, system architecture and experiment structure. Both, the architecture and the experiment information are loaded by a parser object respectively. These parser objects fill some `static` fields with the extracted data and then get discarded immediately. Some static data then gets copied over into the `MainModel`. The actual starting of the experiment, report collection and the initializing of the simulated objects, like `Microservices` or `Generators` is also done within the `MainModel`.

5.1.2 Critic on *MiSim* 2.0

The architecture of *MiSim* 2.0 does have a shortcoming when it comes to the separation of concerns. As mentioned in Section 5.1.1 there are at best three classes that take care of all main simulation steps with very nested code, i.e. `StartEvent`, `StopEvent` and `CPU`. All of these can be considered God objects that utilize lasagna code, which are known software anti-patterns [TI14]. Most of the other existing classes are only data classes without any specific functionality. They only encapsulate raw data, rather than actually working with their data and modifying it on their own. In the interest of maintainability and extensibility it does make sense to split these classes into multiple ones and to extract reappearing patterns. Further, since a lot of different strategies can be applied when its comes to microserivce architectures, it would make sense to utilize the strategy architectural pattern.

This point of critic also carries over to the `MainModel`. It also takes care of responsibilities that could be divided up into multiple classes. Further, the `MainModel` and input parser provide static data that is created by the initiation of an object. This relation does not make sense, since the purpose of `static` data and objects is to be mostly independent of other objects. To solve this problem a singleton pattern could be utilized. These singletons would then encapsulate the static data that is currently public within the parsers.

Due to the god object property and high cohesion of the existing classes it would be hard to add further functionalities without further worsening the existing problems. Therefore, extensibility of *MiSim* 2.0 can be considered bad.

When it comes to simulation features there is one missing core feature. This feature is the simulation of network delay. Network communication is one of the key aspects to consider when deciding for a microservice architecture. As microservices are very loosely coupled and only fulfill a single business function, they usually communicate heavily back and forth. Therefore, the impact of communication needs to be considered when evaluating the performance of a microservice architecture. Further, the target of the simulator is to investigate the resilience of a microservice system. Many

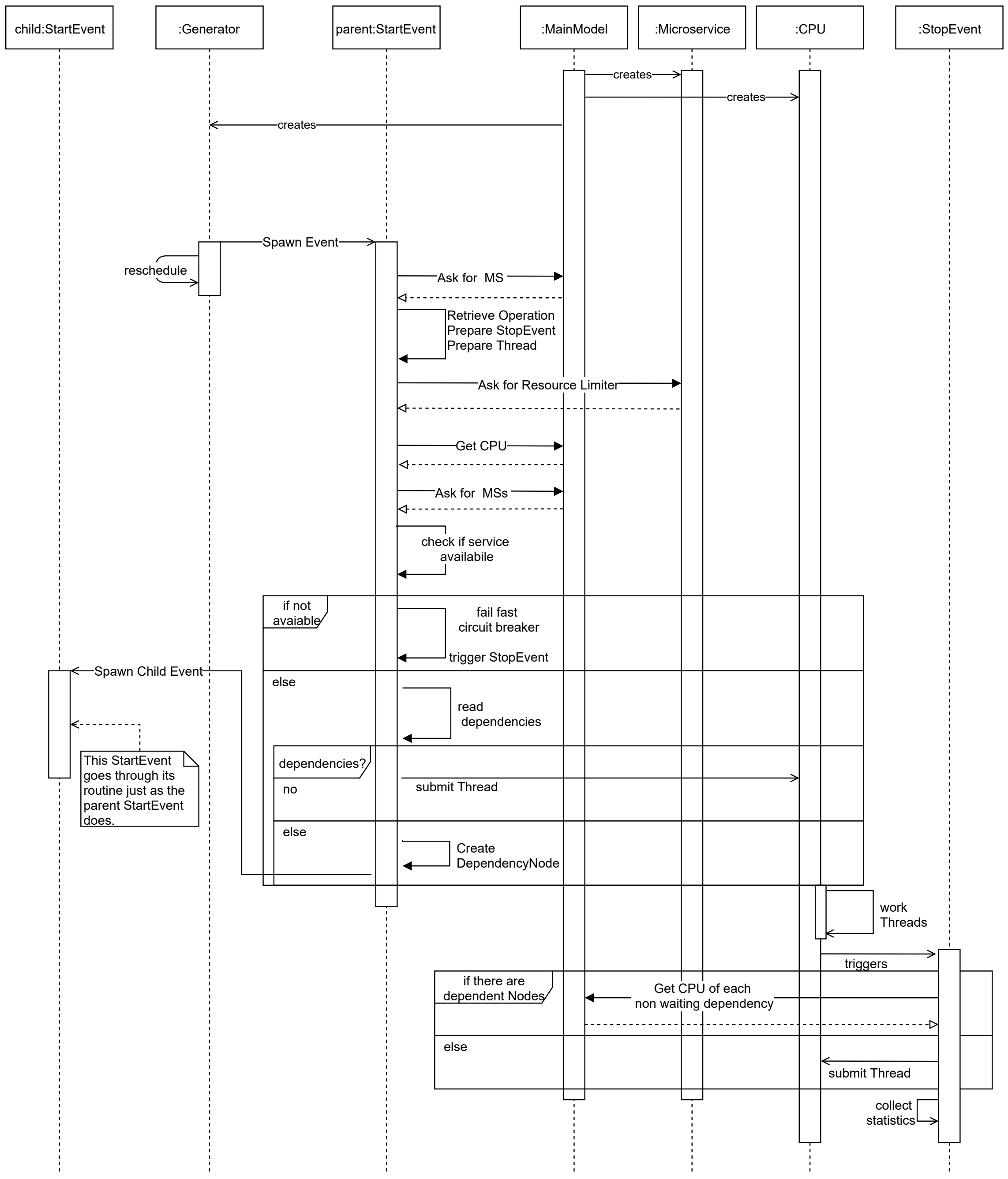

Figure 5.1: Data and control flow of *MiSim* 2.0

patterns that try to increase the resilience are based on monitoring network communication, e.g. circuit breaker or retry [Lup17]. An accurate simulation of these should only be possible in the context of an accurate network simulation.

Following that, there is another point to be made regarding the same type of patterns. The simulator does not distinguish between failed and successful messages. However, some of such resilience patterns need this information to work properly. Additionally, this would be relevant when evaluating the resilience of a system via success rates. Lastly, I want to touch on the `CPU` implementation. It uses a round-robin scheduler with a hard-coded fixed quantum. This type of scheduling does not try to optimize turnaround time, response time, context switching or the average waiting time of processes [SGG05]. More importantly, since the operation demands and service capacities can differ by up to 15 orders of magnitude in between simulations, a fixed quantum is not suitable for every architecture.

5.2 The Architecture of *MiSim* 3.0

The inspection of the original architecture in Sections 5.1.1 and 5.1.2 revealed that it lacks a separation of concerns and extensibility. Further, some used concepts are not applicable to the use case of the simulator and a lot of the code needs refactoring. Therefore, a re-engineering of the code took place, to allow a feature richer simulation and a easier extension in the future. In general, a more object orientated approach was chosen. This decision was based on the clear responsibility structure that a simulation of microservice architecture provides. To name a few: user request generators should be sending and monitoring messages, microservices should handle and inform about their instances and microservice instances should be able to decide themselves, how to handle incoming requests. Further, requests should have an actual sending process, that can be interrupted, timeout or fail otherwise. The later provides a clear entry point for all network related resilience mechanics or fault loads.

In part of this thesis, Section 5.2.1 gives a general view over the life cycle of a request and how the central classes interact. Then Sections 5.2.2 and 5.2.3 explain how the requests can move through the system during the simulation. Section 5.2.4 goes into detail about the supported resilience pattern and how they were implemented. Lastly, Section 5.2.5 lists some minor improvements that developed naturally during the modifications.

5.2.1 General Architecture Overview

Figure 5.2 shows the new object orientated structure of the simulator and an overview of the responsibilities of the most important classes and components. Two subprocess nodes are used to simplify this graph. Those are not actual components of the architecture. The life cycle of requests, as described by the red path, take place as the following process.

After a request is created by a Generator, a `RequestSendingProcess` is started. This process first asks the respective target `Microservice` object for an instance that should handle the request. The `Microservice` object then utilizes its `LoadBalancer` to find a suitable target instance. Once an instance is picked, the `RequestSendingProcess` submits the request to it. Based on the configured network latency of an operation the arrival at the instance will be appropriately delayed. The instance

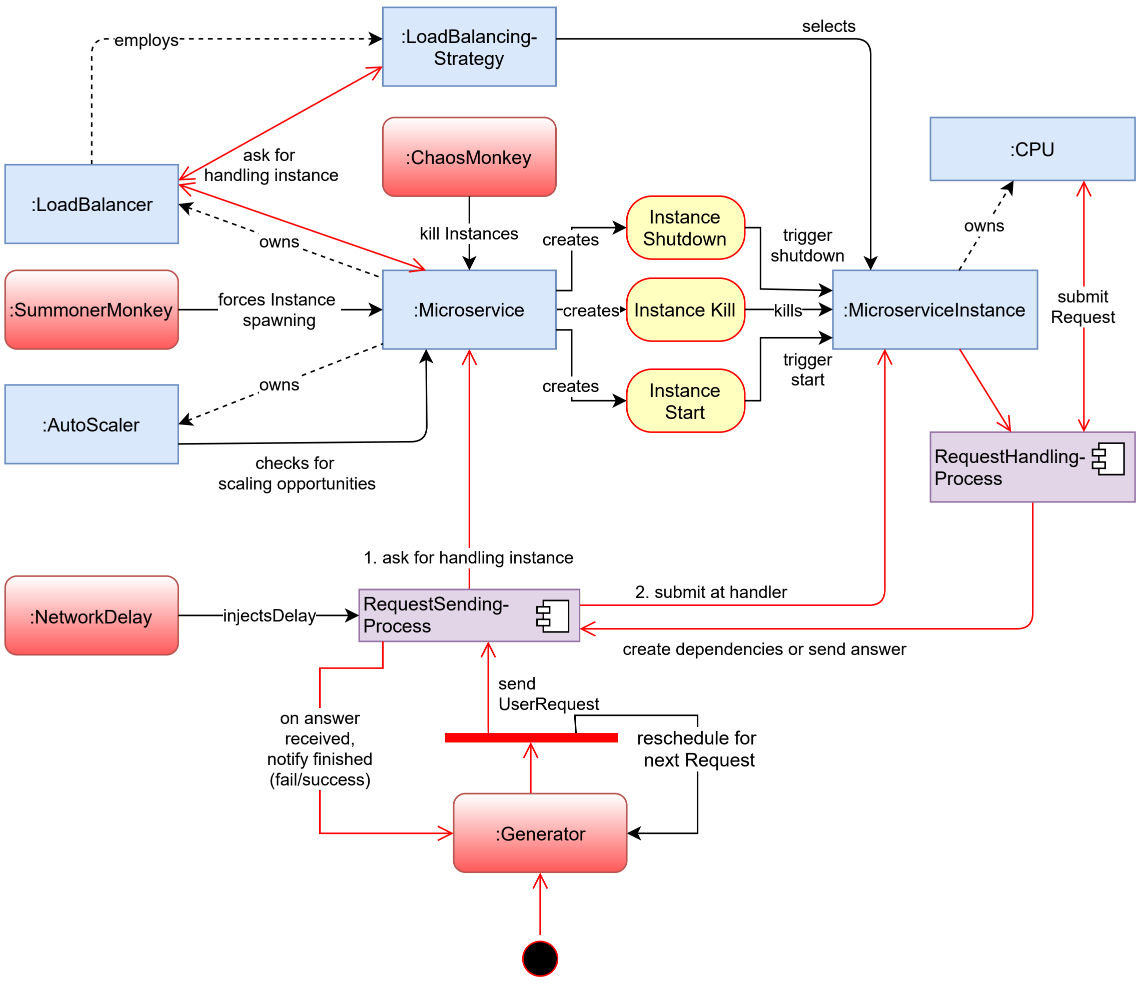

Figure 5.2: General structure and responsibilities of *MiSim* 3.0

then goes about executing its `RequestHandlingProcess`, i.e. creating and sending child requests to complete dependencies or scheduling the requests' process on its `CPU`. Once all dependencies of the root request are collected and it is processed, an answer gets send back to the `Generator`.

When looking at the architecture with a more static view, the `Microservice` class acts as a central reference point. It can be used to manage `MicroserviceInstances` and to look up their current status. Each `Microserivce` owns a `LoadBalancer` that takes care of instance selection by employing a `LoadBalancingStrategy` via a strategy pattern. This ensures flexibility and extensibility [Gea02]. The `AutoScaler` and `CPU` are structured similarly. Experiment events like `ChaosMonkeys` or `NetworkDelays` can interact with a `Microserivce` object to modify the underlying instances.

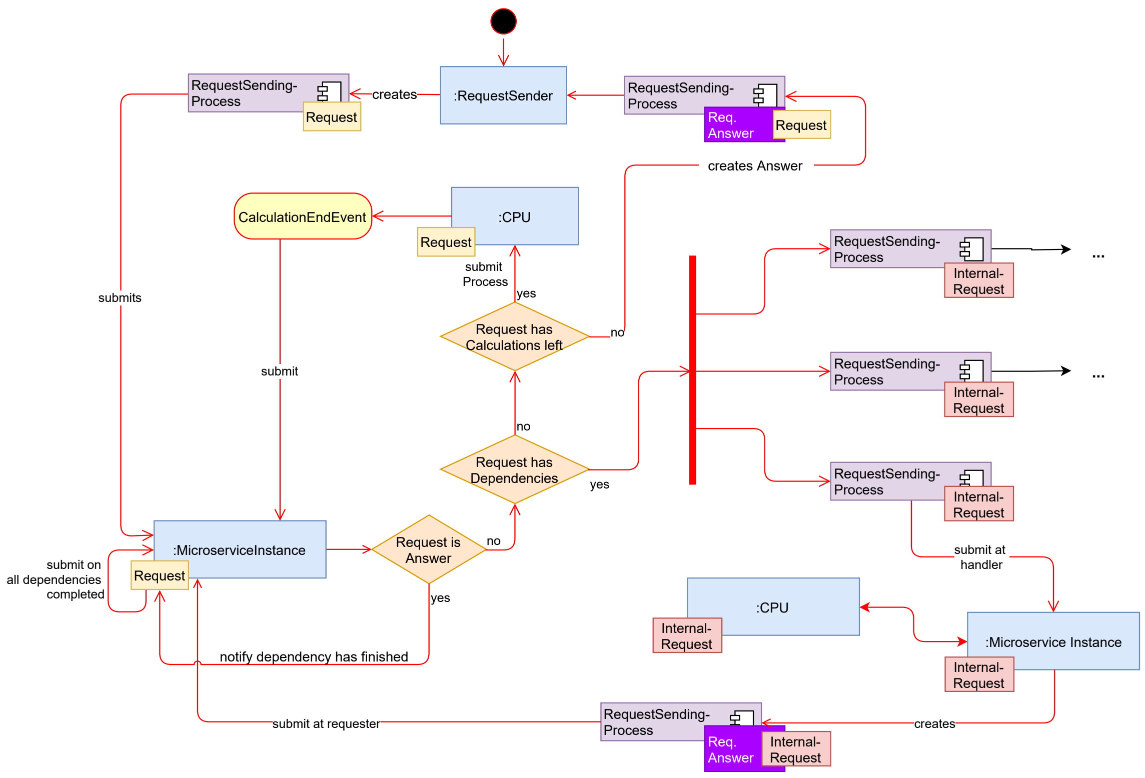

Figure 5.3: Visualization of the `RequestHandlingProcess`

5.2.2 Request Handling Process

The `RequestHandlingProcess` describes the algorithm executed by a `MicroserviceInstance` upon the arrival of a request. It is visualized in Figure 5.3. In this graph components are annotated with the request they are handling or transporting to increase comprehensibility. Further, the graph omits the handling of timeouts or canceled messages in the interest of readability. Before going into detail about the process it is necessary to establish the three currently existing types of requests.

- `Request` is the base type for all requests. It is an abstract class that represents an entity that can be send and monitored during a `RequestSendingProcess` and handled by a `RequestHandlingProcess`. All other request types extend this class.
- `UserRequests` represent root requests that are directly send by the user. In the simulation all `Generators` exclusively create `UserRequests`. They implicitly do not have a parent `Request` object.
- `InternalRequests` represent the requests that are created to satisfy the dependencies of other `Requests`. They contain additional information for which dependency they were created.
- `RequestAnswers` are a wrapper for a `Request` to represent the sending of an answer. They exist to simulate the actual sending process of an answer with an object orientated approach.

A `RequestHandlingProcess` always starts off with a `RequestSender` (i.e. a `Generator` or `MicroserviceInstance`) creating a `Request` object and sending it via a `RequestSendingProcess` to a `MicroserviceInstance`. Upon receiving this root request, the `MicroserviceInstance` can handle it in four different ways, depending on the requests' type and status. For this the following rules apply.

First, if the request is a `RequestAnswer` then it will be unwrapped and the dependency of its parent request will be marked as completed. If in doing so all dependencies of a request are completed, it will be resubmitted to its handling instance.

Secondly, if the request has uncompleted dependencies, `InternalRequests` will be created and send to the respective microserivces to handle. In the interest of readability, Figure 5.3 shows the handling process of `InternalRequests` only simplified.

Thirdly, if all dependencies of the request are complete, but it has open calculations left, it will be submitted to the `CPU`. The `CPU` then simulates the computation time of a request and fires a `CalculationEndEvent` upon completion. This event resubmits the request at its handling instance.

Lastly, in any other case the request is considered completed. Therefore, the handling instance wraps the original `Request` in a `RequestAnswer` and sends it back to the `RequestSender`.

The structure of this handling process enforces the interaction with a request during all its possible states. Combined with the concept of resubmitting the request at every state, this allows for an easy extension or modification of this process, by adding further states or types of `Requests`. Regarding the separation of concerns, each `MicroserviceInstance` object now has full control of how it can handle the request, which was not the case in *MiSim* 2.0.

5.2.3 Request Sending Process

The simulated sending process is described in Figure 5.4. Like in the `RequestHandlingProcess` that is described in Section 5.2.2 a `RequestSender` object is the first to interact. Before the actual sending process starts, this sender can register `IRequestUpdateListeners`. These listeners will be notified about status updates of all send requests. As examples the `CircuitBreaker` and `RetryManger` pattern can be seen in the graph. This relation is explained in more detail in Figure 5.6.

When triggering the sending of a request, the `RequestSender` schedules an immediate `NetworkRequestSendEvent`. Upon execution of this event the following checks are done. First, if the request is a `RequestAnswer` wrapping a `UserRequest`, the later will be immediately considered completed and arrived at its target. Secondly, if the traveling request is a `UserRequest`, an immediate `NetworkRequestReceivedEvent` will be scheduled. This has the effect, that the request arrives instantly at its handling microservice instance without considering network latency. In both cases this is not simulated, because the simulator does not consider the latency between a user and the system. If none of the previous checks were true, the default network latency will be calculated based on the properties of the connection. A `NetworkDelay` injection can take effect here, adding extra delay to the connection. This delay can be a fixed value or a Gaussian distribution. Once the network latency is finalized, a `NetworkRequestReceiveEvent` is scheduled to be executed with the appropriate delay. `NetworkRequestReceiveEvents` submit the traveling requests at the target instance upon execution. If anything interrupts the sending process, e.g. no instance is available or the `RequestSender` dies, a `NetworkRequestCanceledEvent` is immediately scheduled. It represents the failure of that request.

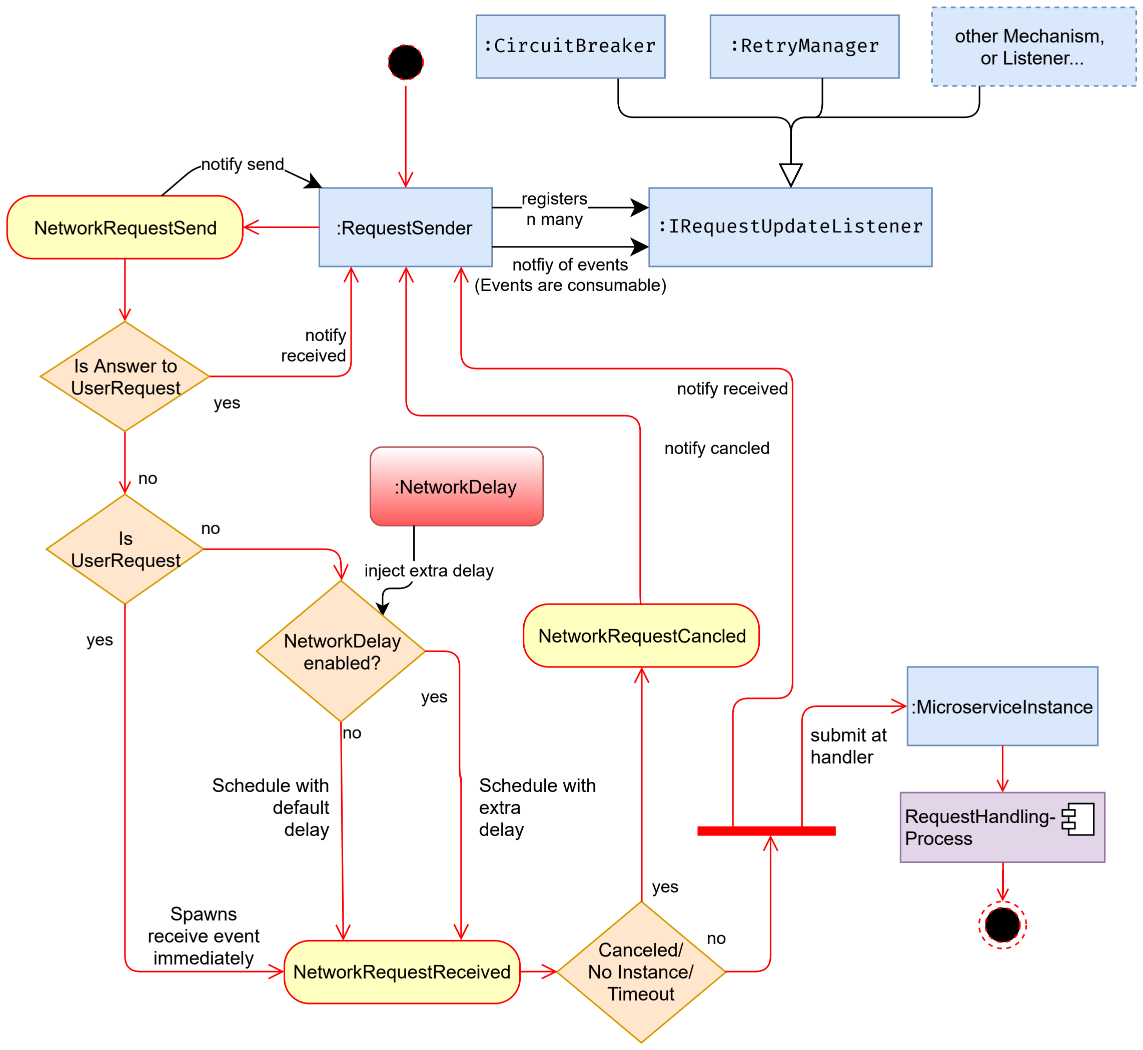

Figure 5.4: Visualization of the `RequestSendingProcess`

The `RequestSender` collects all updates and notifications about each of its send requests. Upon receiving an update it tries to notify all `IRequestUpdateListeners` that were previously registered. Each of these listeners can give itself a priority. Listeners with a higher priority will be notified first. However, these notifications are consumable. This means, that a Listener can prevent all subsequent Listeners from receiving an update. This is utilized for example in the relation between a circuit breaker and a retry pattern (s. Section 5.2.4). The code structure behind this interaction is also present in Figure 5.6.

5.2.4 Implemented Resilience Patterns

MiSim 3.0 supports the simulation of a hand full of resilience patterns, i.e. Circuit Breaker, Retry, Autoscaling and Load Balancing. Each pattern is configurable in the architecture description. Whilst a Hystrix-based Circuit Breaker implementation is already present [Bec18], the others had to be added during the expansion. The following subsections describe the aforementioned patterns and which thoughts and decisions went into their implementation.

Retry Implmentation

When using a retry pattern in *MiSim* 3.0, it simulates a full jittered exponential backoff implementation. Jittering can be turned off by demand. The other types of retries that were presented in Section 2.2.1 were also considered, but this choice was made based on the results of Brookers' simulations [Bro15]. They revealed that a "Full Jitter" implementation potentially performs better than an "Equal Jitter" and roughly equal to a "Decorrelated Jitter". Furthermore a "Full Jitter" implementation allows an easy fallback to a exponential backoff, in case no jittering is required. This is specifically the case for the scenarios used for the evaulation of the simulator in Chapter 6. On the architectural side, retries are currently attached on an instance level. This means, that all outgoing requests of an instance will be monitored by the retry pattern. All parameters of the pattern can be configured via the architecture definition.

Circuit Breaker Implementation

The circuit breaker implementation of *MiSim* 3.0 is Hystrix-based. Since *MiSim* currently does not simulate the content of messages, it does not implement the aforementioned caching. Furthermore, instead of limiting concurrent requests via a thread pool the circuit breaker can limit the amount of concurrent connection between an instance and a service. Fallback strategies are also not supported for now. Like the retry pattern, that was presented in Section 5.2.4, the circuit breaker is instance-owned. Each connection from an instance to a service is assigned a circuit breaker and monitored separately. The configuration of a circuit breaker is done via the architecture definition.

Relation between Retry and Circuit Breaker

Both the retry and the circuit breaker pattern are interfering with and monitoring network communications. Therefore, their interaction has to be defined. Two of the most popular resilience frameworks, i.e. Hystrix and resilience4J, embed the retry within the circuit breaker pattern [Igo20; Lup17]. This interaction is visualized by Figure 5.5. The most important property of this interaction is the ability of the retry pattern to conceal the failing of requests until the retry limit is reached. One disadvantage of this implementation is, that the circuit breaker does not immediately open upon a failure. Rather, all retries have to be exhausted first. Therefore, its fail-fast property is somewhat limited. Also, the implementation of the half-open state needs to compensate for a (potentially infinite) retry.

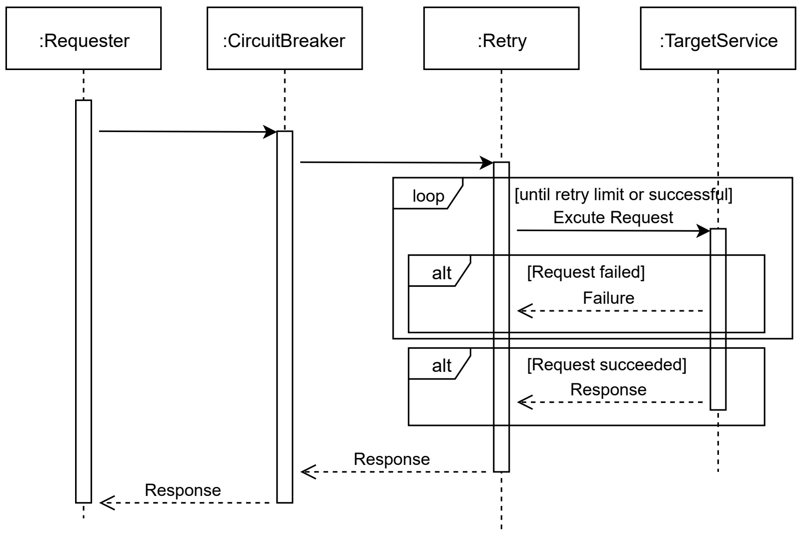

Figure 5.5: Relation between Retry and Circuit Breaker [Lup17].

In *MiSim* this interaction is realized with a consumable-event system. Once, a retry notices a failed request it consumes the failure event, if the retry threshold is not reached. Otherwise, the retry ignores the failure and the event falls through to the circuit breaker. This allows both patterns to be completely independent.

Loadbalancer

Section 2.2.3 presented the different styles of load balancers. *MiSim* 3.0's implementation comes closest to the service-oriented approach, that is explained by Niu et al. [NLL18]. Specifically, a target instance is chosen when the sending process starts. Neither does the client know of a server list, nor is there a central load balancing node. Rather, microservices act as meta entity that are able to choose the next receiving instance by themselves. Unlike a practical implementation of the service-orientated load balancer, there is no specific load balancing node that is part of the network.

The microservices can employ different load balancing algorithm each. *MiSim* 3.0 supports a randomized and a deterministic load balancing strategy. The randomized approach simulates a highest random weight equal-cost multi-path routing [Hop00]. Each time a request should be routed to an instance, one is picked at random from the available ones. The deterministic approach uses a QoS-orientated method, that always sends requests to the least busy instance. How busy an instance is, is directly determined by its current relative workload demand ρ. It represents the ratio between left over calculation demand and the processing capacity per simulation time unit (STU). This is implemented as the formula given in Equations (5.1) and (5.2). The `argmin` function is defined to return the first argument that produces a minimal value of the examined function. instance.#threads represents the number of threads of the current instance. A round-robin-based approach [Idz10] will be added in the future.

$$\rho_{instance} = \frac{(instance.\text{remainingActiveDemand} + instance.\text{remainingQueuedDemand})}{\text{instance.\#threads} * \text{instance.perThreadCapacity}} \tag{5.1}$$

$$\text{instance}_{\text{next}} = \underset{i \in \text{RunningInstances}}{\texttt{arg min}} (\rho_i) \tag{5.2}$$

Autoscaling

MiSim 3.0 implements a QoS and efficiency orientated horizontal scaling approach [NCCA14]. The autoscaler periodically checks all instances of a service for their current relative CPU utilization. This utilization value is calculated, as described in Equation (5.1) and can therefore exceed 100%, if the services experiences more load than it can handle in one STU In case the utilization of any CPU is above a certain threshold another instance will be spawned. Similarly, if it is below a set downscaling threshold the instance with the lowest current utilization will be asked to shut down. The autoscaler also prevents a service from downscaling within a set duration after the last upscaling-event to prevent premature downscaling during a workload ramp up. Currently, this implementation concentrates on horizontal scaling, but in the future it would also be possible to easily add vertical scaling techniques.

Extensible Class Structure of the Resilience Patterns

The class structure around the patterns was designed to support further implementations. Figure 5.6 shows an excerpt of this structure. The `Pattern` class is a base class that allows extending objects to fill their `@FromJSON`-annotated fields[1] with the configuration defined in the architecture description. This field initialization is automatically triggered once the `Pattern`-owing entity (i.e. a `Microservice` or `MicroserviceInstance` object) is created. `Networkpatterns` are a specific type of pattern, that monitor the network communication of a `MicroserviceInstance`. As such they will be automatically registered as `IRequestUpdateListeners` at their owning instance, as shown by Figure 5.6. This class structure ensure extensible, however, adding new patterns still requires a small code modification in the `PatternData` class to ensure the previously mentioned parsing automation.

5.2.5 Other Improvements

The focus of the modification of *MiSim* was mainly on improving its ability to simulate network behavior and further resilience patterns. However, additional general improvements were made, which will be presented in the following subsections.

[1] https://docs.oracle.com/javase/tutorial/java/annotations/index.html

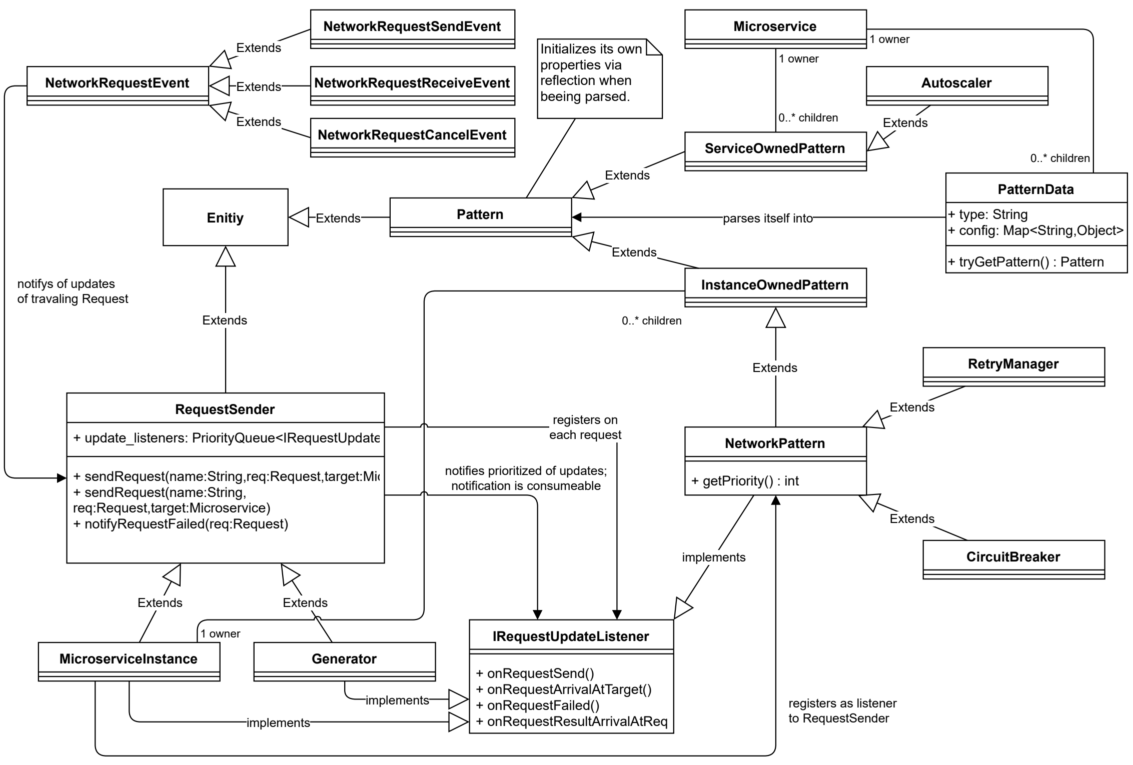
NetworkRequestSendEvent
NetworkRequestReceiveEvent
NetworkRequestCancelEvent
NetworkRequestEvent
Extends
Initializes its own properties via reflection when beeing parsed.
Microservice
1 owner
Autoscaler
0..* children
ServiceOwnedPattern
PatternData
+ type: String
+ config: Map<String,Object>
+ tryGetPattern() : Pattern
Enitiy
Pattern
parses itself into
InstanceOwnedPattern
notifys of updates of travaling Request
RequestSender
+ update_listeners: PriorityQueue<IRequestUpdate
+ sendRequest(name:String,req:Request,target:Mi
+ sendRequest(name:String, req:Request,target:Microservice)
+ notifyRequestFailed(req:Request)
registers on each request
notifies prioritized of updates; notification is consumeable
RetryManager
NetworkPattern
+ getPriority() : int
CircuitBreaker
implements
MicroserviceInstance
Generator
IRequestUpdateListener
+ onRequestSend()
+ onRequestArrivalAtTarget()
+ onRequestFailed()
+ onRequestResultArrivalAtReq
registers as listener to RequestSender

Figure 5.6: Class Structure Excerpt

Load Intensity Model Request Generator

Version 2.0 of *MiSim* only supported interval base request generators. These can periodically create a fixed number of requests to specific endpoint. During the modification of *MiSim* these were extended to support start and stop behavior, allowing a more precise usage during specific simulation times. However, interval based generators are still respectively static and can not comfortably represent the complex load behavior of real system. Linear or exponential trends and workload curves can only be accurately realized by utilizing many generators and a Fourier analysis, making an experiment definition overcomplicated. Therefore, a new type of Generator was created, the `LIMBOGenerator`. It supports load intensity models created with LIMBO [KHK14a] which is based on the Descartes Load Intensity Model [KHK14b], a sophisticated tool for creating load profiles. As this is the same format of load profiles used in the preceding case study and other correlating projects, the load profiles of these experiments can be precisely simulated. Additionally, `LIMBOGenerators` can also be set to a *repeating*-mode in which they repeat the current profile indefinitely.

CPU Behavior and Responsibilities

With the restructuring and new separation of concerns, the CPU class was also reworked. In version 2.0 of *MiSim* this class was not only involved in scheduling threads, but also took care of some network behavior. I.e. it took care of the circuit breaker pattern. The new implementation fully concentrates on accepting and monitoring threads. The original version only supported an unmodifiable fixed quantum Round-Robin scheduling. As mentioned in Section 5.1.2, this is not suitable approach. Therefore, the new CPU implementation leverages a `CPUProcessScheduler` that can utilize the SARR algorithm that was proposed by Matarneh [Mat09]. Instead of using a fixed time quantum, this implementation calculates the time slices based on the median of the remaining burst time of all scheduled processes. Further, additional scheduling strategies were added. Specifically, the new default scheduling is a 3-Layer MLFQ utilizing SARR queues. To be more flexible and allow further use cases of the simulator, the ability to simulate FIFO and SPN scheduling was also implemented (see [SGG05]). Since the CPU utilizes the strategy design-pattern [Gea02], further scheduling strategies can be added easily.

Report Framework

The original version of *MiSim* has a very rudimentary statistics framework. It is very static and utilizes a fixed amount of nested hashmaps. Objects that want to write data into a specific hashmap had to search it themselves via concatenated method calls. Further, the export of data was not elegant, containing a lot of redundant code blocks and static identifiers. Therefore, a secondary report framework was created. It only requires the class or object that wants to report data, to create a `Reporter` object, e.g. a `MutliDataPointReporter` and provide a self-identifying dataset prefix. This `Reporter` object can then be used to create datapoints in any dataset and of any type. At the end of the simulation the data gathered by all `Reporters` is automatically collected and saved in a raw *csv*-format. Additionally, the simulator comes with a set of python scripts that can visualize some of its output metrics, like response times, instance counts and CPU usage. The dependency graph that is produced in *MiSim* 2.0 is also still working and will be reported on every run.

Extensibility

Extensibility was not the focus of the reengineering of *MiSim*, the focus was rather on creating a stable and well-working core implementation. However, the new architecture still offers some features that improve its extensibility, thanks to the use of architectural patterns. The `CPU`, `Loadbalancer` and `Autoscaler` classes are using the strategy design-pattern [Gea02], therefore new strategies can be added effortlessly. The definition and importing of new pattern is mostly automatized, as described by Section 5.2.4. Further types of requests can be added effortlessly (e.g. content-rich requests for caching) and the `RequestHandlingProcess` allows the integration of new handling strategies. Lastly, the aforementioned report framework is also extendable. Both the reporter and exporter of this framework can be easily switched out or extended, while still keeping compatibility.

6 Evaluation

The evaluation of the *MiSim* 3.0 was conducted on several layers. It aims to show that *MiSim* 3.0 satisfies the requirements that were presented in Section 4.2 sufficiently and that the different implemented simulation features behave as expected. Besides a static analysis of the simulation features and behavior, the evaluation utilizes two real world systems to calibrate an architecture model and run four actual resilience scenarios. These scenarios will then also be simulated and their result accuracy analyzed.

First, Section 6.1 presents an overview of which requirements the new architectures supports in comparison to the former version. Then, Section 6.2 presents the findings of a code review of the usability and architecture of the simulator. In the interest of performance, Section 6.3 explains some of *MiSim* 3.0 internal behavior and especially what problems can potentially arise and how they might be fixed. Section 6.4 then shows that the newly implemented features behave as expected. Specifically, this section looks at the behavior of the implemented patterns. Additionally, Section 6.5 looks at the general accuracy of the response time simulation with the use of four real world scenarios.

The version of the simulator that was used for the evaluation is available on Zenodo [Wag].

6.1 Requirements Analysis

In general, most of the requirements that are presented in Section 4.2 are implemented into *MiSim* 3.0. An overview of these and a comparison to *MiSim* 2.0 can be found in Table 6.1.

Most of the requirements were implemented successfully. An exception are the requirements 5.2 and 5.3, that were not implemented as direct outputs. However, these metrics can be calculated easily by processing the raw output data of the simulation. Furthermore, automatic self-restarting was not yet implemented. But the usage of a `SummonerMonkey` can simulate this behavior manually. Lastly, as previously mentioned in Section 5.2.4, *MiSim* does not simulate the content of requests. Therefore, a caching implementation was left for a later extension.

6.2 Code and Usablity Review of *MiSim* 3.0

In the interest of code quality a code review of *MiSim* 3.0 was conducted. In total, five senior researches and a student who previously worked on *MiSim* took part. The review was structured into three parts. First the general usability was discussed, then comments on the architectural style were collected and lastly the correctness of the implemented algorithms was evaluated.

	MiSim 2.0	*MiSim* 3.0
R1 (uses DES)	Y	Y
R2 (headless mode)	Y	Y
R3 (lightweight)	Y	Y
R4 (parallel runs)	Y	Y
R5 (output metrics)		
R5.1 Response Times	Y	Y
R5.2 Error Rates	N	N
R5.3 Throughput	N	N
R5.4 Queue lengths	Y	Y
R5.5 Execution Traces	Y	Y
R6 (raw output)	Y	Y
R7 (common architecture desc.)	~ [1]	Y
R8 (case study faultloads)	~ [2]	Y
R9 (CTK faultloads)	~ [2]	Y
R10 (LIMBO support)	N	Y
R11 (resilience features)		
R11.1 Self-healing (restart)	N	N
R11.2 Auto Scaling	N	Y
R11.3 Load Balancing	N	Y
R11.4 Retry	N	Y
R11.5 Circuit Breaker	Y	Y
R11.6 Rate Limiter	Y	Y
R11.7 Caching	N	N
R12 (compatability)		
R12.1 Cambio Scenarios	N	Y
R12.2 [Zor21]	~	Y
R12.3 TransVis [Bec21]	~	~

[1] Architectural description is supported by tools and extractable from Jaeger and Zipkin traces.
[2] Supports instance/service/device killing.

Table 6.1: Requirements evaluation comparision of *MiSim* 2.0 and 3.0.

Most participants had no problems setting up the simulator and running some exemplary and manually created simulations. However, it was noted that a "Hello World" example would be helpful and the manual installation of the DESMO-J library can run into troubles. Both these complaints were fixed after the review.

The general reception of the new architecture was positve. Most participants noted that it is sensible choice. There was no major critic, but every participant had several minor improvement suggestions. Some noted, that the documentation could be clearer. Also, the general package structure was criticized. It was further remarked, that the usage of the strategy pattern could be more elaborate, e.g. it would also be applicable to the retry and circuit breaker patterns.

On the algorithmic side no complaints were raised. There was a discussion about the inner workings of the circuit breaker that resulted in an agreement, that the related classes should be renamed for clarification.

Since there were no major findings, most of the critiqued parts are scheduled to be fixed in a later version.

6.3 Performance Evaluation and Problems

The performance of *MiSim* 3.0 strongly depends on the executed scenario. As it is a DES it can theoretically perform simulations very fast. In practice, simulations may take only 200 ms but also can last up to several minutes depending on the amount of scheduled events. Similarly, the memory usage of the simulation grows with the size of the simulation. However, the size of the input architecture does only influence the used memory slightly, since only actively interacting parts of the system are simulated.

Most of the runtime performance costs can be traced back to two concrete sources, (1) data collection methods and (2) the DESMO-J scheduling engine. (1) often involves searching through one or multiple lists of data and is most often executed every time the data set changes. This combines to a performance impact in which over 65% of the total computation time is spent on the data collection of the `MicroserviceInstance` class alone. A very noticeable speed-up is gained when the data collection for this class is disabled. In the future, a more suitable solution to this problem will be implemented.

(2) is rooted deeper in the simulation engine. When turning off data collection, a flame graph, as shown in Figure 6.1, reveals that most of the simulation time is spent on modifying the scheduler's `EventTree`. This tree holds a `TreeList` containing all scheduled events. DESMO-J keeps this list sorted (by scheduling time and priority) upon element insertion, based on a binary search. Therefore, the insertion has a fast run time classification of $O(2\log n)$. However, canceling events is costly, as this requires the `EventTree` to linearly search for the respective node ($O(n)$) [Fou05] and then to remove it ($O(\log n)$). This effectively makes it more efficient to flag events as canceled and checking this flag during their event routine. An example for this can be found in the `NetworkRequestTimeoutEvent` class. Unfortunately this strategy can not be applied to all event cancels. For example, the `CPU` class does reschedule very often. This process involves the canceling and insertion within the `EventTree`. Figure 6.1 shows that around 20% of the total run time is spent on this.

DESMO-J also has some memory usage impacting properties. By design, it gives a unique name to every event and entity. To ensure the uniqueness, each name is concatenated with a number (starting with 1) and both are stored in a look up table named `NamingCatalog`. If an entity or event with an equal name is created, the number is incremented. In the case of *MiSim* 3.0 the naming of entities (e.g. traveling requests) is based on each other. This means, that every time a dependency request is created, a new name and a new entry in the `NamingCatalog` is created. Unfortunately, the `NamingCatalog` does not allow the removal of entries. This leads to an accumulation of Strings that are only used once, for a limited amount of time, but are held in memory forever. For bigger simulations this `NamingCatalog` can take up multiple gigabyte of memory and is therefor not very

```
{
  "type": "retry",
  "config": {
    "maxTries": 6,
    "baseBackoff": 0.1,
    "maxBackoff": 2,
    "base": 2
  }
}
```

Listing 6.1: Retry configuration during the proof of concept scenario.

memory effective. In a later version of *MiSim* it is planned to replace the `NamingCatalog` with a custom implementation that allows removing. However, since this solution requires reflection, it might have an impact on run time performance.

6.4 Behavior Analysis of the Implemented Patterns

As the modification of *MiSim* changed a lot of its original architecture and behavior, it is critical to determine, whether the new features are working as expected and are sufficiently accurate. First Section 6.4.1 introduces a small system that gets reused throughout the following tests. Then Sections 6.4.2 to 6.4.4 will look at the behavior of the circuit breaker and retry respectively.

6.4.1 Proof of Concept System

For the concept proofs a simple architecture consisting out of two services was used. A *depending service* offers a public endpoint for load generation and relies on an *independent service*. A call to the depending service always triggers a single dependency request to the independent service. This architecture is shown in Figure 6.2. In the following sections, these services will be decorated with the different types of resilience features to proof their general behavior correctness. Since there is only one entry point for the system and only one dependency, the behavior of all patterns is as isolated as possible.

6.4.2 Retry Pattern

As explained in Section 5.2.4, *MiSim* 3.0 support a exponential retry strategy. This evaluation looks at both, the jittered and unjittered variant. Details on these strategies are presented in Section 2.2.1. For this proof of concept the simple system that is described in Section 6.4.1 is used. The depending service was equipped with a retry pattern and configured to handle all requests with no computing time. In the executed scenario, the independent service was forcibly killed with a chaos monkey at 30 STU and restarted at 60 STU with a summoner monkey. This forces the retry to act during the downtime. The system is constantly loaded with five requests per 0.1 STU, that arrive simultaneously at the dependent service. The retry pattern was configured described by Listing 6.1.

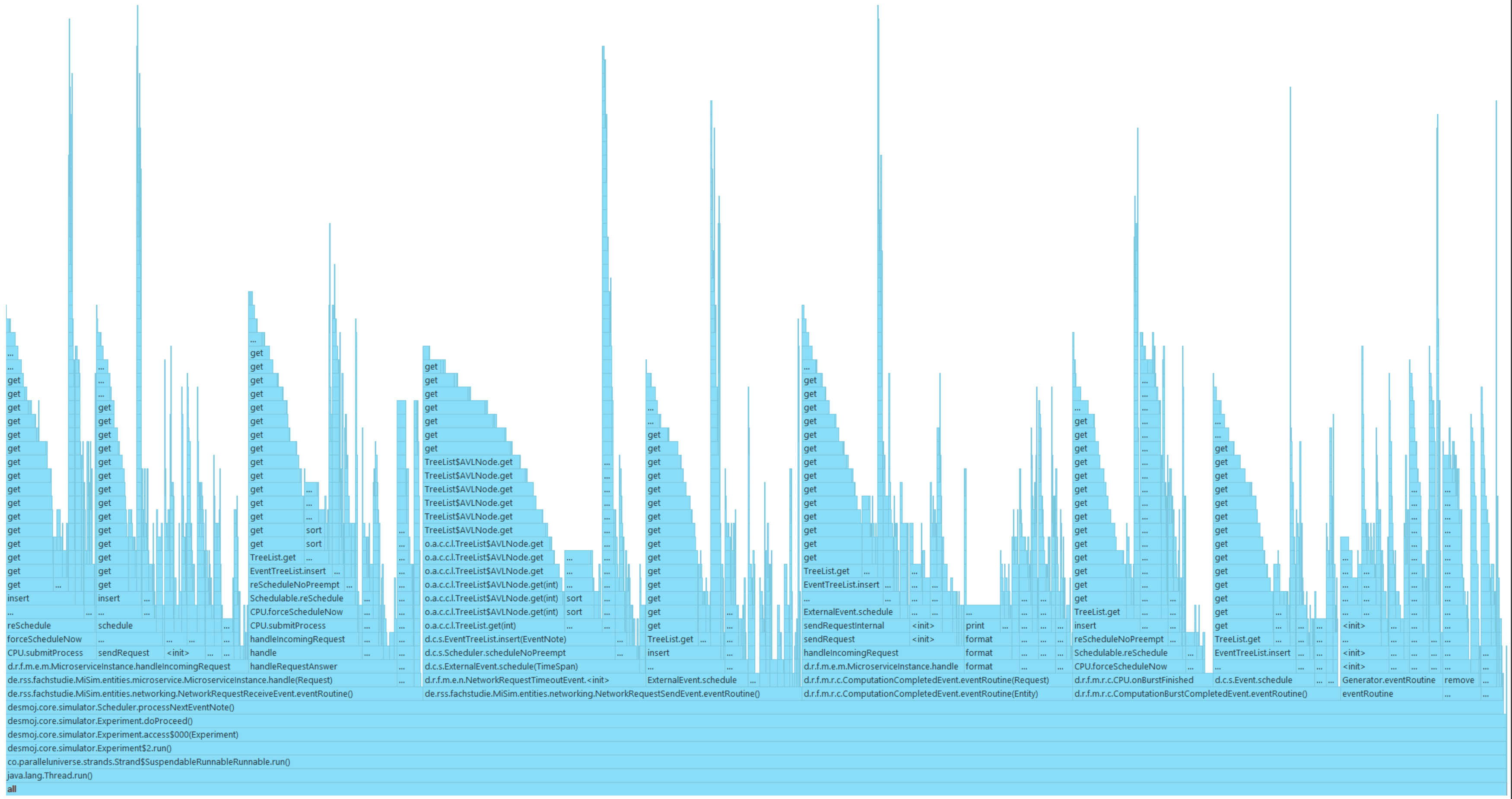

Figure 6.1: Flame graph of an exemplary run. The x-axis depicts the CPU time of the respective method calls and the y-axis the resulting call stack. Here legacy package naming is still in use.

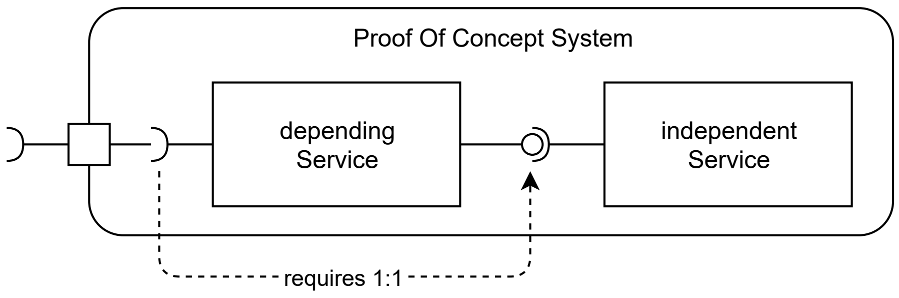

Figure 6.2: Architecture of the simple system that is used for the proof of concept testing.

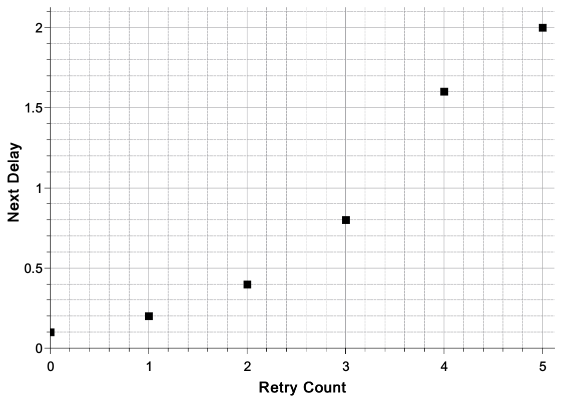

Figure 6.3: Discrete delay values of the used retry function.

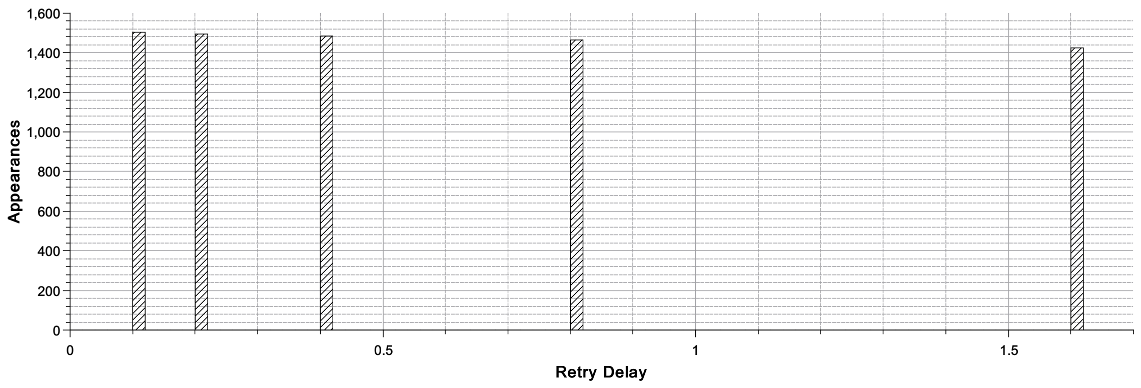

Figure 6.4: Unjittered retry delay distribution, displayed as a histogram with 0.1 STU bins.

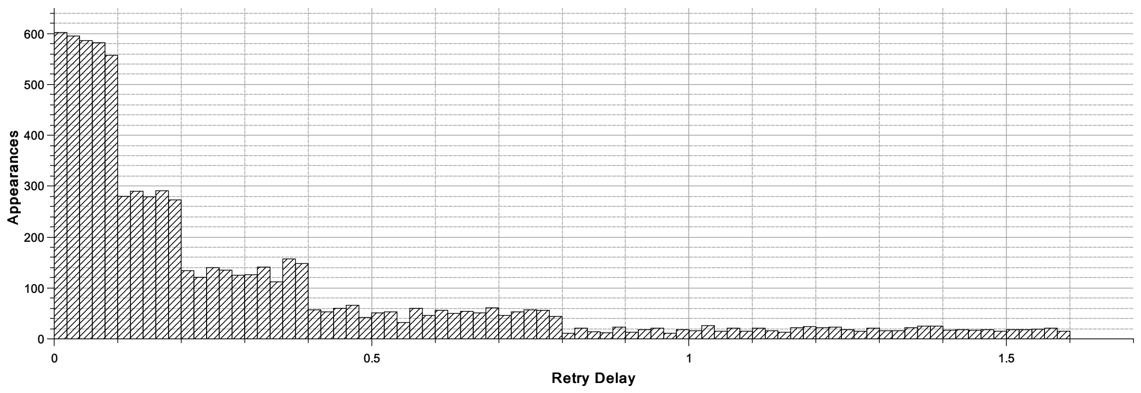

Figure 6.5: Jittered retry delay distribution, displayed as a histogram with 0.2 STU bins.

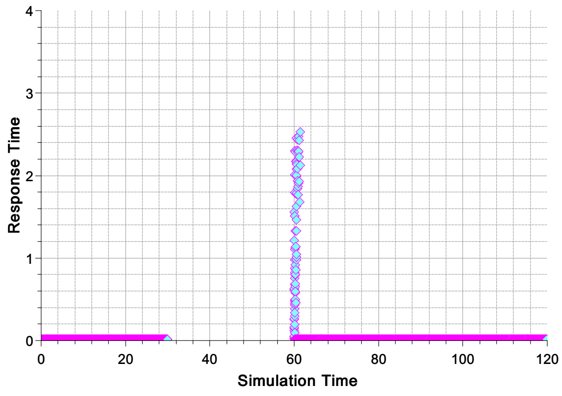

Figure 6.6: Respones time graph of the jittered Retry.

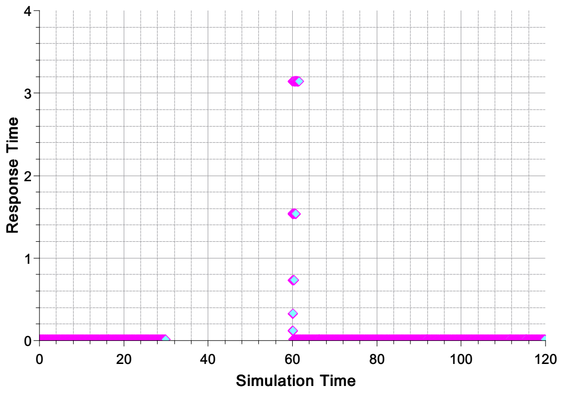

Figure 6.7: Respones time graph of the unjittered Retry.

The property `jittering` for the Retry was set to `false` for the unjittered version of the scenario. This configuration produces the delays displayed in Figure 6.3. The squares show the discrete values that should be generated by unjittered retry.

Figure 6.4 summarizes the unjitterd delay distribution during the scenario. The histogram clearly shows that the retry delays are calculated as expected, since the values present in Figure 6.3 are reappearing. In contrast to this Figure 6.5 shows the histogram of all calculated retry delays of the jittered retry. The five exponentially distributed groups of delays that are found in Figure 6.4 are clearly visible as steps in the graph. Overall, a good distribution inside the groups can be observed.

The actual response times results of the scenario are shown in Figures 6.6 and 6.7. These response time graphs clearly show that the retry patterns is taking effect. In both cases requests are accumulated during the down phase, due to dependent service retrying them up to five times, whilst other additional requests arrive. These request do have a higher response time than normal requests and appear higher on the graphs. Further, these graphs also show that the jittering property of the retry has its expected effect of distributing retried requests. In Figure 6.7 there are clusters of retries that arrived roughly at the same time and are therefore retried and executed at the same time. In contrast to that, Figure 6.6 does show almost no clustering and a higher distribution of jittered request responses. Since the full jitter cuts the delay of retries on average in half, in comparison to

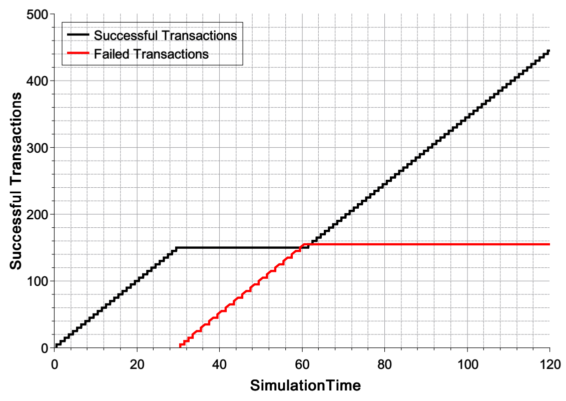

Figure 6.8: Accumulated successful and failed transactions, monitored by the circuit breaker.

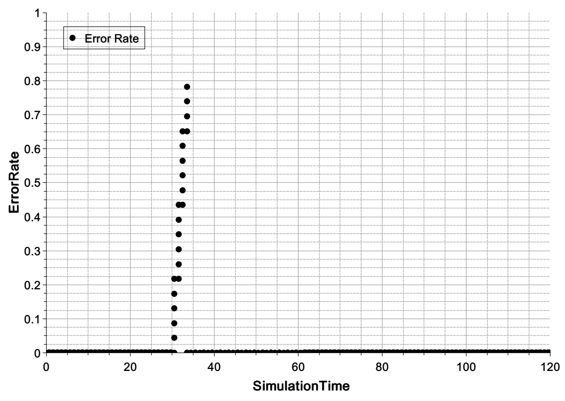

Figure 6.9: Error rates monitored by the circuit breaker.

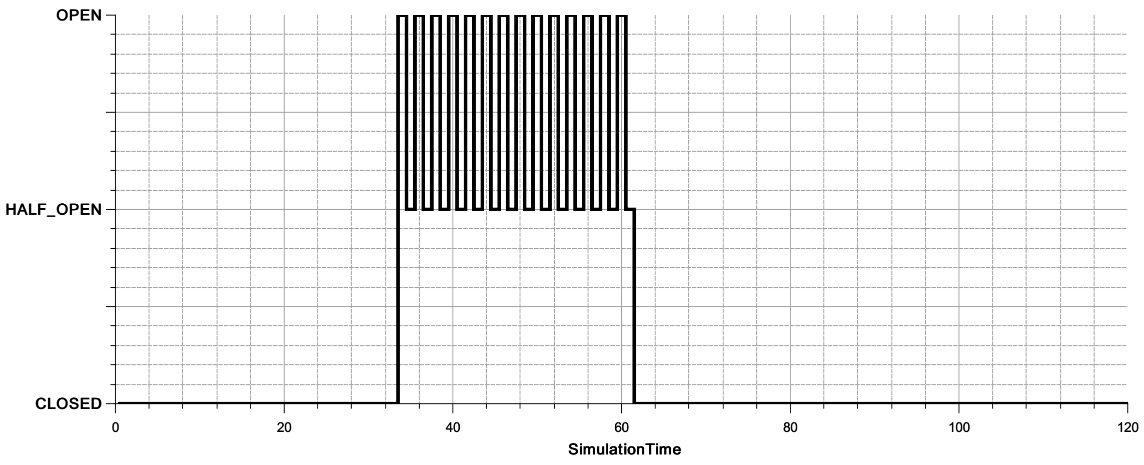

Figure 6.10: State graph of the circuit breaker.

the unjittered variance, response times are a bit lower. However, this also has the effect that requests fail earlier. This also halved the amount of accumulated request at the point of restart from 35 (unjittered) to 16 (jittered). In practice this could be compensated for by doubling the `baseBackoff` value of the configuration of the jittered retry.

6.4.3 Circuit Breaker

The proof of concept for the circuit breaker utilizes a similar scenario as Section 6.4.2. In this case the independent service is also shut down at 30 STU and restarted at 60 STU. The system is under a load of five simultaneous requests every second. The depending service was decorated with a circuit breaker pattern that was configured with the configuration shown in Listing 6.2

The system behavior during the scenario as seen by the circuit breaker is best visible in Figure 6.8. The amount of successful dependency completions raises until the 30 STU mark, then plateaus until the restart of the service at 60 STU. Meanwhile, the collected amount of failed transactions raises during the downtime, but does not change otherwise.

```
{
  "type": "circuitbreaker",
  "config": {
    "errorThresholdPercentage": 0.8,
    "sleepWindow": 1,
    "rollingWindow": 23
  }
}
```

Listing 6.2: Circuit breaker configuration during the proof of concept scenario.

```
{
  "type": "autoscale",
  "config": {
    "lowerBound": 0.2,
    "upperBound": 0.9,
    "holdTime": 20
  }
}
```

Listing 6.3: Autoscaler configuration during the proof of concept scenario.

Figure 6.9 shows the respective error rates. They are calculated by the common ratio formula, that is displayed in Equation (6.1). However, since the circuit breaker was configured with a fixed rolling window size, only the last 23 transactions are considered for this evaluation. Also, shortly after the error rate surpasses the threshold of 80% the rolling window is cleared, as the circuit breaker opens.

$$ErrorRate = \frac{\#FailedTransactions}{\#FailedTransactions + \#SuccessfulTransaction} \tag{6.1}$$

Lastly, Figure 6.10 shows how the state of the circuit breaker oscillates between the open and half-open state during the downtime. As expected, at 33.5 STU the circuit breaker opens shortly after the independent service was killed and after the error rate surpassed the target threshold. After the defined sleep window duration, it goes into the half-open state. At this point, a single request is let through and after observing this request's failure the circuit breaker goes back into its open state. Only at the 60 STU mark, the circuit closes again, after requests are successfully completed again.

6.4.4 AutoScaling

The proof of concept for the autoscaler uses the same simple architecture as before. This time, the depending service is configured to handle up to two million requests per STU and therefore only impacts response times minimally. The independent service is configured to handle up to 100 requests per STU per instance. During the scenario, the system is loaded with a linear increasing load

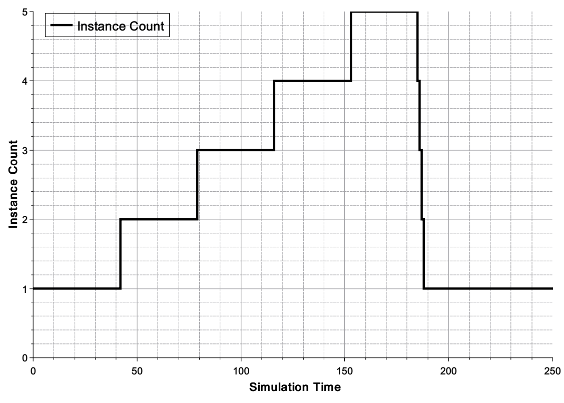

Figure 6.11: Instance count of the scaling service.

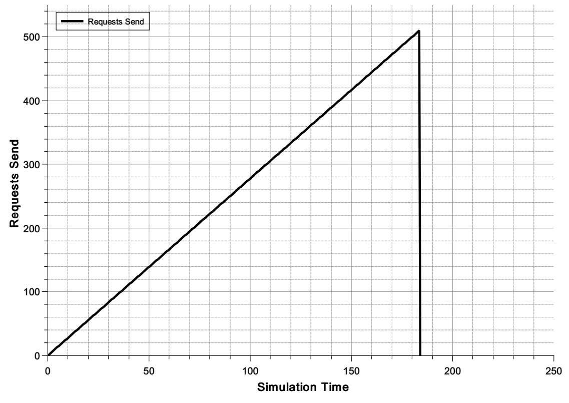

Figure 6.12: Load during the autoscale scenario.

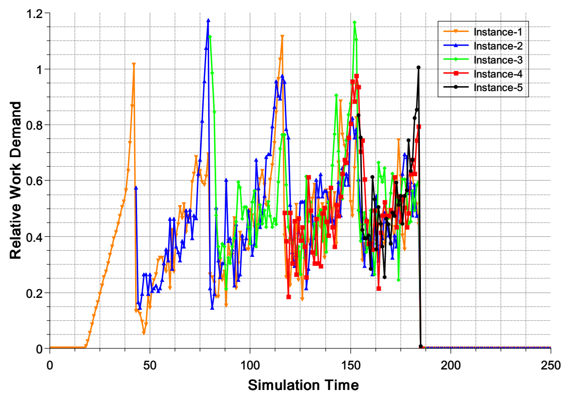

Figure 6.13: Utilization of the instances of the independent service.

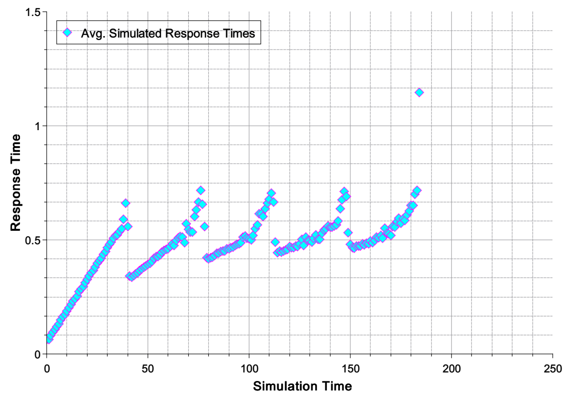

Figure 6.14: Response times during the autoscale scenario.

that slowly rises from 0 to 500 simultaneous requests. This load curve is displayed in Figure 6.12. The autoscaler is enabled for the independent service with the configuration shown in Listing 6.3. The service starts with a single instance.

This leads to the service incrementing its number of instances whenever the average relative utilization of all instances is above 90%. Further the service is not allowed to downscale for 20 STU after the latest upscale event. When the average relative utilization is below 20% the services if forcibly downscaled one instance at a time. These utilization checks are executed on every passing STU.

Figure 6.13 shows the respective relative load of all instances during the scenario. Each time this load exceed the 90% mark a new instance is started. Therefore, the relative utilization is dropping back down after each spike. This graph also visualizes that the load balancer is working properly, as the load between instances is almost always distributed evenly.

The evolution of the instance count is shown in Figure 6.11. Similar to Figure 6.13 it shows an increase in instance each time the utilization surpasses the upscale threshold up to a maximum of five instances. Since five instances are enough to handle the maximum load, there are only four

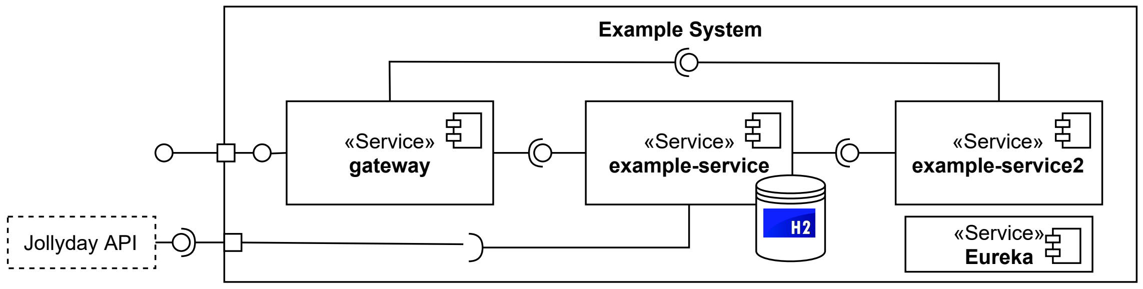

Figure 6.15: Example Architecture that is used for the Evaluation.

upscale events. The fifth and last spike in utilization at 180 STU does not exceed 90% average relative utilization. Once the load and utilization drop to zero at 180 STU, the autoscaler starts reducing the amount of instances. In the end, only one instance remains.

Lastly, Figure 6.14 shows how the request response times develop during the scenario. It can be observed that they never exceed 2 STU. After every upscale event, the response times drop back below 1 STU. In the end the average response time is 0.569 STU with a standard deviation of 0.313 STU. Configuring the autoscaler more aggressively improves these results. For example, setting the `upperBound` to 0.7, lets the service scale to six instances and reduces the average response times to 0.573 ± 0.288 STU.

6.5 Scenario Evaluation

To check the accuracy of the simulation a comparison to a real previously elicited scenario is conducted. First, Section 6.5.1 presents the example system that is used for this evaluation. Section 6.5.2 presents how the simulation was calibrated to match the behavior of the real evaluation system. Section 6.5.3 presents the evaluated real world scenarios. Sections 6.5.4 to 6.5.7 go into detail about how accurate *MiSim* 3.0 simualtes these scenarios.

6.5.1 Evaluated System

Figure 6.15 shows the actual system that is used for the following response time accuracy and scenario simulation. The system consist of 3 services. A gateway service receives all incoming requests and distributed them onto the other services. There are five endpoints that are explained in Table 6.2. The gateway has exactly one instance and is equipped with a retry mechanism. It is configured with the following configuration:

```
- name: Retry
  args:
    retries: 5
    statuses: BAD_GATEWAY,SERVICE_UNAVAILABLE
    methods: GET, POST, PUT
    backoff:
    firstBackoff: 200ms
    maxBackoff: 2s
    factor: 3
```

Endpoint Name	Description
External_Dependency	Triggers a call from example-service to the external Jollydays API. Results are cached indefinitely.
Internal_Dependency	Triggers a call from example-service to example-service2.
Unaffected_Service	Directly calls example-service2.
DB_Write	Creates an entry in example-service's H2 database.
DB_Read	Queries an entry in example-service's H2 database.

Table 6.2: Endpoints of the example Architecture.

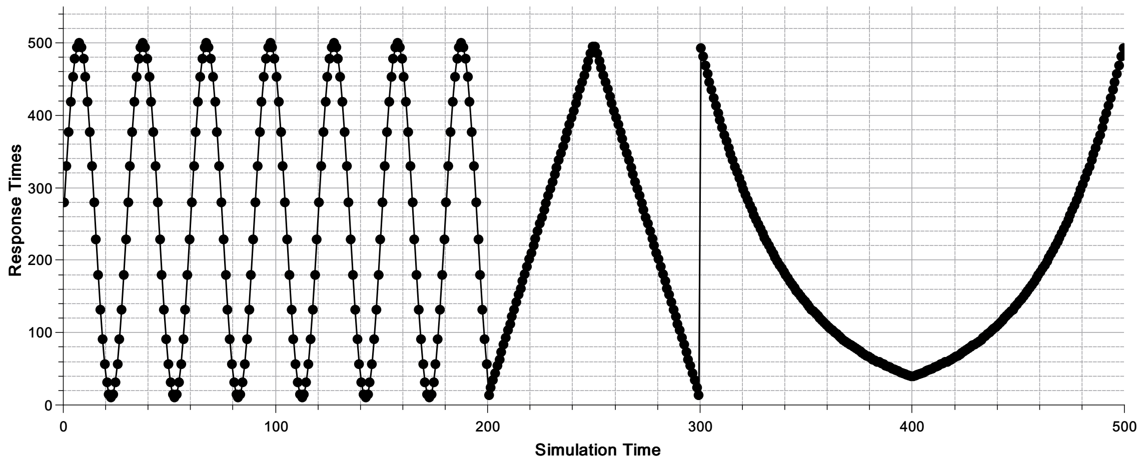

Figure 6.16: Load profile that was used for the calibration.

```
basedOnPreviousValue: false
```

Listing 6.4: Retry configuration of the example system.

By default, the example-service and example-service2 have two instances. The example-service is equipped with an in-memory database. As a service discovery mechanism, a eureka service is utilized.

6.5.2 Calibration

To calibrate the simulation a load test was run against the real system. Using the HTTP Load Generator [KDK18] the system received a workload that contained multiple types of load curves. Figure 6.16 shows the exact workload model. During the whole time the workload is in between 500 and 10 requests per second. The model starts of with a cosine wave between 0 and 200 STU and then transitions into a linear growth and decline between 200 and 300 STU. Then a jump from 10 to 500 requests per second happens and a exponential decline to 40 requests per second at 400 STU follows. Lastly, the load grows exponentially back up to 500 requests per second.

The results of the load generator provide mean response times and the coefficient of variation for all endpoints. Since *MiSim* can currently only specify one general network latency, the following statistics will only look at the average over all five endpoints. Additionally, the calculation time

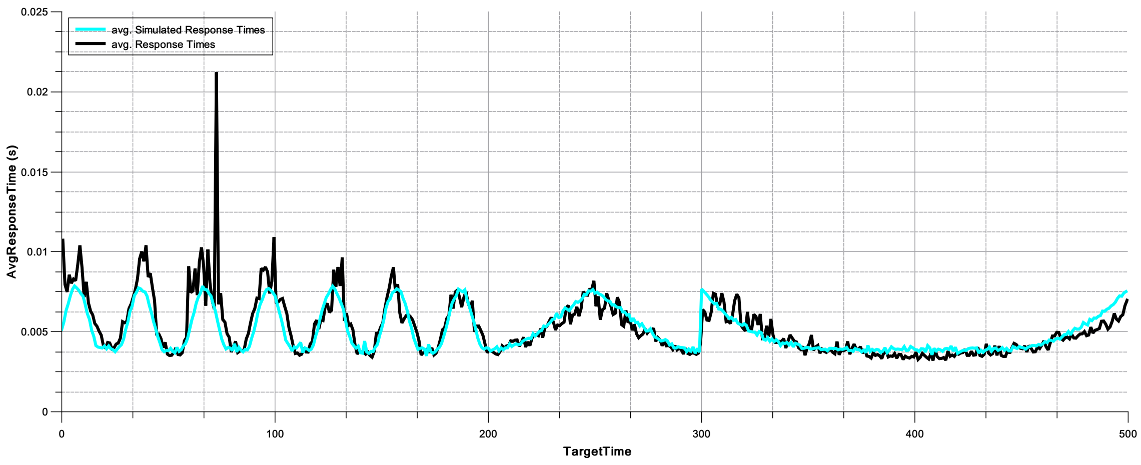

Figure 6.17: Comparison of simulated and real response times.

in every service was observed. In any case this duration was usually between 1 and 2 ms. Only the *External_Dependency* endpoint has a higher response time of around 30 ms, since it needs to collect data from an external API. However, this services uses a non-expiring cache and the possible argument range is rather small. Therefore, the cache is quickly saturated and it can also be assumed that this service answers immediately. With this information the architecture model for the simulation can be created. Obviously, the structure that is shown in Figure 6.15 is taken over. No resilience patterns are applied. The network delay was adjusted to recreate the response values that are collected during the load test. Lastly, the network latency of 1.6 ± 0.6 ms was calculated based on the load generator statistics.

Since the HTTP Load Generator evenly distributes the workload over all five endpoints, the simulation experiment defined five LIMBO-Generators that produce one fifth of the original load on every endpoint.

Figure 6.17 compares the simulated response times to the response times of the real system. On average the difference between both lines is 0.6 ms, which relates to an average inaccuracy of 10%. In general, the simulated and real response time curves behave similarly, however the simulation behaves more consistent than the real system. This is specifically visible during the sinusoidal curves, where with the workload the deviation in response times also grows. Further, in every run the real system experiences a reproducible large spike in response times 60 seconds into the test. Unfortunately, this can not be simulated in the current version of *MiSim*. However, these results show that *MiSim* 3.0 can be calibrated to accurately simulate a microservice architecture.

6.5.3 Scenario Overview

Table 6.3 lists the scenarios that are used for the further evaluation. They are orientated after the original scenarios we elicited during our preceding case study [KWK+20]. #1 relates to scenario 01 Peak(LinCo)/Ser/Abr of the case study. It describes a case in which the system experiences a sudden, linear growing load spike due to user requests. Usually the whole system is affected by this stimulus, as it does not concentrate on a specific use case. If this scenario occurs, the system is expected to still respond in under one second for 99% of requests within the next 20s.

ID	Source	Stimulus	Artifact	Enviorment	Response	Resp. Measure
#1	User	System experiences a linear workload spike	All services	Normal business operation	Request are answered correctly and in time.	Response times should be lower or equal than 1 second in 99% of cases within 20 seconds after the last load spike.
#2	User	System experiences a exponential workload spike	All services	Normal business operation	Request are answered correctly and in time.	Response times should be lower or equal than 1 second in 99% of cases within 20 seconds after the last load spike.
#3	A Service *R*	*R* does not respond.	Services that communicate with *R*	Normal business operation	Service *R* gets restarted	Downtime should be below 1 minute.
#4	A Service *R*	*R* responds with delay.	Services that communicate with *I*	Normal business operation	Restart *R* if it is too slow.	Response times should be lower or equal than 1 second in 99% of cases over the last 30s.

Table 6.3: Scenario descriptions that are used for the evaluation.

#2 is very similar to #1 and relates to scenario 02 Peak(ExCo)/Ser/Abr. In this case the workload spike has an exponentially characteristic. Otherwise, the scenario is identical to #1.

#3 revolves around the failure of a whole service. In the preceding case study this scenario was named 11 Failure(SerE)/Ser/Ber. Source for the stimulus is a service *R* of the system. The stimulus describes, that the service becomes unresponsive and does not answer to any requests. This effects all services (and instances) that want to communicate with *R*. The system is expected to restart at least one instance of *R* during the next minute to handle this scenario.

Lastly, #4 is a delay scenario to demonstrate further capabilities of the simulator. In this case a service *R* is again the source of the scenario. It stimulates the system by always responding with a delay. Again, all services that want to communicate with *R* are affected. The system is expected to replace or restart *R* if it produces too much delay.

All scenarios describe the expected system behavior during normal business operation. This means that the system is in a steady state before the respective scenario occurs. E.g. all services are running and all measurable business metrics are within the SLO limits.

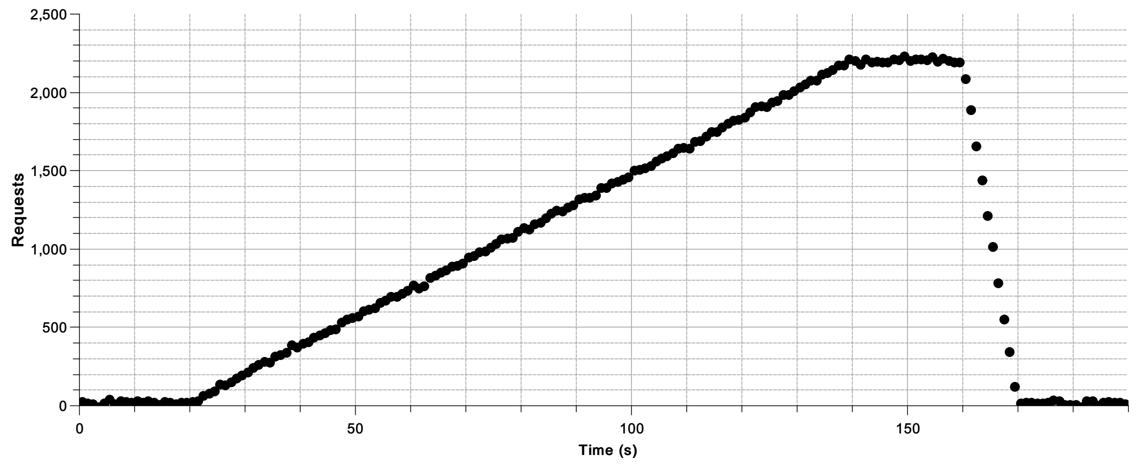

Figure 6.18: Workload curve used for scenario #1.

6.5.4 Scenario #1

As presented in Section 6.5.3 scenario #1 describes the expected system behavior under a linear rising workload. The specific workload that is used of this scenario was modeled in LIMBO and is shown in Figure 6.18. It starts off with a steady warm-up phase that also is thought to simulate the workload the system experiences during normal business operation. In this case it is assumed to be equivalent to around 20 requests a second. After a minute the workload starts increasing. Over the period of the next two minutes it evenly rises up to 2200 requests per second, effectively creating 110 times more load onto the system. After a relatively short period of 20 seconds the workload drops quickly back down to the base level and the system is given a chance of to reach its steady state again. There is a slight normalized noise of 1 ± 5 added to the curve, to add a bit of randomness.

Unfortunately, the results of the simulation are very inaccurate. Figures 6.20 to 6.24 show the results for each endpoint and that they are not accurately simulated. The general response time calibration that was explained in Section 6.5.2 creates correctly simulated response times during the initial steady state and the cool down phase. However, from 100 seconds onward the simulated response times deviate heavily from the real response times. Whilst the actual system is slowing down under the load, the simulation is not affected. This hints to a failed calibration or more general major problem with the configurability of the architecture model under load.

To test this theory, this scenario was run with a small range of architecture models, that represented the same system, but scaled in terms of capacity and available threads. Neither of the models did recreate the system behavior that is seen in the aforementioned graphs. Even tho the calibration shown in Section 6.5.2 produced usable and accurate results, it can be assumed that this calibration is not usable for environment or use case of the system.

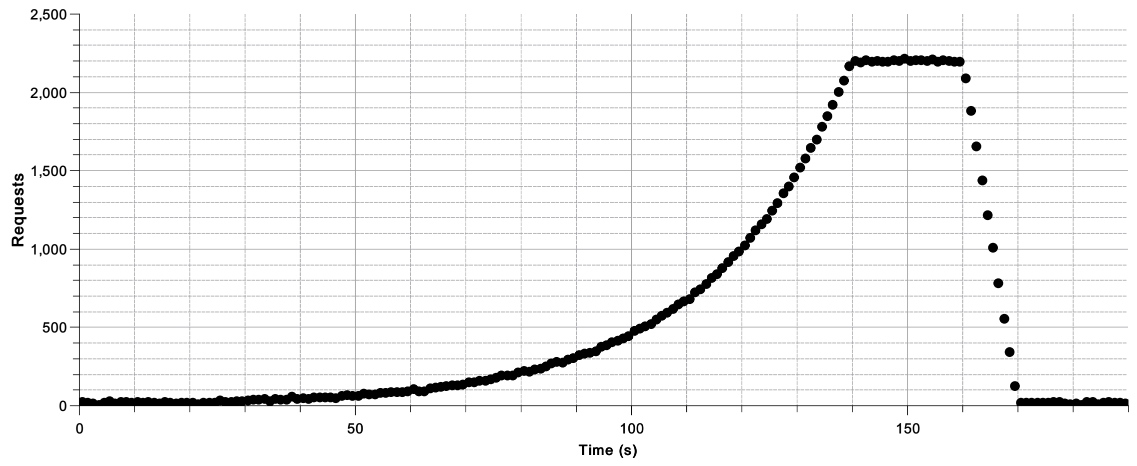

Figure 6.19: Workload curve used for scenario #1.

6.5.5 Scenario #2

Similar to scenario #1, scenario #2 puts a growing workload onto the system. In this case it grows exponentially. For the evaluation the workload model, that is displayed in Figure 6.19 is used. It is very similar to the model of scenario #1, but replaces the linear trend with an exponential one.

Unfortunately, the results of the evaluation of scenario #2, show again, that there might be an underlying problem with the configurability. Figures 6.25 to 6.29 show that the response times of the real system behave as expected. They grow exponentially with the rising workload demand and also fall back down as the workload does. Meanwhile, the simulated response times keep a steady response time and are not influenced by the workload.

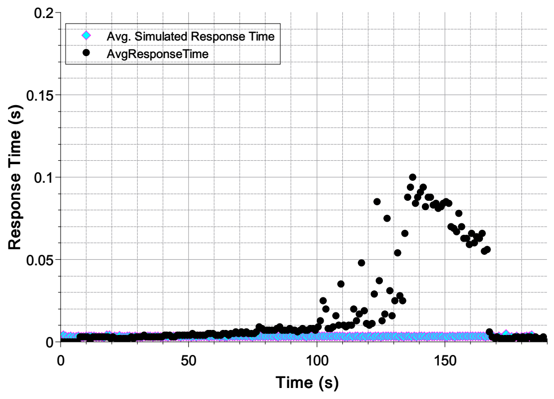

Figure 6.20: Response times of the External_Dependency endpoint during scenario #1.

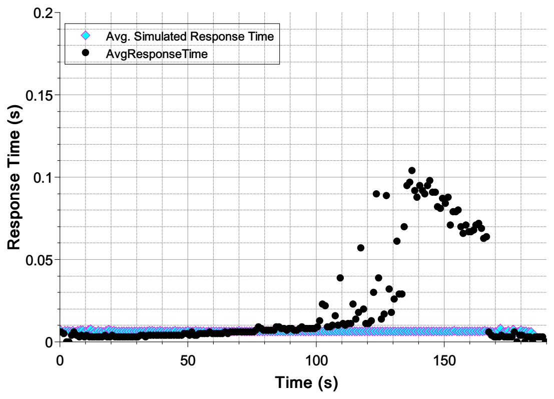

Figure 6.21: Response times of the Internal_Dependency endpoint during scenario #1.

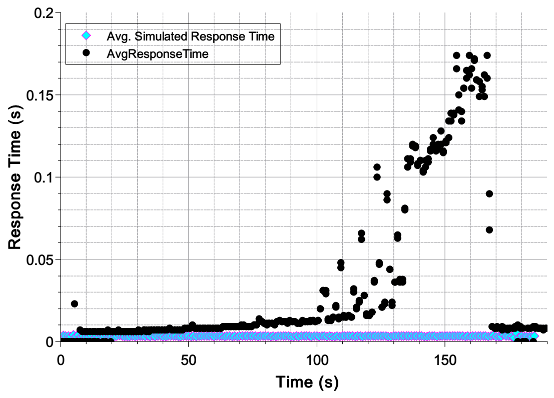

Figure 6.22: Response times of the DB_READ endpoint during scenario #1.

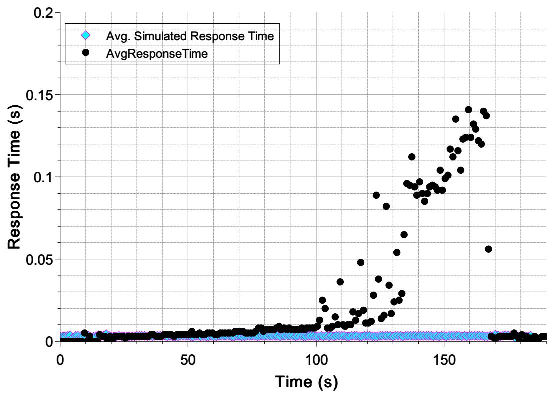

Figure 6.23: Response times of the DB_WRITE endpoint during scenario #1.

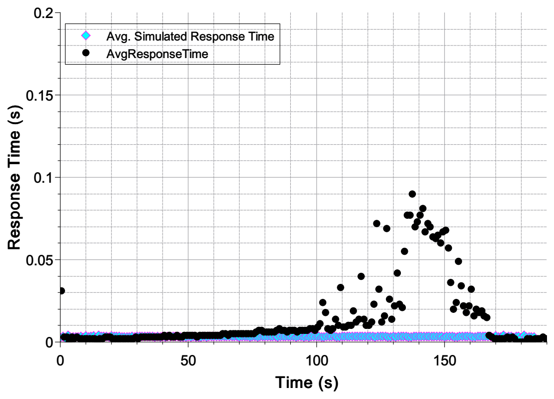

Figure 6.24: Response times of the Unaffected_Service endpoint during scenario #1.

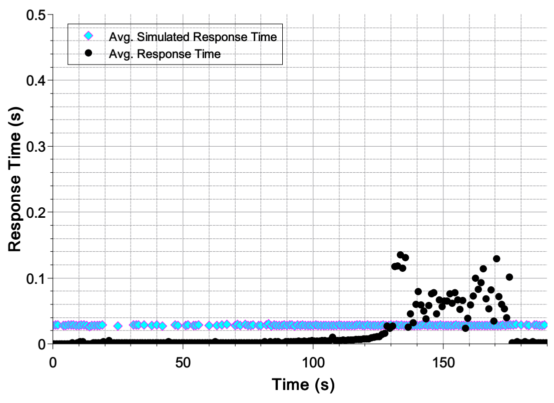

Figure 6.25: Response times of the External_Dependency endpoint during scenario #2.

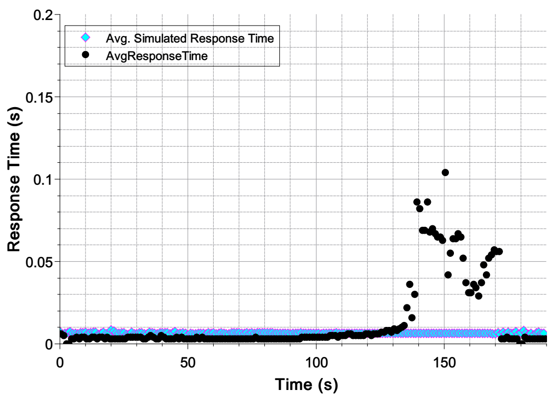

Figure 6.26: Response times of the Internal_Dependency endpoint during scenario #2.

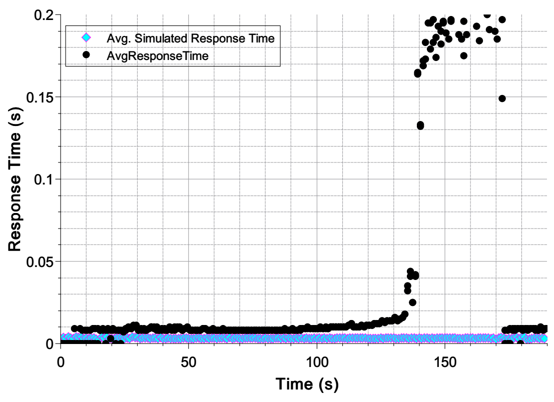

Figure 6.27: Response times of the DB_READ endpoint during scenario #2.

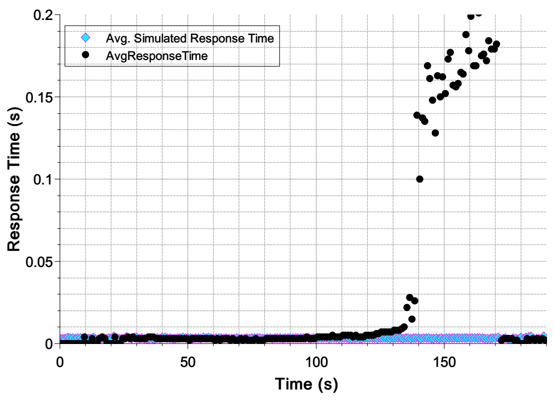

Figure 6.28: Response times of the DB_WRITE endpoint during scenario #2.

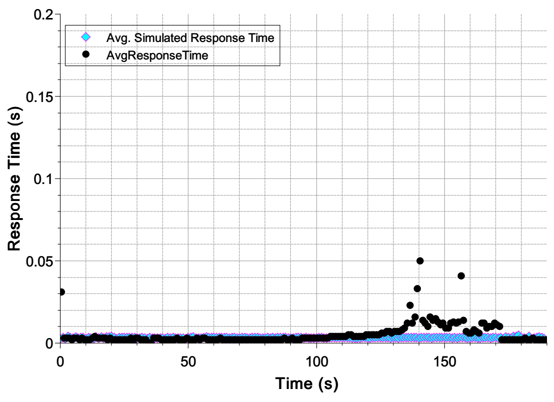

Figure 6.29: Response times of the Unaffected_Service endpoint during scenario #2.

6.5.6 Scenario #3

Scenario #3 describes a service failure. For this scenario, the system was loaded with an arrival rate a constant 1000 requests per seconds. During the scenario the example-service is killed at the 60s mark. In the real evaluation system it is restarted with both instances 30 seconds later. But the service is only available after the 125 second mark. This is most likely due to its start up time (container booting and Spring initialization) and the duration of the registration at the eureka server. Therefore the simulation restarts the service at 125 seconds into the scenario with a `Summonermonkey`.

For this scenario, the success and failure counts are specifically interesting. Therefore, Figure 6.30 compactly displays those. The simulation results are drawn in with dashed lines, whilst the real results are represented by solid lines. The output data of the HTTP Load Generator does additionally track dropped messages. These are messages the load generator planned on sending, but their requirements were not satisfied in time (8s timeout). For unknown reason these get aggregated, therefore the load generator drops many requests simultaneously.

Specifically during the steady phases, between 0 to 60 and 130 to 150 seconds, the amount of failed and successful requests are almost identical. During the failure state, between 60 and 125 seconds, the simulator tracks a constant amount of 800 failed requests per second. Since one of the five endpoints should be unaffected, the simulation also registers 200 successful requests per second during the downtime. The load generator reports the dropped requests instead, during this phase. Once it processed all dropped requests around the 95 seconds mark, the statistics show, that it also begins to count failed requests. This count evens out around the 110 seconds mark into a stable rate of 1000 failed requests per second, similar to the simulated results. However, since the load generator requires all other endpoints to be available before it can request the unaffected endpoint it also fails all requests. On the restart at 125 seconds its again visible that the calibration of the system is potentially not on point. The simulated system is able to cope with all (by the the retrier) aggregated request immediately, whilst the actual system slowly recovers to its normal steady state over the next 10 seconds. However, This could also be due to typical cold start behavior.

Figures 6.31 to 6.35 display the tracked response times during the scenario. In most cases the simulated response times match the real ones, with the exception of the restart moment at 125 seconds. Similar to the successful requests count, it becomes apparent that the real system is not able to handle the aggregated messages immediately. Over the period of the next 10 seconds after the restart the response times are slightly increased, which is not visibile in the simulated response times. Similaryl the response times of the actual system are slightly higher than the simulated ones at the beginning of the scenario.

In conclusion, it can be seen that general behavior during the scenario is closely simulated. The simulator correctly calculates the response times, shut down behavior and restart behavior. However, it currently does not support the simulation of cold start behavior, which diminishes the simulation accuracy immediately after the restart. The count of successful and failed messages behaves as expected, but differs from the data generated by the load generator, since it creates dependencies between the endpoints, which *MiSim* does not.

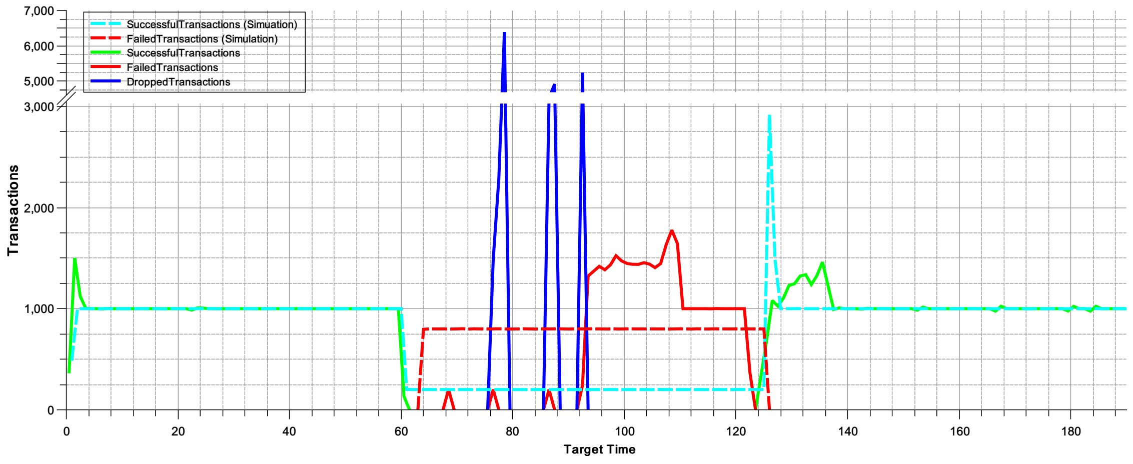

Figure 6.30: Amount of successful and failed requests of the real run and simulation during scenario #3.

6.5.7 Scenario #4

During scenario #4 the a service is answering with a noticeable delay. To produce this effect the evaluation system was injected with a network delay.

For this I utilized the Pumba chaos testing tool [Led] to manipulate the real evaluation system's network. For unknown reasons, the WSL2 Docker engine, that was used in the previous scenarios, was not able to execute the traffic control (`tc`) command on the docker containers. Therefore, this scenario was run on the slower Hyper-V based engine. This results into slightly altered behavior of the system and slower response times. To compensate for this, the architecture model was recalibrated.

During the actual execution of scenario #4, the system is loaded with a simplistic load curve. It creates a workload that rises from 20 to 1000 requests per second over a period of 60 seconds. The scenario execution injects a network delay into the example-service at 90 seconds. The network delay is configured to delay messages by 500 ± 200 ms for 1 minute. This delay should be applied to all incoming and outgoing requests.

Figures 6.36 to 6.40 compare the response time results of the simulation compared with the real ones. Looking at all graphs simultaneously shows, that the simulation is fairly inaccurate during the ramp up phase of the workload. This again shows, that *MiSim* may not accurately simulate load variations under certain conditions.

Once the workload and system reach a steady state, the simulation of the response times becomes more accurate. Looking at the delay period between 90 and 150 seconds reveals, that the delay simulation takes effect, but can be accurate on different levels. For the Internal_Dependency endpoint, Figure 6.37 shows that delay is accurately simulated. The Unaffected_Service endpoint, that is represented in Figure 6.40, is also not effected in both, the simulation and the real system. However, in the real system it its response times are stabilizing at a low point. Whereas in the simulation, no impact can be observed.

For the other three endpoints the simulation predicts higher response times than there actually are. Specifically in the case of the DB_WRITE endpoint the simulated times are nearly doubled, compared to the real results. This is an expected behavior of the evaluation system, as the added delay should be 1 s on average. The other database related endpoint DB_READ, that is represented in Figure 6.38, does also behave similarly unexpected. Its variation of response times is significantly higher than the one of the other endpoints.

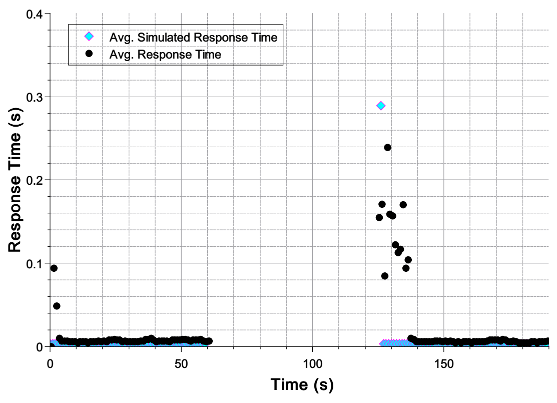

Figure 6.31: Response times of the External_Dependency endpoint during scenario #3.

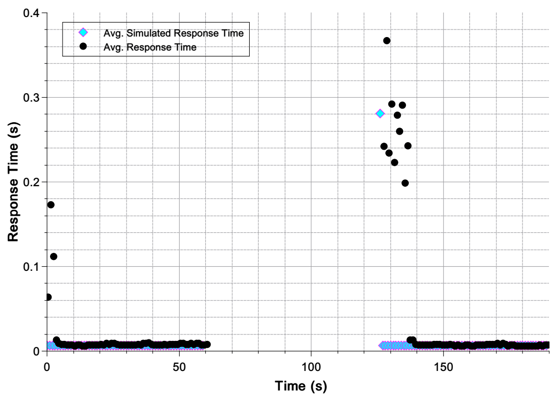

Figure 6.32: Response times of the Internal_Dependency endpoint during scenario #3.

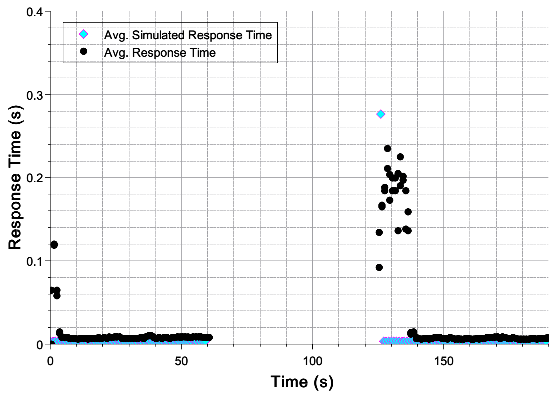

Figure 6.33: Response times of the DB_READ endpoint during scenario #3.

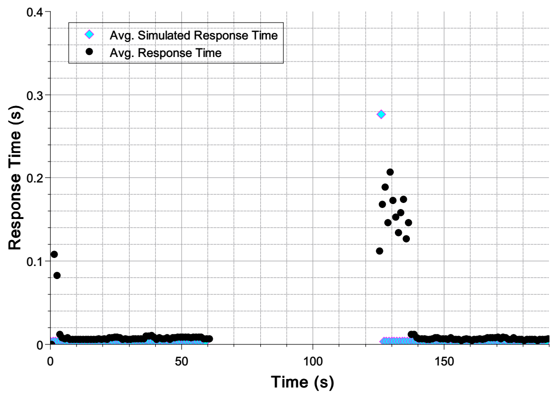

Figure 6.34: Response times of the DB_WRITE endpoint during scenario #3.

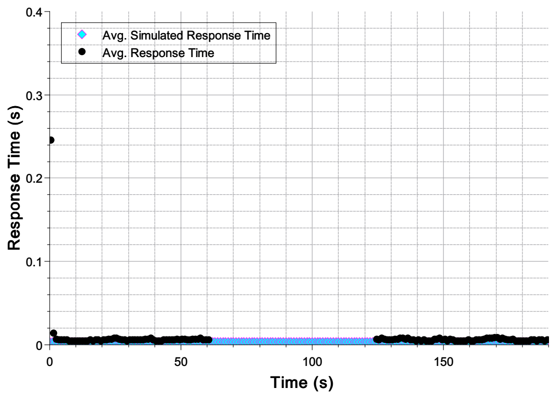

Figure 6.35: Response times of the Unaffected_Service endpoint during scenario #3.

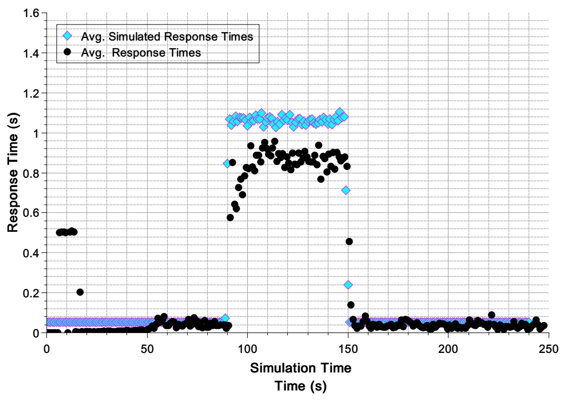

Figure 6.36: Response times of the External_Dependency endpoint during scenario #4.

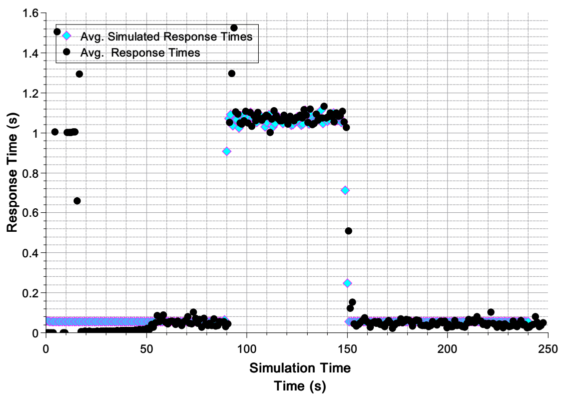

Figure 6.37: Response times of the Internal_Dependency endpoint during scenario #4.

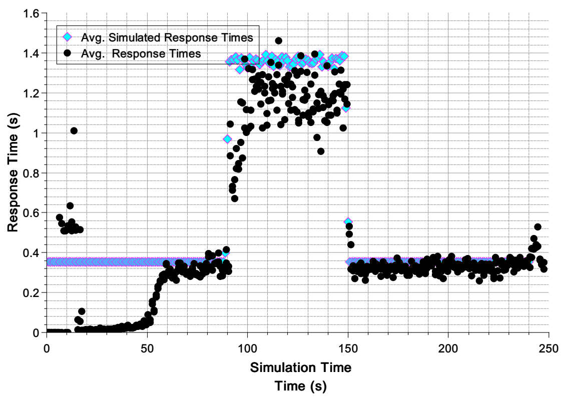

Figure 6.38: Response times of the DB_READ endpoint during scenario #4.

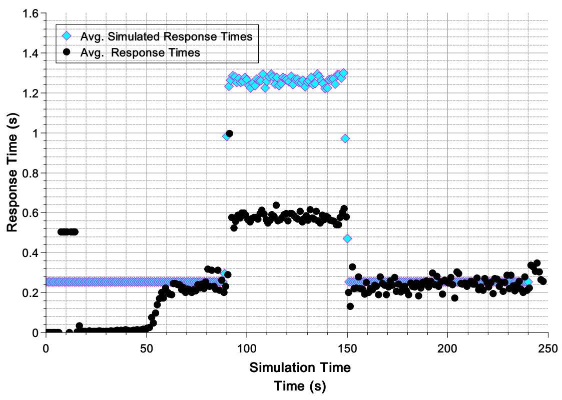

Figure 6.39: Response times of the DB_WRITE endpoint during scenario #4.

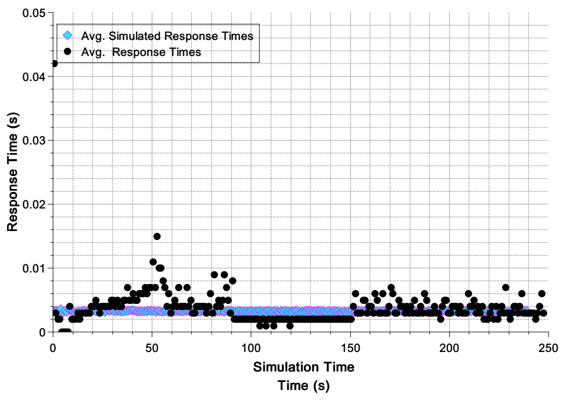

Figure 6.40: Response times of the Unaffected_Service endpoint during scenario #4.

6.6 Discussion of Results

Overall the results of the evaluation are mixed. *MiSim* 3.0 manages to simulate the newly implemented resilience patterns and workload generator accurately. However, the simulation of the four actual resilience scenarios lacked accuracy, specifically, when it came to varying workloads. In scenario #1 and #2 both the linear and exponential workload were not simulated at all, even tho the previous calibration showed the system correctly reacting to these stimuli. Therefore, it is concluded that the calibration of the architectural model was not accurately enough possible. Scenario #3 revealed, that the simulation of a failed service yields the expected results. However, the results of the actual system were slightly divergent since the used HTTP Load Generator monitors and sends requests slightly different to the system. Nevertheless, the response time results showed that *MiSim* is not accurate, when it comes to simulating cold start behavior. Lastly, scenario #4 had very mixed results. The general behavior of the network delay was seemingly correctly simulated, but results of the real data partially deviated heavily. Since the endpoint with the worst accuracy are also related to the database, it may be assumed that the response times of the real system are mainly influenced by the behavior of its in-memory database. This database-oriented behavior is currently not supported by *MiSim*.

In the end, it was shown that *MiSim* 3.0 is potentially capable of simulating scenario-based resilience scenarios accurately. But its accuracy strongly depends on the quality of the calibration.

6.7 Threads to Validity

There are several small threads to the validity of these results.

First, the participants of the code review are mostly part of research groups that work on similar topics. They were already superficially familiar with the *MiSim* simulator before the code review. Therefore, the results of the code review could be influenced by a slight confirmation or belief bias.

Regarding the evaluation, it could be argued that it was only done on small systems and therefore is lacking representatives. However, I would argue, that components of a microservice architecture are specifically designed to work in a rather isolated environment. Therefore, concentrating on the interactions between three or less services should be representative enough to draw general conclusions about systems' or patterns' behavior. Nevertheless, an actual evaluation of a larger production system is still necessary to examine this assumption.

The results and processed data shown in Section 6.5 are mostly based on only averages and are missing crucial additional data such as the relative standard deviation or the coefficient of variation. In the case of the average simulated response times, this was a deliberate choice, since the standard deviation was always very small (< 0.001 ms) and not visible in the graph at an appropriate scale. However, the response times of the specific endpoints of the real system, that were created by the HTTP Load generator, did not contain additional statistical data.

The HTTP Load Generator was run on the same local system as the calibration and evaluation systems. Other programs also ran parallel on the same machine. Therefore, the performance of the evaluation system might be slightly worse than they would be in practice. Additionally, the system was run as a docker-compose setup which can distorted performance based benchmarks [Tur21]. To compensate for this, the services were assigned a fixed amount of resources and CPU cores that are independent of each other. A similar situation occurred with the *MiSim* simulator itself. It was also run as isolated as possible on a machine, that also ran other programs and processes, which might influence its performance. However, since it utilizes DES this should not affect the simulation results.

Both the calibration and scenario evaluation were presented only based on a singular run, rather than on multiple averaged runs. This may leaves the consistency of the results open for discussion. However, during both these phases of the evaluation multiple runs of the experiments and scenarios were executed that had consisted results.

7 Conclusion and Future Work

The evaluation of *MiSim* 3.0 showed that it fullfills its requirements sufficiently, but there are still open points of improvement. Section 7.1 presents some potential research topics that took shape in the course of this thesis. At last, Section 7.2 gives a short summary of the thesis.

7.1 Future Work

This thesis leaves open a wide array of potential future works, that can contributed to the simulation of scenario-based chaos experiments. For example, this thesis concentrated on failure scenarios, however it would be interesting to see, how good other scenarios (e.g. scaling or deployment scenarios) can be simulated with the available simulators.

Also, one of the major drawbacks of *MiSim* that was revealed in this thesis is, that its is hard to calibrate accurately. In the future, the evaluation of tools for calibration automation (e.g. trace extraction or configuration exploring) could be a valuable continuation of this thesis.

Even tho *MiSim* 3.0 fulfills most of the requirements that are presented in Section 4.2, there are still other potential features that might be interesting for future work. These include:

- Other key resilience and performance concepts like deployment models, caching and rate limiting injection.
- Support for live hypothesis or SLO checking.
- Simulation of closed workloads.
- Automatic configuration exploration and optimization.

7.2 Conclusion

This thesis presents *MiSim* 3.0 as a simulator that it is capable of accurately simulating scenario-based chaos experiments. To achieve this, a list of 12 requirements for such a simulator was created by a group of subject matter experts.

Based on this, seven potential simulator candidates were found with a structured research. These included *SimuLizar* [BBM13], *Slingshot* [KB20], *DRACeo* [VDR+20], *MiSim* [BZG17], *μqsim* [ZGD19], *MuSim* [Flo] and a currently unnamed simulator [Kur]. With the exception of *Slingshot* and the unnamed one, these were selected for further examination. An evaluation of the simulators with respect to the aforementioned requirements revealed, that none of these simulators were capable of fully satisfying the needs of the stakeholders. Specifically, none of the simulators was able to provide sufficient combination of faultload and resilience simulation features.

Of the simulator candidates *MiSim* was selected as the most suitable simulator to be extended, since it already supported the simulation of some chaos injections and resilience patterns. Additionally, it is very light weight and already had some compatibility to existing tools [Bec18]. In the end *MiSim* was re-engineered to version 3.0 and now fulfills most of the original requirements. Besides some general improvements towards the quality of its architecture, it now supports the load balancer, autoscaler and retry pattern. The simulation of network delay injections is now also possible. Additionally, it does now also support the simulation of LIMBO [KHK14a] workload model. Furthermore, the simulator now supports a common scenario description of the Cambio project [1].

An evaluation of the new version showed that it can potentially and accurately simulate a real scenario-based chaos experiments on microservice architecture with various resilience mechanisms. All implemented patterns behave as expected during their isolated tests. The calibration process produced and a seemingly accurate architecture description. However, the accuracy of the simulation of a systems behavior under a growing workload is very bad. This hinted towards a currently major underlying problem of an inaccurate architecture model calibration. But, further investigations have to be done to find an exact cause.

[1] https://github.com/Cambio-Project/ScenarioDescriptor

Bibliography

[Ani21] N. Anil. *Implement HTTP call retries with exponential backoff with IHttpClientFactory and Polly policies*. https://docs.microsoft.com/de-de/dotnet/architecture/microservices/implement-resilient-applications/implement-http-call-retries-exponential-backoff-polly. 2021 (cit. on p. 6).

[BBC+17] G. Brataas, S. Becker, M. Cecowski, V. Čuček, S. Lehrig. "ScaleDL". In: *Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications*. Cham: Springer International Publishing, 2017, pp. 61–82 (cit. on p. 18).

[BBM13] M. Becker, S. Becker, J. Meyer. "SimuLizar: Design-Time Modeling and Performance Analysis of Self-Adaptive Systems". In: *Software Engineering 2013*. Ed. by S. Kowalewski, B. Rumpe. Bonn: Gesellschaft für Informatik e.V., 2013, pp. 71–84 (cit. on pp. 1, 11, 17, 63).

[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. 10, 11).

[BBS] M. Becker, S. Becker, C. Stier. *SimuLizar*. https://sdqweb.ipd.kit.edu/wiki/SimuLizar (cit. on p. 18).

[BCK03] L. Bass, P. Clements, R. Kazman. *Software Architecture in Practice*. 2nd ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2003 (cit. on pp. 9, 10, 13).

[BCNN10] J. Banks, J. Carson, B. Nelson, D. Nicol. *Discrete-Event System Simulation*. English. 3rd. Prentice Hall, 2010 (cit. on p. 11).

[Bec18] S. Beck. *Simulation-based Evaluation of Resilience Antipatterns in Microservice Architectures*. Bachelor's Thesis, University of Stuttgart. 2018 (cit. on pp. 14, 16, 17, 31, 64).

[Bec21] S. Beck. *Evaluating Human-Computer Interfaces for Specification and Comprehension of Transient Behavior in Microservice-based Software Systems*. Masters's Thesis, University of Stuttgart. 2021 (cit. on pp. 16, 38).

[BG20] L. Baresi, M. Garriga. "Microservices: The Evolution and Extinction of Web Services?" In: *Microservices: Science and Engineering*. Ed. by A. Bucchiarone, N. Dragoni, S. Dustdar, P. Lago, M. Mazzara, V. Rivera, A. Sadovykh. Cham: Springer International Publishing, 2020, pp. 3–28 (cit. on p. 3).

[BHK11] F. Brosig, N. Huber, S. Kounev. "Automated extraction of architecture-level performance models of distributed component-based systems". In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings*. Nov. 2011, pp. 183–192 (cit. on p. 22).

[BLB13] M. Becker, M. Luckey, S. Becker. “Performance Analysis of Self-Adaptive Systems for Requirements Validation at Design-Time”. In: *Proceedings of the 9th International ACM Sigsoft Conference on the Quality of Software Architectures*. June 2013 (cit. on p. 18).

[Blo20] T. Blogumas. *Learning Chaos Engineering — Exploring Fragile Software Systems*. `https://levelup.gitconnected.com/learning-chaos-engineering-exploring-fragile-software-systems-1e9f319098f`. May 2020 (cit. on p. 10).

[Bro15] M. Brooker. *Exponential Backoff And Jitter*. `https://aws.amazon.com/de/blogs/architecture/exponential-backoff-and-jitter/`. 2015 (cit. on pp. 5, 31).

[Bro19a] M. Brooker. *Amazon’s approach to building resilient services*. `https://aws.amazon.com/builders-library/amazon-approach-to-building-resilient-services/`. 2019 (cit. on p. 6).

[Bro19b] M. Brooker. *Timeouts, retries, and backoff with jitter*. `https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/`. 2019 (cit. on pp. 4–6).

[Bur19] R. Burch. “Defining and Evaluating Resilience”. In: *Resilient Space Systems Design An Introduction*. 1st ed. Boca Raton: CRC Press, Sept. 2019. Chap. 2, pp. 27–29 (cit. on p. 4).

[BZG17] S. Beck, C. Zorn, J. Günthör. *Simulation-based resilience prediction of microservice architectures*. Students’ Reserach Project, University of Stuttgart. 2017 (cit. on pp. 1, 11, 13, 14, 17, 63).

[CCY99] V. Cardellini, M. Colajanni, P. Yu. “Dynamic load balancing on Web-server systems”. In: *IEEE Internet Computing* 3.3 (1999), pp. 28–39 (cit. on p. 6).

[Cha18] Chaos Community. *Principles of Chaos Engineering*. 2018. (Visited on 03/28/2020) (cit. on pp. 1, 9).

[Cis21] Cisco Networking Academy. `https://www.netacad.com/courses/packet-tracer`. 2021 (cit. on p. 13).

[CRRB09] R. N. Calheiros, R. Ranjan, C. A. F. D. Rose, R. Buyya. *CloudSim: A Novel Framework for Modeling and Simulation of Cloud Computing Infrastructures and Services*. 2009 (cit. on pp. 1, 13).

[DB21] P. Djundik, M. Benjamins. *SteamDB — Steam Charts*. `https://steamdb.info/graph/`. 2021 (cit. on p. 8).

[DGL+17] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina. “Microservices: Yesterday, today, and tomorrow”. In: *Present and Ulterior Software Engineering*. Springer International Publishing, Nov. 2017. Chap. Microservice, pp. 195–216 (cit. on pp. 1, 3).

[Flo] L. Florio. *MuSim - The Microservices simulator*. `https://github.com/elleFlorio/mu-sim` (cit. on pp. 1, 16, 19, 63).

[Flo17] L. Florio. “Design and Management of Distributed Self-Adpative Systems”. PhD thesis. Politecnico di Milano, 2017, p. 139 (cit. on p. 19).

[Fou05] T. A. S. Foundation. `https://commons.apache.org/proper/commons-collections/apidocs/src-html/org/apache/commons/collections4/list/TreeList.html`. 2005 (cit. on p. 39).

[Fur15] K. Furuta. “Resilience Engineering”. In: *Reflections on the Fukushima Daiichi Nuclear Accident*. Cham: Springer International Publishing, Jan. 2015, pp. 435–454 (cit. on p. 4).

[Gea02] D. Geary. *Strategy for success*. https://www.infoworld.com/article/2074195/strategy-for-success.html. 2002 (cit. on pp. 27, 35).

[Gle18] A. M. Glen. *[DZone Research] Microservices Priorities and Trends*. https://dzone.com/articles/dzone-research-microservices-priorities-and-trends. July 2018 (cit. on p. 3).

[GVGB17] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, R. Buyya. “iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments”. In: *Software - Practice and Experience*. Vol. 47. 9. John Wiley and Sons Ltd, Sept. 2017, pp. 1275–1296 (cit. on p. 13).

[HADP18] A. van Hoorn, A. Aleti, T. F. Düllmann, T. Pitakrat. “ORCAS: Efficient Resilience Benchmarking of Microservice Architectures”. In: *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2018, pp. 146–147 (cit. on p. 17).

[HB21] D. Huber, M. Barón. *Scenario-based Resilience Analysisand Benchmarking of Real-worldIncidents in MicroserviceArchitectures*. Students’ Reserach Project, University of Stuttgart. 2021 (cit. on p. 14).

[Hol16] E. Hollnagel. *Resilience Engineering*. https://erikhollnagel.com/ideas/resilience-engineering.html. 2016. (Visited on 03/24/2020) (cit. on p. 4).

[Hol73] C. S. Holling. “Resilience and Stability of Ecological Systems”. In: *Annual Review of Ecology and Systematics* 4.1 (Nov. 1973), pp. 1–23. ISSN: 0066-4162 (cit. on p. 4).

[Hop00] C. Hopps. *Analysis of an Equal-Cost Multi-Path Algorithm*. Tech. rep. Nov. 2000 (cit. on p. 32).

[Idz10] J. Idziorek. “Discrete event simulation model for analysis of horizontal scaling in the cloud computing model”. In: *Proceedings of the 2010 Winter Simulation Conference*. 2010, pp. 3004–3014 (cit. on p. 32).

[Igo20] R. R. Igorevich. *Usage of Resilience4j Retry and CircuitBreaker together*. https://rusyasoft.github.io/java/2020/03/20/Usage-resilience4j-retry-cb/. 2020 (cit. on p. 31).

[JGC17a] M. Jacobs, D. Gross, B. Christensen. *Hystrix - Home*. https://github.com/Netflix/Hystrix/wiki. 2017 (cit. on p. 6).

[JGC17b] M. Jacobs, D. Gross, B. Christensen. *Hystrix - How It Works*. https://github.com/Netflix/Hystrix/wiki/How-it-Works. 2017 (cit. on pp. 4, 6, 17).

[KABC96] R. Kazman, G. D. Abowd, L. J. Bass, P. C. Clements. “Scenario-Based Analysis of Software Architecture”. In: *IEEE Softw.* 13.6 (1996), pp. 47–55 (cit. on pp. 9, 13).

[KB20] F. Klinaku, S. Becker. “The Slingshot Approach”. In: *Advances in Service-Oriented and Cloud Computing*. Cham: Springer International Publishing, 2020, pp. 158–165 (cit. on pp. 17, 18, 63).

[KBK12] D. Kliazovich, P. Bouvry, S. U. Khan. “GreenCloud: a packet-level simulator of energy-aware cloud computing data centers”. In: *The Journal of Supercomputing* 62.3 (Dec. 2012), pp. 1263–1283. ISSN: 0920-8542 (cit. on p. 13).

[KDK18] J. von Kistowski, M. Deffner, S. Kounev. “Run-time Prediction of Power Consumption for Component Deployments”. In: *Proceedings of the 15th IEEE International Conference on Autonomic Computing (ICAC 2018)*. Trento, Italy, Sept. 2018 (cit. on p. 48).

[KES+18] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, S. Kounev. “TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research”. In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2018, pp. 223–236 (cit. on p. 16).

[KHK14a] J. v. Kistowski, N. Herbst, S. Kounev. “LIMBO: A Tool for Modeling Variable Load Intensities”. In: *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*. ICPE ’14. Dublin, Ireland: Association for Computing Machinery, 2014, pp. 225–226 (cit. on pp. 14, 18, 21, 34, 64).

[KHK14b] J. v. Kistowski, N. R. Herbst, S. Kounev. “Modeling Variations in Load Intensity over Time”. In: LT ’14. Dublin, Ireland: Association for Computing Machinery, 2014, pp. 1–4 (cit. on p. 34).

[KKC00] R. Kazman, M. Klein, P. Clements. *ATAM: Method for Architecture Evaluation*. Tech. rep. Carnegie Mellon University, Pittsburgh, PA 15213: Software Engineering Institute, 2000 (cit. on p. 13).

[KS18] S. Kaur, T. Sharma. “Efficient load balancing using improved central load balancing technique”. In: *2018 2nd International Conference on Inventive Systems and Control (ICISC)*. 2018, pp. 1–5 (cit. on p. 6).

[Kur] R. Kurr. *kurron/microservice-simulator*. https://github.com/kurron/microservice-simulator (cit. on pp. 17, 63).

[KWK+20] D. Kesim, L. Wagner, J. von Kistowski, S. Frank, A. Van Hoorn, A. Hakamian. *Scenario-based Resilience Evaluation and Improvement of Microservice Architectures: An Experience Report*. Universität Stuttgart, 2020 (cit. on pp. 1, 13, 14, 16, 49).

[Lan] M. Langhammer. *Extract*. https://sdqweb.ipd.kit.edu/wiki/Extract (cit. on p. 22).

[LBG+15] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, N. Medvidovic. “An Empirical Study of Architectural Change in Open-Source Software Systems”. In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 2015, pp. 235–245 (cit. on p. 22).

[LE15] S. Lehrig, H. Eikerling. “Analyzing Cost-Efficiency of Cloud Computing Applications with SimuLizar”. In: Nov. 2015 (cit. on p. 18).

[Led] A. Ledenev. https://github.com/alexei-led/pumba (cit. on p. 56).

[LF14] J. Lewis, M. Fowler. *Microservices*. https://www.martinfowler.com/articles/microservices.html. 2014 (cit. on p. 3).

[LL13] J. Ludewig, H. Lichter. *Software Engineering. Grundlagen, Menschen, Prozesse, Techniken*. German. Ed. by U. Zimpfer. 3rd. Ringstraße 19B, 69115 Heidelberg, Germany: dpunkt.verlag, 2013 (cit. on p. 15).

[LS20] M. Loukides, S. Swoyer. *Microservices Adoption in 2020 – O’Reilly*. https://www.oreilly.com/radar/microservices-adoption-in-2020/. 2020 (cit. on p. 3).

[Lup17] E. Lupander. *Go Microservices, Part 11: Hystrix and Resilience*. https://callistaenterprise.se/blogg/teknik/2017/09/11/go-blog-series-part11/. 2017 (cit. on pp. 26, 31, 32).

[MACG20] N. C. Mendonca, C. M. Aderaldo, J. Camara, D. Garlan. "Model-Based Analysis of Microservice Resiliency Patterns". In: *2020 IEEE International Conference on Software Architecture (ICSA)*. 2020, pp. 114–124 (cit. on pp. 4–7).

[Mat09] R. Matarneh. "Self-Adjustment Time Quantum in Round Robin Algorithm Depending on Burst Time of the Now Running Processes". In: *American Journal of Applied Sciences* 6 (Oct. 2009) (cit. on p. 35).

[Mer11] P. Merkle. *Comparing Process-and Event-oriented Software Performance Simulation*. Master Thesis. Masters's Thesis, Karlsruhe Institue of Technology. 2011 (cit. on p. 18).

[MJ09] A. Madni, S. Jackson. "Towards a Conceptual Framework for Resilience Engineering". In: *IEEE Systems Journal* 3.2 (June 2009), pp. 181–191. ISSN: 1932-8184 (cit. on p. 4).

[MWW12] D. Meisner, J. Wu, T. F. Wenisch. "BigHouse: A simulation infrastructure for data center systems". In: *2012 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, Apr. 2012, pp. 35–45 (cit. on p. 13).

[NCCA14] M. A. Netto, C. Cardonha, R. L. Cunha, M. D. Assuncao. "Evaluating Auto-scaling Strategies for Cloud Computing Environments". In: *2014 IEEE 22nd International Symposium on Modelling, Analysis Simulation of Computer and Telecommunication Systems*. 2014, pp. 187–196 (cit. on p. 33).

[New14] S. Newman. *Building Microservices*. Ed. by M. Loukides, B. MacDonald. First Edit. Sebastopol: O'Reilly Media, Inc., 2014 (cit. on p. 3).

[NGN17] Z. Nikdel, B. Gao, S. W. Neville. "DockerSim: Full-stack simulation of container-based Software-as-a-Service (SaaS) cloud deployments and environments". In: *2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. Vol. 2017-Janua. IEEE, Aug. 2017, pp. 1–6 (cit. on pp. 1, 13).

[NLL18] Y. Niu, F. Liu, Z. Li. "Load Balancing Across Microservices". In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 2018, pp. 198–206 (cit. on pp. 6, 32).

[nsn21] nsnam. *ns-3 Network Simulator*. https://www.nsnam.org/. 2021 (cit. on p. 11).

[Nyg07] M. Nygard. *Release It! Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007 (cit. on p. 4).

[Pac18] Packetstorm Communications. *Network Simulation - PacketStorm*. 2018. (Visited on 05/13/2021) (cit. on p. 13).

[PAP+18] I. Prachitmutita, W. Aittinonmongkol, N. Pojjanasuksakul, M. Supattatham, P. Padungweang. "Auto-scaling microservices on IaaS under SLA with cost-effective framework". In: *2018 Tenth International Conference on Advanced Computational Intelligence (ICACI)*. 2018, pp. 583–588 (cit. on p. 8).

[PL99] B. Page, T. Lechler. *DESMO-J*. http://desmoj.sourceforge.net/. 1999 (cit. on pp. 11, 17).

[RA20] G. Rauch, S. Augsten. *Was ist "Chaos Engineering"*. https://www.dev-insider.de/was-ist-chaos-engineering-a-971111/. Oct. 2020 (cit. on p. 9).

[Res] Resilience4j Authors. *Retry Documenation*. https://resilience4j.readme.io/docs/retry (cit. on p. 4).

[RHB+17] C. Rosenthal, L. Hochstein, A. Blohowaik, N. Jones, A. Basiri. *Chaos Engineering. Building Confidence in System Behavior through Experiments*. Ed. by B. Anderson, C. Cole, C. Edwards. 1st. Sebastopol: O'Reilly, 2017 (cit. on pp. 10, 11).

[RPT19] M. I. Rahman, S. Panichella, D. Taibi. "A curated Dataset of Microservices-Based Systems". In: *CoRR* abs/1909.03249 (2019) (cit. on p. 16).

[RSW+20] M. Reimann, S. Seifermann, M. Walter, R. Heinrich, T. Bureš, P. Hnětynka. "Towards Language-Agnostic Reuse of Palladio Quality Analyses". In: *Proceedings of the 11th Symposium on Software Performance 2020*. 2020, p. 3 (cit. on p. 18).

[Sei21] S. Seifermann. https://sdqweb.ipd.kit.edu/wiki/PCM_5.0. Feb. 2021 (cit. on p. 17).

[SGG05] A. Silberschatz, P. B. Galvin, G. Gagne. *Operating System Concepts*. Ed. by K. Santor. 7th. 2005, p. 946 (cit. on pp. 26, 35).

[Sha20] R. Sharma. https://www.linkedin.com/pulse/server-vs-client-side-load-balancing-ramit-sharma?trk=read_related_article-card_title. Mar. 2020 (cit. on p. 6).

[Sho04] J. Shore. "Fail Fast". In: *IEEE Software* 21.5 (2004), pp. 21–25 (cit. on p. 6).

[Sma21] M. Smallcombe. *Chaos Testing: Resilience Testing Gone Wild*. https://www.xplenty.com/blog/what-is-chaos-testing/. Jan. 2021 (cit. on p. 9).

[SSS08] S. Sharma, S. Singh, M. Sharma. "Performance Analysis of Load Balancing Algorithms". In: *International Journal of Civil and Environmental Engineering* 2.2 (2008), pp. 367–370 (cit. on p. 6).

[Str15] M. Strittmatter. https://sdqweb.ipd.kit.edu/wiki/PCM_AddOns. Apr. 2015 (cit. on p. 18).

[TI14] L. Tomov, V. Ivanova. "Teaching good practices in software engineering by counterexamples". In: *Proceedings of the 10th Annual Computer Science And Education In Computer Science Conference*. July 2014 (cit. on p. 24).

[Tur21] I. Turner-Trauring. https://pythonspeed.com/articles/docker-performance-overhead/. May 2021 (cit. on p. 61).

[Val21] Valve Corporation. *Steam*. https://store.steampowered.com/about/. 2021 (cit. on p. 8).

[VDR+20] H. H. A. Valera, M. Dalmau, P. Roose, J. Larracoechea, C. Herzog. "DRACeo: A smart simulator to deploy energy saving methods in microservices based networks". In: *2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. Vol. 2020-Septe. IEEE, Sept. 2020, pp. 94–99 (cit. on pp. 1, 13, 16, 19, 63).

[VHK15] C. Vögele, A. van Hoorn, H. Krcmar. "Automatic Extraction of Session-Based Workload Specifications for Architecture-Level Performance Models". In: *Proceedings of the 4th ACM/SPEC International Workshop on Large-Scale Testing, in Conjunction with ICPE 2015*. Feb. 2015, pp. 5–8 (cit. on p. 22).

[Vuč20] J. Vučković. “You Are Not Netflix”. In: *Microservices: Science and Engineering*. Ed. by A. Bucchiarone, N. Dragoni, S. Dustdar, P. Lago, M. Mazzara, V. Rivera, A. Sadovykh. Cham: Springer International Publishing, 2020, pp. 333–346 (cit. on p. 3).

[Wag] L. Wagner. *Artifacts*. `https://zenodo.org/record/4889805` (cit. on p. 37).

[YF14] L. Yazdanov, C. Fetzer. “Lightweight Automatic Resource Scaling for Multi-tier Web Applications”. In: *2014 IEEE 7th International Conference on Cloud Computing*. 2014, pp. 466–473 (cit. on p. 8).

[YKXY19] R. Yu, V. T. Kilari, G. Xue, D. Yang. “Load Balancing for Interdependent IoT Microservices”. In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. 2019, pp. 298–306 (cit. on p. 6).

[ZGD19] Y. Zhang, Y. Gan, C. Delimitrou. “µqSim: Enabling Accurate and Scalable Simulation for Interactive Microservices”. In: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Mar. 2019, pp. 212–222 (cit. on pp. 11, 13, 16, 18, 63).

[Zor21] C. Zorn. *Interactive Elicitation of Resilience Scenarios in Microservice Architectures*. Masters’s Thesis, University of Stuttgart. 2021 (cit. on pp. 14, 16, 38).

[ZPX+18] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, W. Zhao. “Benchmarking Microservice Systems for Software Engineering Research”. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ICSE ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 323–324 (cit. on p. 16).

[ZQL+16] S. Zhang, Z. Qian, Z. Luo, J. Wu, S. Lu. “Burstiness-Aware Resource Reservation for Server Consolidation in Computing Clouds”. In: *IEEE Transactions on Parallel and Distributed Systems* 27.4 (2016), pp. 964–977 (cit. on p. 8).

All links were last followed on May 30th, 2021.

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.

Plochingen, 01.06.2021, [illegible]

place, date, signature