MiSim: A Simulator for Resilience Assessment of Microservice-based Architectures

Sebastian Frank[1,2], Lion Wagner[2], Alireza Hakamian[2], Martin Straesser[3], André van Hoorn[1]
[1]University of Hamburg, Hamburg, Germany
[2]University of Stuttgart, Stuttgart, Germany
[3]University of Würzburg, Würzburg, Germany
{sebastian.frank, andre.van.hoorn}@uni-hamburg.de, mir-alireza.hakamian@iste.uni-stuttgart.de,
martin.straesser@uni-wuerzburg.de

***Abstract*—Increased resilience compared to monolithic architectures is both one of the key promises of microservice-based architectures and a big challenge, e.g., due to the systems' distributed nature. Resilience assessment through simulation requires fewer resources than the measurement-based techniques used in practice. However, there is no existing simulation approach that is suitable for a holistic resilience assessment of microservices comprised of (i) representative fault injections, (ii) common resilience mechanisms, and (iii) time-varying workloads. This paper presents *MiSim* — an extensible simulator for resilience assessment of microservice-based architectures. It overcomes the stated limitations of related work. *MiSim* fits resilience engineering practices by supporting scenario-based experiments and requiring only lightweight input models. We demonstrate how *MiSim* simulates (1) common resilience mechanisms — i.e., circuit breaker, connection limiter, retry, load balancer, and autoscaler — and (2) fault injections — i.e., instance/service killing and latency injections. In addition, we use TeaStore, a reference microservice-based architecture, aiming to reproduce scaling behavior from an experiment by using simulation. Our results show that *MiSim* allows for quantitative insights into microservice-based systems' complex transient behavior by providing up to 25 metrics.**

Keywords*—*microservices; resilience; scenarios; simulation

I. Introduction

With the growing popularity of the microservice-based architectural style [1], there is a need for an effective resilience assessment of such systems. Laprie [2] defines resilience as "the persistence of service delivery that can justifiably be trusted when facing changes". Resilience assessment in microservice-based architectures is crucial as configurations of such architectures constantly change, e.g., due to frequent (re-)deployments facilitated by modern DevOps processes.

Resilience assessment is often done in a production environment using so-called chaos experiments by injecting faultloads and observing the behavior of the system quality [3]. While producing representative results, chaos experiments often require (1) a significant investment of time and effort in experiment set-up, (2) a significant amount of execution costs, and (3) in-depth knowledge regarding the underlying technology stack. Simulating chaos experiments is an alternative approach to running experiments in a real cloud environment. Today, many simulators for distributed and service-oriented architectures exist. Popular examples are *SimuLizar* [4], *DRACeo* [5], *BigHouse* [6], *μqsim* [7], *PacketStorm* [8], *iFogSim* [9] and *GreenCloud* [10]. Most of these focus on performance analysis using simulation or solving queuing models (analytically). However, none satisfies the following requirements for resilience assessment: (1) supporting common resilience mechanisms, which enable architectures to handle failures gracefully and recover from them, and (2) simulating multiple typical failure injections such as killing a service instance.

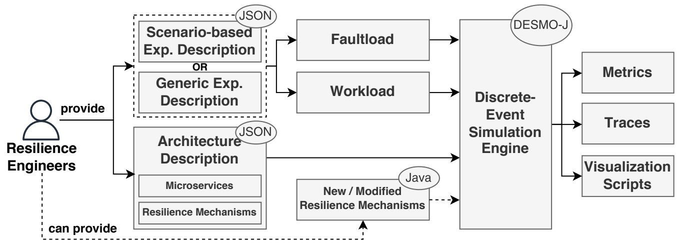


Figure 1: *MiSim* overview

We present *MiSim* — a simulator specialized in the simulation of resilience mechanisms and chaos experiments. Figure 1 shows the workflow, inputs, and outputs of *MiSim*. Resilience engineers must provide experiments in the generic description format or, more intuitively, as scenarios. Scenarios are means for requirements specification and evaluation, which have been used by qualitative architecture evaluation methods such as the Architecture Tradeoff Analysis Method [11]. The experiment descriptions contain information about the workload and faultload. *MiSim* also requires an architecture description of the system under test, which contains information about the services, interdependencies, and the implemented resilience mechanisms. *MiSim* implements resilience mechanisms based on existing pattern descriptions in literature [11] [12] and practice [13] but can also use custom extensions and modifications. Furthermore, it has no external dependencies, such as Platform as a Service (PaaS), submodules, or libraries. *MiSim* provides metrics, simulation traces, and visualization scripts that enable further processing and inspection of the results.

While *MiSim* may be used to simulate other types of distributed systems, we explicitly focus on microservice-based systems. Due to their high degree of independence regarding development and technologies, microservice-based systems must be carefully designed to be resilient. However, they can also reach a high degree of resilience when designed correctly, making resilience simulation of microservice-based systems particularly relevant. Therefore, *MiSim* implements common resilience mechanisms for microservice-based systems, e.g., as provided by the resilience library Resilience4j [14]. Further, *MiSim* can simulate mechanisms like autoscaling, a key benefit of microservice-based systems according to Newman [15].

We simulate three scenario-based experiments on an architecture description obtained from an industry system [16] to evaluate the simulator. The scenarios require and trigger most of the implemented resilience mechanisms. In a further experiment, we partially replicate an experimental setting using the benchmarking application TeaStore [17] to compare the actual scaling behavior against the scaling behavior simulated by *MiSim*. All results are reproducible, and we provide both the simulator code [18] and the evaluation data [19].

In the scenario-based experiments, we show that the simulator is able to mirror the behavior of (1) common resilience mechanisms — i.e., circuit breaker, retry, and autoscaler — and (2) chaos injections — i.e., instance/service killing and latency injections. The TeaStore experiment shows that the actual and the simulated scaling behaviors are similar.

We summarize our contributions as follows:

- The extensible *MiSim* simulator for resilience assessment of microservice-based architectures.
- Default implementations for common resilience mechanisms and strategies used in microservice-based systems.
- A lightweight architecture description for resilience assessment.
- A demonstration of *MiSim*'s usefulness and applicability.

The remainder of this paper is structured as follows. In Sections 2 and 3, we summarize the foundations of this work and compare *MiSim* to other microservice simulators. Section 4 provides an overview of *MiSim*'s general simulation process. Sections 5 and 6 elaborate on the input models and potential simulation results of *MiSim* supported by a running example. Section 7 evaluates *MiSim*'s capabilities to simulate different resilience mechanisms in three selected scenarios and its usefulness in predicting actual scaling behavior. Conclusions are drawn in Section 8.

II. Background

This section explains essential concepts regarding resilience patterns, chaos engineering, and scenario-based quality requirements specification.

A. Resilience Mechanisms

Cataloging architectural patterns has been discussed in both academic literature [11] [12] and practice [13]. According to Bass et al. [11], patterns (1) exist to achieve a particular quality attribute, and (2) comprise tactics to augment the patterns. We use the notion of *resilience mechanism* to refer to both patterns and tactics used in the context of fault tolerance and the design of resilient systems. The main reason for choosing the name resilience mechanism is that sometimes tactics and patterns have been used interchangeably in existing pattern collections. For example, retry is listed as a pattern in the Microsoft Collection [13], but as a tactic by Bass et al. [11]. In the following, we describe the resilience mechanisms supported by *MiSim*.

Circuit Breaker [13]. In distributed environments, service calls can fail, e.g., due to network issues. The caller's resources get wasted on a non-responding service, which impacts the service's quality. The circuit breaker pattern solves the problem of cascading failures by disallowing further calls to the target service if a failure threshold is reached.

Retry [13]. Transient failures are common in a distributed environment. Retrying a failed operation ensures continued operations despite such transient failures.

Autoscaling [11]. In cloud-based distributed systems, it is common to design the system to use additional resources autonomously. Automatic scaling can be vertical, meaning the addition/removal of resources to physical units, and horizontal, meaning the addition/removal of resources to logical units.

Load Balancing [11]. As an application of resource scheduling, load balancing ensures that one resource is not overloaded while the other is idle.

Connection Limiter [13]. The bulkhead pattern suggests grouping application elements into pools so that when one group fails, the other stays unaffected. We implemented thread pooling such that the number of possible requests to be sent by a service is limited. Because we do not implement the grouping of all application elements, we use connection limiter as the pattern's name throughout the paper.

B. Chaos Engineering

Chaos Engineering is a set of disciplines for evaluating a system's capability to withstand disturbances in production [3]. The core idea is to design and execute experiments. The experiments describe injecting failures and monitoring system behavior regarding, e.g., a performance or availability quality metric. Based on an initial hypothesis, the test team assesses system resilience based on collected measurements of the particular metrics from failure injection time until the distribution of measurements reaches a steady state. The experiments can be executed manually or automated by frameworks like Chaos Toolkit [20] or Gremlin [21].

C. Scenario-based Quality Requirements Specification

In software engineering, scenarios describe quality requirements in an unambiguous and testable way [11]. In this work, we focus on resilience scenarios, which are scenarios that concentrate on resilience metrics.

Chaos experiments provide a means to describe and execute resilience scenarios against the system implementation. Figure 2 shows the mapping from elements of the scenario description to the chaos experiment description. For example, stimulus in the scenario description is mapped to the specification of workload and faultload in chaos experiments. Moreover, the response measure is mapped to hypothesis and steady-state metrics in chaos experiments. In our preceding

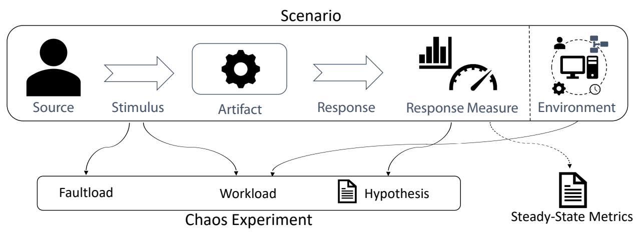


Figure 2: Transposing resilience scenarios to chaos experiments [16]

industrial case study [16], we showed that scenarios are suitable for specifying and quantitatively evaluating resilience requirements in microservice-based applications. It is shown that the hypotheses in chaos experiments and the response measure both describe the system's expected behavior with dedicated steady-state metrics. Moreover, a combination of stimulus, environment, and artifact describes the expected system execution context.

III. Related Work

We used the statement *simulator AND (microservice OR microservices)* in Google Scholar, Google Search, and CORE to collect simulators tailored for resilience assessment of microservice-based architectures. The simulators we found are *DRACeo* [5], *iFogSim* [9], *µqsim* [7], and *MuSim* [22]. In addition, we added *SimuLizar* [4] to the list of simulators for comparison. *SimuLizar* has been proposed for the performance evaluation of self-adaptive software systems. *SimuLizar* interprets the Palladio Component Model (PCM) [23], a state-of-the-art model-driven approach for predicting quality attributes. We found and included *SimuLizar* as PCM is seminal work in architecture-based quality prediction.

In further search, we found various simulators for distributed or service-oriented architectures, e.g., *BigHouse* [6] and *GreenCloud* [10] for the simulation of data center systems, and *PacketStorm* [8] for the simulation of networks. As these simulators are not designed for resilience assessment and focus primarily on cloud infrastructure, we excluded them from the comparison. We also excluded *iFogSim* [9], as it focuses on the internet of things (IoT) and mobility simulation.

Each simulator has unique characteristics, as visualized in Table I, based on a list of features that simulators shall support for resilience assessment of microservice-based architectures. We considered technical, domain-specific, and compatibility viewpoints. Microservice-based architectures can be complex regarding size, and an analytical solution for the underlying performance model either does not scale well or may not exist. Therefore, most simulators (including *MiSim*) are based on Discrete Event Simulation (F1).

Chaos Experiments are commonly used to assess resilience. Therefore, a simulator for resilience assessment should allow for the simulation of faultloads (F4) as available in state-of-the-art chaos engineering tools, e.g., Chaos Toolkit [20]. However, besides *MiSim*, only *DRACeo* partially supports faultloads like killing instances and injecting delays. More

TABLE I: A comparison of *MiSim* to similar simulators ([4], [5], [7], [22]) for features relevant in the context of resilience simulation

Features	*SimuLizar*	*DRACeo*	*µqsim*	*MuSim*	*MiSim*
F1 Discrete Event Simulation	Y	Y	Y	N	Y
F2 Headless Mode	N	N	Y	N	Y
F3 Output Metrics					
F3.1 Response Times	Y	Y	Y	Y	Y
F3.2 Error Rates	N	N	N	N	N
F3.3 Throughput	Y	N	Y	N	N
F3.4 Queue Lengths	Y	N	Y	N	Y
F3.5 Execution Traces	N	N	N	N	Y
F4 Chaos Toolkit Faultloads	N	∼[3]	N	N	Y
F5 Varying Workload	Y[1]	N	N	Y[2]	Y
F6 Resilience Mechanisms					
F6.1 Self-Healing (Restart)	N	Y	N	Y[2]	N
F6.2 Autoscaling	Y[1]	Y	N	Y[2]	Y
F6.3 Load Balancing	Y[1]	Y	Y	Y	Y
F6.4 Retry	Y[1]	N	N	N	Y
F6.5 Circuit Breaker	N	N	N	N	Y
F6.6 Rate Limiter	Y[1]	N	N	N	N
F6.6 Connection Limiter	N	N	N	N	Y
F6.7 Caching	Y[1]	N	N	N	N
F7 Scenarios	N	N	N	N	Y

[1] Supported as part of the Palladio Component Model (PCM).
[2] Supported due to the usage of a PaaS/Docker.
[3] Supports instance/service/device killing.

complex and realistic scenarios also require support for dynamic workloads (F5), e.g., load spikes. To our knowledge, *DRACeo* and *MuSim* do not have this capability. Furthermore, to assess the influence of resilience mechanisms (F6) on the system's resilience, the simulator must be capable of simulating these mechanisms. In that regard, *SimuLizar* and *MiSim* are similarly powerful but consider different mechanisms. Circuit breakers and connection limiters are only available in *MiSim*.

A headless mode (F2), i.e., availability of a CLI or API, eases the automation of experiments and the integration into resilience assessment workflows. However, only *MiSim* and *MuSim* have a headless mode, while *SimuLizar* is coupled to the Eclipse Platform and *DRACeo* just provides a graphical user interface. Furthermore, the available output metrics (F3) determine what requirements can be verified. *MiSim* provides a broad range of up to 25 metrics (see Section VI), even containing full execution traces, which are not provided by the other simulators. *MiSim* is also the only simulator supporting scenarios (F7) for the description of experiments.

In conclusion, *MiSim*'s capabilities to simulate multiple resilience mechanisms, support typical chaos injections, and its low overhead in simulation and modeling are unique among the investigated simulators.

IV. MiSim Architecture: Process View

This section presents the process view of the *MiSim* architecture, which supports the features introduced in Section III.

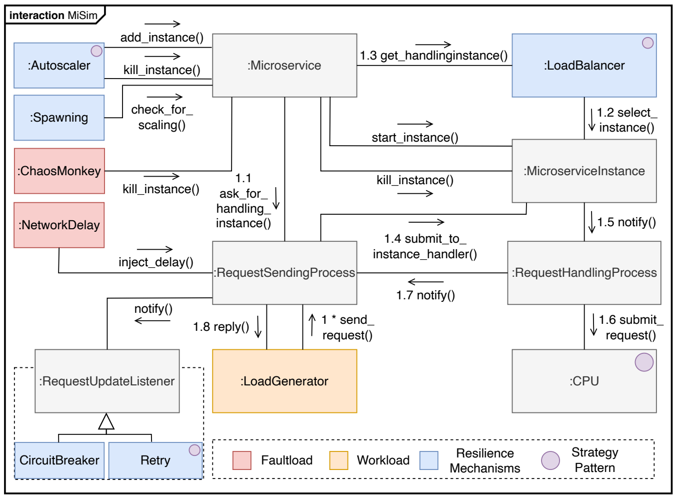


Figure 3: Interaction between conceptual objects in *MiSim*

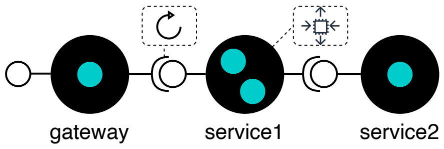


Figure 4: The running example's architecture: The *gateway*-*service1*-dependency is equipped with a retry and *service1* with a utilization-based autoscaler.

Our architecture follows an event-driven style as we rely on the discrete-event simulation framework DESMO-J [24].

Figure 3 visualizes the interaction between conceptual objects in the *MiSim* architecture in a UML communication diagram. Note that this is just a simplified representation of the actual implementation. The *LoadGenerator* sends a request to the *RequestSendingProcess*, which can be repeated by an unspecified number shown as $*$. The *RequestSendingProcess* looks for the instance that handles the request by sending a message to a *Microservice*. The *Microservice* uses the *LoadBalancer* to choose the responsible instance to handle the user request. After that, the *RequestSendingProcess* has the handling instance, and the object sends the user request to the *RequestHandlingProcess*, shown by the message *submit_to_instance_handler*. The *RequestHandlingProcess* executes the request by passing it to the Central Processing Unit (*CPU*).

For the *CPU*, by default, we simulate the *Round Robin* scheduling algorithm, where each process is cyclically assigned a time slot. *Round Robin* scheduling is commonly used in multi-tasking environments for fair execution of processes. Furthermore, *MiSim* supports *First Come First Serve*, *Multi Level Queue*, *Shortest Job Next* scheduling algorithms. Further extension is possible through strategy patterns. The *RequestSendingProcess* is notified after the execution of the user request. The *RequestSendingProcess* notifies registered listeners, e.g., a *CircuitBreaker* or *Retry*, when handling a request. During the process, faults can be injected by *ChaosMonkey* and *NetworkDelay*. Furthermore, the *Autoscaler* can automatically scale the system based on the current situation, while *Spawning* allows manually creating new instances.

V. Scenario and System under Test Descriptions

In the following, we introduce a particular system under test as a running example for this paper. Next, we describe *MiSim*'s two required types of input models. The first input is an architecture model that describes the structure and performance properties of the system under test. The second input is an experiment model that describes workloads, faultloads, and metadata for the respective run. Finally, we briefly discuss the available extension points of *MiSim*.

A. Running Example

To demonstrate the capabilities of *MiSim*, we use a running example throughout this paper. We based the running example on a proxy system for an existing industry system in an early stage of development [16]. The original microservice-based system's purpose is to support use cases in the domain of payment accounting. It is developed to replace a legacy system that handles up to 13 million calculation requests in peak times. The investigated part of the microservice-based system currently consists of seven different services. The proxy system is a real implementation with the same technology stack as the actual system, with reduced complexity regarding the number of services. It is provided by the company developing the actual system and being used to evaluate the system's quality. We converted the proxy's architecture and resilience characteristics into a description that is interpretable by *MiSim*.

As depicted in Figure 4, the running example system consists of three services: *gateway*, *service1*, and *service2*. The *gateway* has a single instance, and the dependency on *service1* is equipped with a retry pattern. By default, *service1* has two instances and a utilization-based autoscaler to start new instances on demand. *service2* only has a single instance and no resilience mechanisms. The services are linearly dependent on each other. A single request to the *gateway* causes a single request to *service1*, which sends another request to *service2*.

B. Modeling System Architectures

MiSim architecture models describe mostly static properties of a system's architecture. They offer the opportunity to define the network latency in the system and services with up to five properties. These properties are listed in Table II and contain predominantly technical information, such as the number of default instances and implemented resilience mechanisms.

Further, a service definition requires a description of the supported service endpoints. This is done within an `operations` array and requires the properties listed in Table III for each description. In particular, each operation requires the specification of a unique name and a computational demand for executing the operation. Each operation can also describe dependencies on other operations. Each of these dependencies can be probabilistic to account for behavior that is not executed every time. Such probabilities can

TABLE II: Supported service properties

Property Name	Description
name	Name of the service.
loadbalancer	Load balancing strategy for incoming requests.
instances	Number of default instances
capacity	Calculation capacity that each service instance has.
service_patterns	List of resilience mechanisms of the whole service (e.g. a load balancer).
instance_patterns	List of resilience mechanisms that will be created for each instance (e.g. circuit breaker).
operations	List of operations/endpoint that a service provides.

TABLE III: Supported operation properties

Property Name	Description
name	Name of the operation.
demand	Computational demand for processing the operation call.
dependencies	List of dependencies that this operation needs to complete before it is considered completed itself. A dependency description defines a target endpoint and can set an activation probability.

easily be computed from measurements of existing software systems. However, probabilistic dependencies alone are not sufficient to model actual system behavior. For example, the SAGA pattern [25] requires an operation only to perform compensation actions when it fails to execute its normal behavior. To support such cases, *MiSim* allows the nesting of dependencies with (probabilistic) alternative and loop dependencies as intermediate dependency types.

Listing 1 shows the architecture description of the *gateway* service, which was introduced earlier as part of the running example. As described earlier, this gateway only has one instance and a single endpoint called *API_Endpoint* by default. This operation has a computational demand of 1 and a non-nested, non-probabilistic dependency on the *dependentCalculation* endpoint of *service1*. Combining this with the set instance capacity of 10000 results in a throughput of 10000 requests per simulation time unit (STU). Further, the description defines the properties of the used retry pattern. In this case, it configures an exponential back-off with the delay formula $d(t) = \texttt{min}(0.1 \cdot 3^{(t-1)}, 7)$, with t being the number of failed tries. The exponential back-off strategy is a common approach to avoid retry storms and has been implemented in retry pattern implementations such as Resilience4j [14]. The mentioned exponential back-off delay and its importance are further detailed in the Amazon Web Services blog [26].

As shown in Table II, a service in *MiSim* can support two categories of resilience mechanisms: service patterns and instance patterns. Service patterns act on all service instances and modify the owning service itself. By default, *MiSim* only supports a simple utilization-based autoscaler as a service pattern. Instance-owned patterns are instantiated for each instance separately. A concrete list of all currently available patterns is shown in Table IV. The properties of each supported pattern

```
{
  "name": "gateway",
  "instances": 1,
  "capacity": 10000,
  "loadbalancer_strategy": "even",
  "operations": [
    {
      "name": "API_Endpoint",
      "demand": 1,
      "dependencies": [
        {
          "service": "service1",
          "operation": "dependentCalculation"
        }
      ]
    }
  ],
  "patterns": [
    {
      "type": "retry",
      "strategy": {
        "type": "exponential",
        "config": {
          "baseBackoff": 0.1,
          "maxBackoff": 7,
          "base": 3
        }
      }
    }
  ]
}
```

Listing 1: Architecture description of the *gateway* service

TABLE IV: Supported resilience mechanisms

Pattern	Category	Purpose
Retry	Instance Pattern	Resends failed internal requests after a configurable delay.
Circuit Breaker	Instance Pattern	Cleanly breaks a connection if it becomes unhealthy. Automatically reopens it periodically for status checks.
Connection Limiter	Instance Pattern	Restrict the maximum number of open connections between two instances/services.
Load Balancer	Instance Pattern	Handles the distribution of outgoing messages.
Autoscaler	Service Pattern	Increases or decreases the instances count of a service.

can be configured as shown in the retry definition in Listing 1.

C. Modeling Experiments

MiSim experiment descriptions are input files that describe the actual simulation process. They always contain metadata such as a name, description, and output location. Further, they define what events the simulator should schedule during the simulation. These events are called *stimuli* and can be the generation of arriving messages, forced starting of instances, or injection of chaotic behavior.

MiSim supports five stimuli by default, outlined in Table V. Two create workloads, two create faultloads, and one allows spawning instances. The two types of workload generators support periodic and varying load profiles. The first type is the interval generator, which can be configured to send requests in spikes or evenly distributed over a time interval.

TABLE V: Supported stimuli

Type	Name	Description
Workload	Interval Gen.	Loads the system with a periodic workload.
	Time Series Gen.	Can create a varying workload.
Faultload	Chaos Monkey	Shuts down instances of a service.
	Delay	Injects a latency into a service connection.
Misc	Summoner	Can create new instances of a service.

```
{
  "name": "Minimal Scenario",
  "duration": 180,
  "artifact": "gateway",
  "component": "GET",
  "stimulus": "LOAD~
  ./Examples/PaperExample/paper_limbo.csv AND KILL
  service1 2@40"
}
```

Listing 2: Example for a minimal scenario-based experiment description loading a load profile (repeating as indicated by ∼) and killing two instances of *service1*

The second type is the time series generator, which can create varying workloads, e.g., using LIMBO models [27], [28]. Chaos monkeys and summoners can be used to manipulate a service's number of active instances. They allow the simulation of unexpected shutdowns or manual restarts of services. Lastly, a latency injection (delay) can be used to simulate a slow network connection. It acts similarly to the `tc` command[1] by slowing down incoming messages of the target service or endpoint.

There are two formats of experiment descriptions that *MiSim* accepts. The first one is scenario-based and relates closely to the aforementioned scenario. The second and more powerful way to describe an experiment is using a generic experiment description. It is less compact but allows for more details.

1) Scenario-based Experiment Description: In addition to the mandatory metadata information, a scenario-based experiment description of *MiSim* has a fixed format containing up to seven properties that mirror a scenario description. Three of them are required for a simulation. These are the *artifact*, *component*, and *stimulus*. The remaining four scenario components are optional since *MiSim* cannot interpret them yet. The *artifact* provides the simulator with the name of the microservice that should be stimulated during the experiment. The *component* indicates which of the *artifact*'s endpoints should be targeted if the experiment description contains a workload definition. Lastly, the *stimulus* property offers a way to define multiple stimuli. It uses keywords such as `AND`, `KILL`, and `LOAD` to build events for the experiment.

Listing Listing 2 shows a minimal scenario-based experiment description for the running example. The experiment uses a linked load profile to load the GET endpoint of the *gateway* service and kills two instances of *service1* at 40 STU. The simulation takes 180 STU (`default:` seconds).

[1]https://man7.org/linux/man-pages/man8/tc.8.html

```
{
  "simulation_metadata": {
    "name": "Minimal Scenario",
    "duration": 180,
    "seed": 42
  },
  "generators": [
    {
      "type": "limbo",
      "config": {
        "model":
        "./Examples/PaperExample/paper_limbo.csv",
        "target_operation": "gateway.GET",
        "repeating": true
      }
    }
  ],
  "Monkey#1": {
    "type": "chaos_monkey",
    "config": {
      "killed_instance_count": 2,
      "arrival_time": 40,
      "microservice": "service1"
    }
  }
}
```

Listing 3: The minimal example from Listing 2 as generic experiment description

2) Generic Experiment Description: Besides the scenario-based description, *MiSim* offers a more in-detail description variant for experiments. This generic form is also a JSON document but gives the creator more control over each component, i.e., more properties and features of the stimuli are exposed. Features that are otherwise not configurable are, for example, the event names of stimuli, generating workloads against more than one service, or the arrival times of workload generators. Listing 3 shows the same experiment as Listing 2 in the generic form. Note that the generic description is almost three times longer but allows for more complex features, e.g., naming of the fault injections.

D. Extension Points of MiSim

To allow for the simulation of various (implementations of) resilience mechanisms and scenarios, we designed *MiSim* to be easily extensible regarding resilience mechanisms, workload generators, and faultloads. *MiSim* employs a smart parsing and class system, which allows a user to extend a set of base classes and interfaces that automatically get parsed, integrated, and called during the simulation. Our provided basic and more complex implementations of common resilience mechanisms can be seen as proof of concept for our extension mechanisms. As shown in Figure 3, our implementations of the autoscaler, retry, and load balancer also use the strategy design pattern to create variations of these resilience mechanisms easily. The exact process of extension is explained in *MiSim*'s wiki [18].

VI. Simulation Results

A. Available Metrics

MiSim can output up to 25 metrics, including response times, queue lengths, instance counts, and circuit breaker

TABLE VI: Outputs of *MiSim*

Name	Format	Description
Raw Data	Folder	Contains all measured data points in *.csv* form.
Experiment Copy	JSON	Copy of the experiment file and some additional Metadata.
DESMO-J Trace	HTML	Simulation traces generated by DESMO-J.
Analysis Scripts	Python Script	Python scripts that visualize key metrics for a quick analysis of the experiment.

states[2]. Further, it generates several supporting metadata files and scripts upon a simulation execution. A list of possible outputs can be found in Table VI. These first contain the raw data collected during the simulation, which are the data points written into a collection of *.csv* files and can then easily be read for analysis. Next, *MiSim* provides a copy of the experiment input files and a metadata file containing some execution information. Lastly, multiple output artifacts help to understand and debug an experiment. A DESMO-J trace presents a human-readable format of the simulation log. Additionally, some analysis scripts can give a diagram-based visual overview of key metrics, such as instance counts or CPU utilization.

B. *Results Visualization*

Using the metrics and provided analysis scripts, up to five different types of diagrams can be generated to inspect the results visually. Diagrams can be generated for (i) instance counts, (ii) response times, (iii) CPU usage, (iv) queue length, and (v) arriving load during the experiment. The diagrams can be used in two ways. First, they help to understand whether the simulation does what it is supposed to do, and second, whether the simulated system satisfies a given requirement.

Executing the scenario shown in Listing 2 on the running example shows how the system could react if it experiences a linear load spike that reoccurs three times within 180 STU while two instances of *service1* crash. In the following, we present and interpret two of the five diagram types that can be generated[3]. These diagrams show the instance count of *service1* (Figure 5) and the overall system's response time (Figure 6). The first of these diagrams shows that the chaos monkey is executed as specified at the 40 STU mark. Additionally, we can see that the autoscaler takes effect. The service is scaled up after the injection and is scaled down and up around the 170 STU mark. The results shown in Figure 6 reveal that response times never exceed 3 and very rarely 2.5 STU.

VII. Evaluation

Our evaluation is divided into two parts. In the first part, we demonstrate *MiSim*'s functional capabilities regarding the simulation of resilience mechanisms and faultloads using the running example. In the second part, we compare simulation

[2]See https://github.com/Cambio-Project/MiSim/wiki/OutputMetrics

[3]The same diagrams can be obtained with the provided scripts. We did some additional post-processing steps to match the format for this paper.

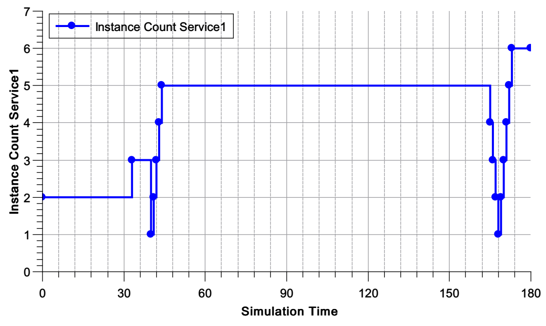


Figure 5: Instance count of *service1* when running the example scenario shown in Listing 2

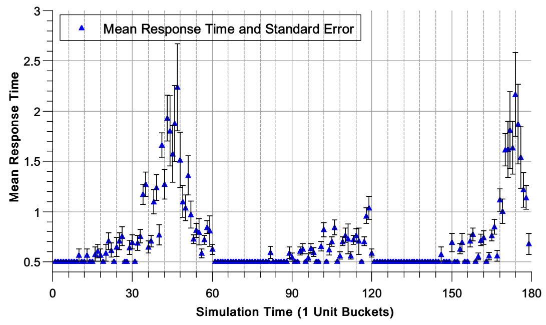


Figure 6: Response times of the API endpoint when running the example scenario shown in Listing 2

results against the actual scaling behavior of the more complex and established benchmarking application TeaStore [17]. Finally, we discuss the threats to validity of our evaluation.

A. *Simulating Resilience Mechanisms and Faultloads*

To demonstrate the functional capabilities of the simulator, we execute and analyze three scenarios that specifically investigate the behavior of five features: autoscaling, retry, circuit breaker, delay injection, and chaos monkey. Table VII describes the scenarios in detail, and Table VIII lists which features of *MiSim* are used in the respective experiments. The system used for these simulations is the running example introduced in Section V-A.

1) Scenario: Autoscaler and Load Balancer (A&L): In the first scenario, the system is expected to scale appropriately under a varying workload and keep the utilization of *service1* between 30% and 75%. We investigate which load balancing strategy is the best for the given scenario.

a) Experiment Description: The respective experiment demonstrates properties of three of the four current types of load balancing *MiSim* can employ, i.e., randomized, round-robin, and utilization-based load balancing. The fourth implementation, a more performant but less accurate variant of round-robin load balancing, is not evaluated. Furthermore, this experiment uses the autoscaler implementing a reactive strategy that checks the current average relative utilization $\overline{u}$ of all instances of a service once per STU. If the autoscaler detects overutilization (in this case: over 75%), it spawns as

TABLE VII: Description of evaluation scenarios

Scenario Name	Source	Stimulus	Artifact	Response	Response Measure	Environment
A&L	Users	Linear and Exponential Load Spikes	Whole System	Adjust instance counts for efficient utilization.	See Equation (1) below.	Normal operation
R&D	Networking Hardware	Network Saturation	Whole System	All requests should complete successfully.	See Equation (2) below.	Normal operation
C&C	Administrator	Service Restart	*service1*	Breach of the error threshold is detected.	All requests during downtime fail fast.	Maintenance

$$\forall i \in Instances: \quad 30\% \leq Util(i) \leq 75\% \tag{1}$$
$$\forall t \in Time: \quad \text{SuccessRate}(t) = 100\% \tag{2}$$

TABLE VIII: Enabled features in each of the experiments

Experiment Name	Related Scenario	Workload Type	Resilience Mechanisms: Load Balancing	Autoscaling	Retry	Circuit Breaker	Faultloads: Delay Injection	Chaos Monkey
E1	A&L	Varying	**X**	**X**				
E2	R&D	Varying	**X**		**X**		**X**	
E3	C&C	Constant	**X**			**X**		**X**

many instances as necessary to keep the utilization below the upscale threshold, assuming an utterly even load distribution. Underutilization is handled alike. We calculate the utilization of a single service instance based on Equation 3.

$$u = \frac{RemainingActiveDemand + TotalQueuedDemand}{CPUCapacity} \tag{3}$$

The experiment starts with one instance per service, and the stimulus is realized by a workload profile that starts with an exponential load spike and then keeps a high arrival rate. A drop to a very low load follows, and lastly, a linear rising load spike. In a previous work [16], we elicited linear and exponential load peaks as two stimuli in resilience scenarios, which are the basis for this experiment.

b) Expected Results: To assess the effectiveness of the load balancing strategies, we analyze how evenly they distribute requests over the active instances in terms of the created workload. For an optimal load balancer, the standard deviation of the utilization of all instances should be close to 0.

c) Experiment Results: In *MiSim*'s case, the random load balancer reaches an average standard deviation of approximately 19%, the round-robin approach 14%, and the utilization-based approach 29%. Only considering these values, the round-robin balancer performs best and the utilization-based approach worst. However, as visualized in Figure 7, the autoscaler behaves more stable and predictably and uses the least number of instances (22 vs. 24 and 37) when using the randomized strategy. On average, the autoscaler starts 1.3 ± 0.7 instances per out-scale with the randomized strategy, whereas the round-robin approach starts slightly more unstable 1.6 ± 1 instances. Again, the utilization-based approach performs the worst with 2.7 ± 2.7 average instance starts. This metric is also directly affected by the measured relative utilization.

These results show that even though *MiSim* only supports a mostly reactive and slightly predictive autoscaler, it is sufficient to simulate elastic and effective scaling. However, the accuracy of the autoscalers and load balancers still needs to be evaluated on measurements from real systems.

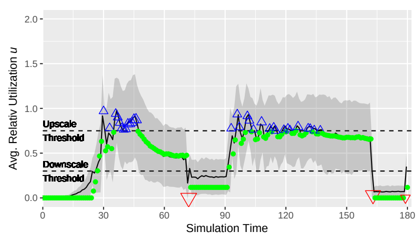
2) Scenario: Retry and Delay (R&D): The second scenario describes the case of network saturation, where a circuit breaker is expected to keep the request success rate at 100%. We investigate the effect of the retry pattern in this scenario.

a) Experiment Description: This experiment runs two system versions with the same load profile as Scenario A&L. One version does utilize a retry pattern, while the other does not. A normally distributed delay of 7 ± 1.1 STU is injected at the 60 STU mark to simulate the network saturation. The delay is chosen close to the default timeout of 8 STU, which occasionally causes requests to time out.

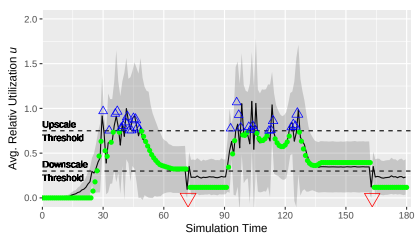
b) Expected Results: Since the delay is mostly below the timeout, almost all requests with the retry pattern are expected to succeed. Without the retry pattern, the probability density function in Equation (4) suggests that approximately 18.2% of requests time out.

$$\int_{8}^{\infty} \frac{1}{1.1 \cdot \sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-7}{1.1})^2} dx \approx 0.181651 \tag{4}$$

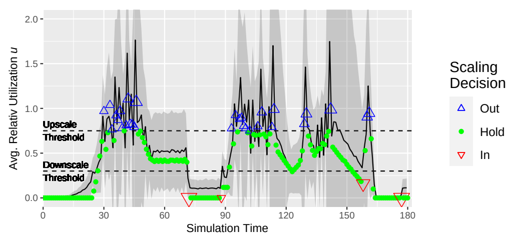
c) Experiment Results: Without the retry, the failure rate averages to approximately $18.6 \pm 0.5\%$ of requests failing, which is slightly above the expected amount of 18.2% due to the increased calculation effort for higher workloads. However, this can still be considered an accurate simulation of the delay. As shown in Figure 8a, the experiment version with the enabled retry drops the failure rate of requests to a flat 0% and therefore conforms with the expectations. Thus, using a retry would be a reasonable design decision for this scenario.



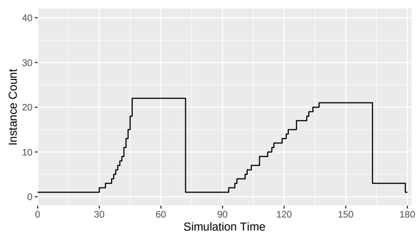
(a) Randomized Load Balancer — Mean Service Utilization



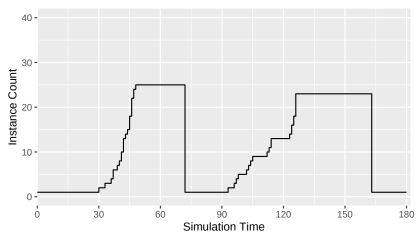
(b) Round-Robin Load Balancer — Mean Service Utilization



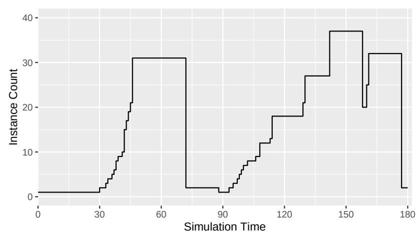
(c) Utilization-based Load Balancer — Mean Service Utilization



(d) Randomized Load Balancer — Instance Counts

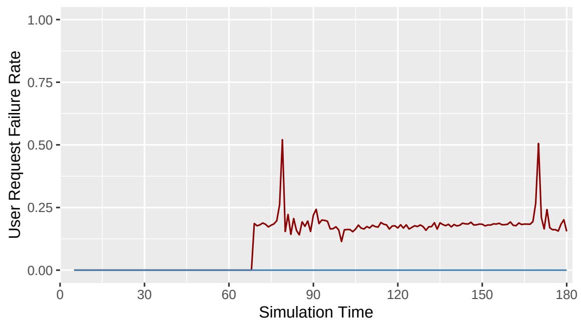


(e) Round-Robin Load Balancer — Instance Counts

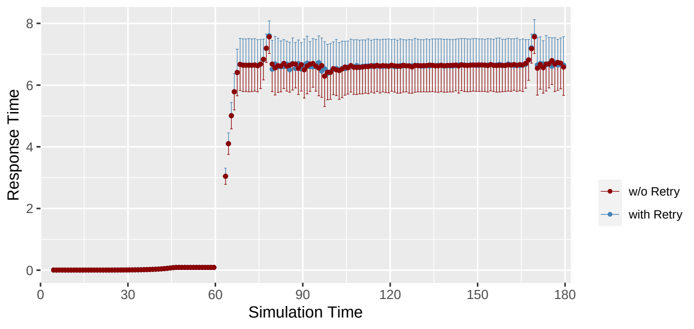


(f) Utilization-based Load Balancer — Instance Counts

Figure 7: Results for Scenario A&L



(a) Request failure rate with and without the retry pattern



(b) Response times with and without the retry pattern

Figure 8: Results for Scenario R&D (1 STU sized bins)

TABLE IX: Circuit breaker states during the maintenance period in Scenario C&C

State	Total Duration in State [STU]	Time Share	Times Entered
Open	75.38860	≈ 83%	126
Half-Open	0.21865	< 1%	126
Closed	15.39275	≈ 17%	2

3) Scenario: Chaos Monkey and Circuit Breaker (C&C): Lastly, this scenario describes the case that *service1* is taken down for maintenance. We investigate the behavior of a circuit breaker with a given configuration in that scenario.

a) Experiment Description: This experiment demonstrates the behavior of the simulated chaos monkey, summoner monkey, and circuit breaker. It uses a constant workload profile with requests arriving at the system every 0.1 STU.

The start of the maintenance period at 30 STU is simulated by using a chaos monkey, which kills all active instances of *service1*. A summoner monkey spawns a new instance at the 120 STU mark to simulate a manual restart.

The circuit breaker is configured to open at a failure rate of 75%. The rolling window over which it calculates the failure rate threshold is set to 20 STU and its sleep window to 0.5 STU. The connection limiter is not covert in any scenario since it is a side product of the circuit breaker and much less complex than the other resilience mechanisms.

b) Expected Results: In this experiment, the system is expected to detect the failing requests to the disabled service around the 45 STU mark and let requests fail fast until the service recovered. It periodically checks the connection in the *half-open* state by letting a single request through. Over the

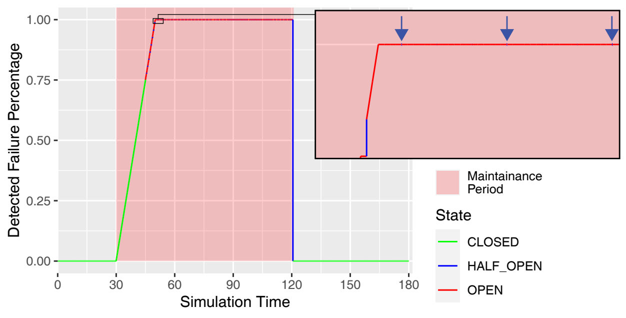


Figure 9: Detected failure percentage of the circuit breaker in Scenario C&C. Arrows indicate short duration in half-open states.

75 STU of downtime, the theoretical maximum of 151 state changes to the (half-)open state can be reached.

c) Experiment Results: The failure rate curve detected by the circuit breaker is shown in Figure 9. At the 45.4 STU mark, the detected failure threshold reaches 75%, and the circuit breaker opens. Afterward, the *half-open* state is entered periodically. Table IX shows that during this phase, both the *open* and *half-open* states were entered equally often (126 times). This number is lower than the theoretical maximum of 151 state changes since the change from *half-open* to *open* is only triggered by a failing request, which does not happen immediately after entering the *open* state. The portion of time spent in the *half-open* state is minimal, less than 1% of the total time. This is due to the relatively high-frequency workload that triggers the transition to *open* relatively quickly. Again, the experiment demonstrates successfully that the circuit breaker behaves according to the expectations and is a possible design for the investigated scenario.

B. Simulating the TeaStore Scaling Behavior

We aim to investigate whether *MiSim* can handle more complex architectures and workloads. Therefore, we partially replicate and simulate a setting described by Kistowski et al. [17] using the benchmarking application TeaStore [17]. In the experiment, we investigate the autoscaling behavior of the system without any further resilience mechanisms involved. In the following, we detail the experiment setting, describe the expected results, and present the results.

1) Experiment Description: Figure 10 displays the architecture of TeaStore [17] as used in the experiment. Designed as a reference application of a microservice-based system for benchmarking and used in many scientific studies, we assume TeaStore to be a representative example of real microservice-based software systems.

As the actual TeaStore system, the architecture description contains six services: *webui*, *image*, *auth*, *persistence*, *recommender*, and *registry*. The *webui* provides 13 different operations to the users and depends on *image*, *auth*, *persistence*, and *recommender*. In total, we modeled 37 operations and 40 operation dependencies. In contrast to the original TeaStore system, we do not actively use the *registry*. Furthermore, we

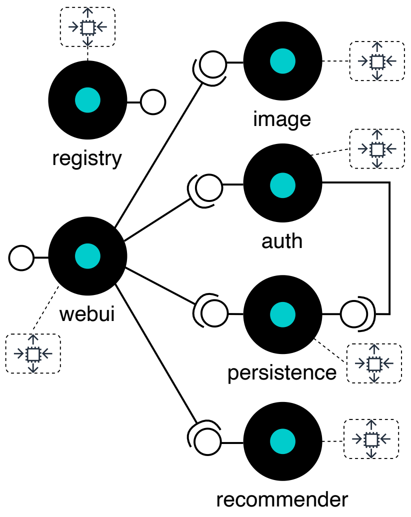


Figure 10: Architecture description of TeaStore [17]. All services are equipped with reactive autoscalers.

equip each service with a reactive autoscaler and initially spawn a single instance of each service.

We assume that an ordinary user of the TeaStore system visits the store, logs into their account, searches for categories and products, eventually buys the product, and finally logs out. This behavior is represented in operation calls to *webui* in Figure 11. Since this usage behavior does not contain any decision points, we model 11 load generators (one for each of these operations) that are executed by *MiSim* in the same order as depicted in Figure 11. Note that this experiment does not use four of *webui*'s available operations.

To apply a representative workload, we use the same workload profile as Kistowski et al. [17], which is based on data from the FIFA World Cup 1998 Website [29]. All 11 load generators use this identical workload profile visualized in Figure 12. The requests are distributed equally over all load generators. Additionally, we scaled the number of requests and capacities down by 10 to reduce the computational demand and memory consumption required for the simulation.

The autoscaler used by Kistowski et al. [17] scales between nine predetermined configurations with fixed instance counts. *MiSim* is currently not capable of simulating such configurations. To make the results comparable, we utilize three properties of the configurations in order to map the state of the simulated system. (1) The total number of instances between configurations is linearly and strictly increasing, allowing for a linear mapping. (2) In relation to each other, the simulated services should scale similarly to the actual services so that we can relate system sizes. (3) Since Configuration 9 is never used, it shall not be considered. Using these properties, we create the mapping shown in Equation (5).

$$\#config(n_{active}) = \left\lfloor \frac{8 \cdot (n_{active} - 5)}{(n_{max} - 5)} + 1 \right\rfloor \qquad (5)$$

2) Expected Results: Since we (partially) replicate the setting for the scaling behavior in the TeaStore [17] paper,

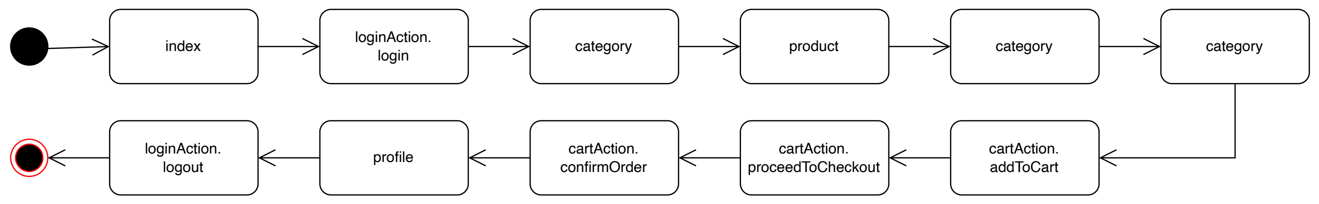


Figure 11: Usage behavior of the users in the TeaStore [17] simulation. Every step is an operation call to the *webui* service.

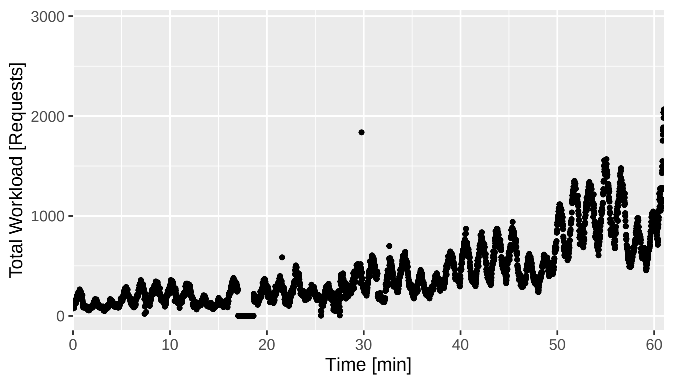


Figure 12: Fifa World Cup '98 based workload [29] used for the TeaStore evaluation

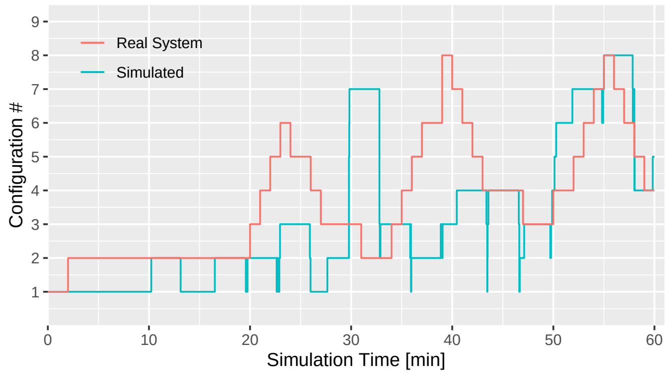


Figure 13: Comparison of simulation results against real system measurements by Kistowski et al. [17]

we expect to see similar scaling behavior in the simulation as for the real system. We do not expect similar instance counts as we could not calibrate the resource demands — due to necessary data not being publicly available — and used a less sophisticated autoscaling mechanism. However, we expect upscaling and downscaling at similar points in time, i.e., the overall shape of the curve should look similar.

Figure 13 depicts the demanded configurations of the real TeaStore system, and four phases are visible. From 0 to 20 minutes, the configuration stays at relatively constant and low levels. Next, there is a medium-sized peak from 20 to 30 minutes. Finally, there are two high peaks from 30 to 45 minutes and 45 to 60 minutes, respectively. Similarly, we expect the mapped configurations to show the same trend in system size as the actual configurations in the experiment by Kistowski et al. [17].

3) Experiment Results: Figure 13 shows the results of the experiment. The scaling behavior in the simulation is similar to the behavior in the real system. From 0 to 20 min, the configuration stays low and relatively constant. From 20 to 30 min, a small peak up to configuration level three can be seen, although this peak is less apparent than for the real system.

In contrast to the actual system, there is a huge spike at around 30 minutes. Here, *MiSim* seems more sensitive to the simultaneously occurring short spike in the workload profile. The different approaches regarding autoscaling can very likely explain this deviation. The reactive autoscaler in the simulation evaluates utilization and queue length at the current time, while TeaStore's autoscaler acts based on many metrics trends. In a big enough time window, such a short spike does not influence the trend enough to stimulate the autoscaler.

The expected two high peaks in the last two phases are also visible in the simulated results, although the first peak is less high and a bit delayed. Since the real system starts to scale roughly 5 minutes before the workload peak arrives (~40 min), we assume that the trend analysis of the real autoscaler kicked in based on the second phase. As we just mentioned, *MiSim* only scales once the load arrives, hence the delay. The real autoscaler corrects its overprovisioning at roughly 43 min, and both autoscalers meet at configuration #4 for the remainder of the peak. For the rest of the simulation, the scaling behavior matches the real TeaStore system closely.

Although the setup and results of the real and the simulated system do not match perfectly, the simulation is able to predict the overall trend of the scaling behavior correctly.

C. *Threats to validity*

To demonstrate the functional capabilities of *MiSim*, we use the architecture description of the proxy system, which contains selected services of the actual microservice-based application. In addition, the reflected industry system was still in an early stage of development.

The scenarios in the evaluation are synthetic. There is a threat that the scenarios may not represent actual resilience requirements. However, the scenarios are loosely based on representative resilience scenarios we elicited for the corresponding system in previous work [16].

We did not conduct a quantitative evaluation of *MiSim*'s accuracy in predicting autoscaling behavior. We did not rigorously validate the accuracy of *MiSim*'s other resilience mechanisms against measurements. However, along with the development, we conducted basic validation showing *MiSim*'s ability to accurately predict the system's transient behavior. The focus of this paper is to demonstrate *MiSim*'s architecture and features as well as the results from simulating typical resilience scenarios.

We were unable to replicate the original TeaStore setting with *MiSim* completely. In particular, we use independent autoscalers for each service, while the TeaStore setting uses a central autoscaler with predefined system configurations. Furthermore, we did not calibrate the services' operation demands based on measurements. However, since the operation calls are equally distributed, these factors are unlikely to strongly influence the overall shape of the scaling behavior on which we base our comparison.

VIII. Conclusion and Future Work

We presented the simulator *MiSim* for resilience assessment of microservice-based architectures. It simulates common resilience mechanisms and faultloads and is easily extensible with new mechanisms and implementations. In three scenarios, we demonstrated how autoscaling, retries, circuit breakers, delay injections, and instance killing are simulated. In another experiment, *MiSim* closely predicts TeaStore's up/down scaling behavior obtained from real measurements. However, more comparisons to measurements on real systems are still necessary to quantify the accuracy of the simulation for the other supported resilience mechanisms.

In future work, we plan to evaluate the accuracy of the simulation based on measurements from real software systems. We will complement *MiSim* with a software architecture extraction tool to ease the provision of architecture descriptions. Furthermore, we aim to support more sophisticated scenarios, e.g., by adding capabilities to interpret temporal logic formulas.

Acknowledgment

The authors thank S. Beck, J. Günthör, C. Zorn, and T. F. Düllmann for their contributions to the initial development of MiSim. Furthermore, the authors appreciate the support of the German Federal Ministry of Education and Research (BMBF) (dqualizer; FKZ: 01IS22007B and Software Campus 2.0 — Microproject: DiSpel; FKZ: 01IS17051). The work was conducted in the context of the SPEC RG DevOps Performance Working Group[4].

References

[1] J. Thönes, "Microservices," *IEEE Softw.*, vol. 32, no. 1, p. 116, 2015.

[2] J.-C. Laprie, "From dependability to resilience," in *38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks*, 2008, pp. G8–G9.

[3] R. Miles, *Learning Chaos engineering: discovering and overcoming system weaknesses through experimentation*. O'Reilly Media, 2019.

[4] M. Becker, S. Becker, and J. Meyer, "Simulizar: Design-time modeling and performance analysis of self-adaptive systems," in *Software Engineering 2013*. Bonn: Gesellschaft für Informatik e.V., 2013, pp. 71–84.

[5] H. H. A. Valera, M. Dalmau, P. Roose, J. Larracoechea, and 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)*. IEEE, 9 2020, pp. 94–99.

[4]https://research.spec.org/devopswg

[6] D. Meisner, J. Wu, and T. F. Wenisch, "BigHouse: A simulation infrastructure for data center systems," in *2012 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 4 2012, pp. 35–45.

[7] Y. Zhang, Y. Gan, and 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, 3 2019, pp. 212–222.

[8] Packetstorm Communications, "Network Simulation - PacketStorm," 2018. [Online]. Available: https://packetstorm.com/network-simulation

[9] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and 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, no. 9. John Wiley and Sons Ltd, 9 2017, pp. 1275–1296.

[10] D. Kliazovich, P. Bouvry, and S. U. Khan, "GreenCloud: a packet-level simulator of energy-aware cloud computing data centers," *The Journal of Supercomputing*, vol. 62, no. 3, pp. 1263–1283, 12 2012.

[11] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*, 4th ed. Addison-Wesley Professional, 2021.

[12] M. Nygard, *Release It! Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007.

[13] Microsoft, "Microsoft classification of design patterns in cloud-native application domain," https://docs.microsoft.com/en-us/azure/architecture/patterns/index-patterns, 2022.

[14] Resilience4j Contributors, "Resilience4j documentation," https://resilience4j.readme.io/docs, 2021.

[15] S. Newman, *Building microservices*. " O'Reilly Media, Inc.", 2021.

[16] S. Frank, M. A. Hakamian, L. Wagner, D. Kesim, C. Zorn, J. von Kistowski, and A. van Hoorn, "Interactive elicitation of resilience scenarios based on hazard analysis techniques," in *15th European Conference on Software Architecture (ECSA), Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 13365. Springer, 2021, pp. 229–253.

[17] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A micro-service reference application for benchmarking, modeling and resource management research," in *Software Engineering and Software Management, SE/SWM 2019*, ser. LNI, vol. P-292. GI, 2019, pp. 99–100.

[18] "MiSim GitHub Project," https://github.com/Cambio-Project/MiSim, 2022.

[19] "MiSim Code Ocean Capsule," https://doi.org/10.24433/CO.9682966.v3, 2022.

[20] Chaos Toolkit, "Chaos Toolkit," https://chaostoolkit.org, 2022.

[21] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: Systematic resilience testing of microservices," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2016, pp. 57–66.

[22] "MuSim - the microservice simulator (github repository)," https://github.com/elleFlorio/mu-sim.

[23] S. Becker, H. Koziolek, and R. H. Reussner, "The palladio component model for model-driven performance prediction," *J. Syst. Softw.*, vol. 82, no. 1, pp. 3–22, 2009.

[24] T. Lechler and B. Page, "Desmo-j: An object oriented discrete simulation framework in java," in *Proceedings of the 11th European Simulation Symposium (ESS)*, 1999, pp. 46–50.

[25] H. Garcia-Molina and K. Salem, "Sagas," *ACM Sigmod Record*, vol. 16, no. 3, pp. 249–259, 1987.

[26] M. Brooker, "Exponential backoff," https://aws.amazon.com/de/blogs/architecture/exponential-backoff-and-jitter/, 2021.

[27] J. v. Kistowski, N. R. Herbst, and S. Kounev, "Modeling variations in load intensity over time," in *Proceedings of the 2014 ACM International Workshop on Large Scale Testing*, ser. LT '14. ACM, 2014, p. 1–4.

[28] J. v. Kistowski, N. Herbst, and S. Kounev, "LIMBO: A tool for modeling variable load intensities," in *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 2014, p. 225–226.

[29] M. F. Arlitt and T. Jin, "A workload characterization study of the 1998 world cup web site," *IEEE Netw.*, vol. 14, no. 3, pp. 30–37, 2000.