Scenario-based Resilience Evaluation and Improvement of Microservice Architectures: An Experience Report

Sebastian Frank[1], Alireza Hakamian[1], Lion Wagner[1], Dominik Kesim[1], Jóakim von Kistowski[2] and André van Hoorn[1]

[1]Institute of Software Engineering, University of Stuttgart. Stuttgart, Germany
[2]DATEV eG, Nürnberg, Germany

Abstract
Context. Microservice-based architectures are expected to be resilient. However, various systems still suffer severe quality degradation from changes, e.g., service failures or workload variations. *Problem.* In practice, the elicitation of resilience requirements and the quantitative evaluation of whether the system meets these requirements is not systematic or not even conducted. *Objective.* We explore (1) the scenario-based Architecture Trade-Off Analysis Method (ATAM) for resilience requirement elicitation and (2) resilience testing through chaos experiments for architecture assessment and improvement. *Method.* In an industrial case study, we design a structured ATAM-based workshop, including the system's stakeholders, to elicit resilience requirements. We specify these requirements into the ATAM scenario template. We transform those scenarios into resilience experiments to quantitatively evaluate and improve system resilience. *Result.* We identified 12 resilience scenarios. We use and extend ChaosToolkit to automate and execute two scenarios. We quantitatively evaluate resilience requirements and suggest resilience improvements in the scope of both scenarios. We share lessons learned from the case study. In particular, our work provides evidence that an ATAM-based workshop is intuitive to stakeholders in an industrial setting. *Conclusion.* Our approach helps requirement and quality engineering teams in the process of resilience requirements elicitation.

1. Introduction

An intrinsic quality property of the microservices architectural style is resilience, i.e., the system meets performance and other Quality of Service (QoS) requirements despite different failure modes or workload variations [1]. However, real-world postmortems [2] show that systems suffer either unacceptable QoS degradation, or recovery time. It is necessary to assure system resilience in the context of microservice-based architectures.

Practitioners use Chaos Engineering [3], including tools such as CTK [4, 5] or Chaos Monkey[1], for resilience testing. They need to (1) think about hazards [6] as causes of QoS degradation, (2) set up chaos experiments by specifying failure mode types and hypotheses of expected quality behavior, and (3) execute each experiment to detect deviations from the hypotheses. First, this approach lacks the systematic identification of causes of a hazard through hazard analysis methods. We contributed to this problem in our previous work [7], which serves as a foundation for this paper. Particularly, we now integrate hazard analysis into a more systematic elicitation process and use a more formal description of requirements (scenarios). Second, the approach lacks a systematic process of eliciting and refining resilience requirements.

In the context of an industrial case study, our objective is to explore the application and adoption of the Architecture Trade-Off Analysis Method (ATAM) [8] for (1) the system's resilience requirement elicitation and (2) resilience testing through resilience experiments (aka chaos experiments) for architecture assessment and improvement. We hypothesize that ATAM has already been used in practice to elicit and specify quality requirements other than resilience, e.g., availability, performance, and maintainability, and can be adopted for effectively eliciting resilience requirements and evaluating them through chaos experiments. Therefore, our research question is: *How to leverage ATAM to elicit resilience requirements, which can be utilized to evaluate resilience through resilience experiments and suggest architectural improvements quantitatively?*

We use an ATAM-based workshop to elicit and specify resilience requirements by involving system stakeholders. ATAM allows us to describe resilience requirements as scenarios in semi-structured textual language. The scenario template consists of the following elements: (1) source, (2) stimuli, (3) artifact(s), (4) system's environment, (5) its response, and (6) response measure.

The designed structured workshop aims to identify hazards and architectural design decisions. During the workshop, we employ a hazard analysis based on the Fault Tree Analysis (FTA) [6]. The result is a set of 12 resilience scenarios, which we turn into experiments. Next, we use CTK to automate these experiments, and conduct a measurement-based resilience evaluation. Furthermore, we improve system resilience by applying a resilience pattern [9, 1], namely *retry*. We validate the improvement by re-executing the respective scenario.

ECSA 2021 Companion Volume, Växjö, Sweden, 13-17 September 2021

© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

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

CEUR-WS.org/Vol-2978/faacs-paper3.pdf

To summarize, the paper makes the following contributions:

- Leveraging ATAM and FTA in an industrial system to elicit resilience requirements, then evaluating the requirements, and improving system's resilience.
- Automating scenario execution using CTK for measurement-based evaluation.
- We share lessons learned that benefit both practitioners and researchers regarding resilience requirement elicitation, evaluation, and improvement.
- Artifacts — including scenarios, resilience experiments, and results of the experimentation — are available online [10].

2. Related Work

A workshop is an effective technique for requirement elicitation [11]. In our case, the workshop's preparation and conduction are based on the scenario template of Bass et al. [12]. Our difference to existing works on measurement-based resilience evaluation is that we have an explicit step on eliciting resilience requirements. In the next paragraphs, we elaborate on this in more detail.

Cámara et al. [13, 14, 15] propose an approach for resilience analysis of self-adaptive systems. The core idea consists of three parts: (1) specification of resilience properties using Probabilistic Computation Tree Logic, (2) modeling causes of a hazard, e.g., high-load using experimentation and collecting traces of system behavior, and (3) verification of resilience properties using model checking. In contrast to model checking-based verification, we evaluate a resilience scenario's response measure by analyzing collected measurements. Furthermore, Cámara et al. do not focus on the elicitation of resilience requirements using requirement engineering methods.

Chaos Engineering [5, 3] is a technique for evaluating system resilience through injecting failures [16]. There are works on both (1) using engineering methods to identify failure modes [7], i.e., causes of a hazard systematically before failure injection, and (2) ad-hoc failure injection with no systematic failure mode identification [2]. However, they do not explicitly specify resilience requirements and lack a methodical way for requirement elicitation. Our work is a step toward closing this gap.

In the context of resilience requirement elicitation, Yin et al. [17] propose a goal-oriented technique for representing resilience requirements. The high-level idea is to represent a resilience goal — e.g., all requests are processed correctly — and identify possible causes of hazards that act as obstacles for achieving a resilience goal — e.g., node failure. However, they do not discuss how to identify hazards and their causes. Goal orientation and developing scenarios are two activities in requirements engineering [11] that benefit the elicitation process. According to Pohl [11], scenario development benefits elicitation by making goals understandable for stakeholders and may refine or identify new goals. Our work uses scenario development without goal-oriented modeling as all stakeholders know the system's high-level quality goal.

To our knowledge, this is the first work using ATAM for eliciting and specifying resilience requirements before evaluating the resilience through experiments.

3. Research Methodology

Section 3.1 explains the domain context and describes the high-level architecture of the case study system. Section 3.2 summarizes our research methodology.

3.1. Domain Context

The case study system's purpose is to calculate payments. An accounting department's wage clerks use the payment accounting system to calculate each registered employee's income taxes. The payment accounting system has to gather data from health insurance providers and send its results to the corresponding tax office to execute the calculations. This process presumes that a company that wants to use the payment accounting system provides its employee and tax information to the health insurance provider and tax offices.

All of the payment accounting system's tasks are currently taken care of by a monolithic legacy system. In peak times, up to 13 million calculation requests have to be handled in a day or single night. Under normal circumstances, this number is significantly lower. In order to handle such varying loads more efficiently, stakeholders desired a better scaling system. Therefore, the old system will be replaced by a more scalable microservice-based Spring application in the coming years. The investigated part of the system under study, which is still under development, consists of seven services. It is deployed to a Platform as a Service (PaaS), i.e., Cloud Foundry (CF). Together with the industrial partner, we decided on a scenario-based approach, as our industrial partner already employed ATAM for other quality attributes.

3.2. Research Procedure

To answer our research question *How to leverage ATAM to elicit resilience requirements, which can be utilized to evaluate resilience through resilience experiments and suggest architectural improvements quantitatively?*, we conduct the following steps:

1. We gather relevant system stakeholders, i.e., product owners, software architects, and quality engineers,

into an ATAM-based workshop. The objective is to identify resilience scenarios that lead to QoS degradation and downtime.

2. We derive resilience experiments from the scenarios. The experiment description comprises the stimuli, artifact, and response according to the scenario.

3. We use CTK to automate the execution of the resilience experiments. We assess the response measure by analyzing the QoS metrics measurements.

4. After executing resilience experiments, we apply suitable resilience patterns. We re-execute the resilience experiments to assess the pattern's effect by comparing QoS-related behavior with and without the resilience pattern.

4. Elicitation and specification of scenarios

This section elaborates on the planning, execution, and results of the workshop.

4.1. Elicitation through Structured Workshop

Before the workshop, we received documentation regarding the architecture of the system. This allowed us to specify an architecture model of the case study system, including a component diagram and an explanation of the implemented components. Using ATAM, we required to know key architectural design decisions. Therefore, knowing the architecture description in advance allowed us to focus more on the hazard analysis and developing resilience scenarios.

The full-day workshop consisted of four sessions leveraging different methods, as described next. The moderators explained each technique and method at the beginning of each session. The participants were stakeholders of the system and comprised two software architects, one product owner, and one quality assurance engineer.

Session 1: Introduction and Architecture Description for achieving a common understanding of the workshop process and the system's architecture. (1) We resolved misunderstandings regarding the elicited architecture description through asking questions, and (2) refined the prepared architectural models.

Session 2: Hazard Analysis to identify potential causes for degradation in QoS. Index cards were used as a means to collect hazards. Afterward, the participants arranged the hazards and their causes in a fault-tree-like fashion. To not break the participants' creative flow, we relaxed the strict construction rules of fault trees, e.g., we allowed events having multiple parents, which resulted in a graph. For this reason, we refer to the result of this session as a *fault graph*. Note that the (directed acyclic) fault graph can be transformed into an equivalent fault tree by creating duplicate sub-trees for nodes having more than one parent.

Session 3: Resilience Scenarios for collecting and prioritizing resilience scenarios based on the previously identified hazards. We provided a scenario template based on the layout used in ATAM. Then, the stakeholders jointly created scenarios by informally analyzing the fault graph in a sequence driven by the associated severity (in descending order) of the hazards.

Session 4: Retrospective to collect feedback about the workshop from the participants and to inform them about the next steps, which comprise (1) refinement resilience requirements, and (2) execution of resilience experiments.

4.2. Workshop Results

Elicited Architecture Description: Figure 1 shows the component diagram of the system as specified in the first session of the workshop. It describes a snapshot of the system as used in the workshop and the subsequent activities. It represents a typical microservice-based architecture. As such, the system is deployed to a CF and contains several services. Each service has its own PostgreSQL database. The only exception is the *Calculations* service, which employs a Mongo database. The *API-Gateway* service handles all incoming connections and routes all communications. A *Eureka* service is employed to provide service discovery for all internal components. The *Frontend* service is the only external component that a user can directly access. The *Calculations* service is the central hub of the system since the calculation of payments is the system's main feature. Once this service receives a calculation request from the gateway, it collects all necessary data asynchronously from the other services. The *Companies* service is used to handle data the *Frontend* displays, but is not relevant for the calculation.

Hazard Analysis: Figure 2 shows the fault graph created in the second workshop session. The stakeholders agreed on unavailability or long response of settlement calculations as the main system hazard. Therefore, *user's settlement can not be calculated* is the top event in the fault graph. We analyzed possible causes from the top event until we reached basic events that we could not further decompose. We connected different causes by logical operators, i.e., *AND* and *OR*. For example, users can not calculate their settlement if it is not processed in time. This can occur when the assigned *instance stalls* OR *responds to slow*. We argue that the latter can be experienced if the system receives a sudden *(work)load peak* AND its *(auto) scaling does not work correctly*. The hazards at the leaf nodes are potential candidates for fault/failure injection during resilience experiments and

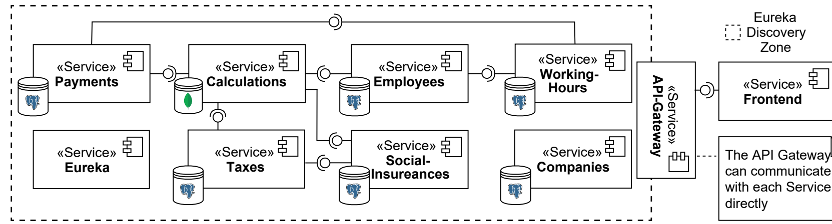


Figure 1: Component Diagram of Payment Accounting System

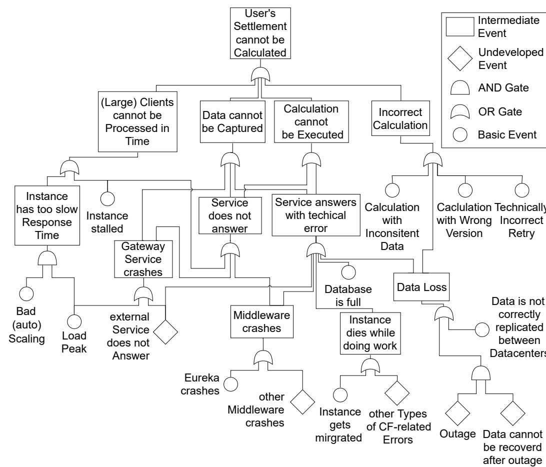


Figure 2: Cleaned Fault Graph

can be initiated by tools such as CTK. The stakeholders selected and prioritized the set of resilience experiments.

Resilience Scenarios: We gave the participants an empty table according to the ATAM scenario template with the columns (1) source, (2) stimulus, (3) artifacts, (4) environment, (5) response, and (6) response measure. Further, we explained the meaning of each table column to the participants. By using index cards again, the participants steadily added content to the table. We began by identifying possible *sources*. The *stimuli* and *artifacts* were then derived from the previously created fault graph. The *environment* represents different time periods when the identified stimuli occur. The *responses* are the stakeholders' assumptions about how the system should respond to the particular stimulus. The *response measures* are based on their internal Service Level Objectives (SLOs). For example, a workload peak resulting in a system failure was transposed into multiple scenarios. Users of the system are the source of the scenario since they cause the load peak. The respective stimulus is the workload peak itself. A service was chosen as the artifact to represent that a load peak can influence all service instances. As the environment, the payslip calculation period was chosen to imply an existing base workload. At last, the stakeholders chose the responses and response measures based on their SLOs.

The stakeholders elaborated 12 resilience scenarios, summarized in Table 1. Scenarios 01 to 04 are different variations of an unexpected load peak, including linear and exponentially increasing loads. Scenario 05 and 06 describe the failure of a single service instance. Scenario 07 and 08 are about middleware failures. Scenario 09 and 10 revolve around gateway failures. Lastly, Scenario 11 and 12 describe the failure of multiple instances. Actors such as end-users, elements of the CF platform, different bugs, and technical issues caused by the middleware or deployment artifacts and issues intrinsic to individual services of the system comprise the established sources. In total, all scenarios can affect all services. The environments cover different states of the system according to the identified system domain context, e.g., payslip calculation periods or simply services being non-idle independent of the different calculations. The response and response measures were specified by the stakeholders based on their internal SLOs.

Retrospection: The brief retrospective at the end of the workshop showed that the participants were satisfied with the agenda, content, and outcomes. However, comments were made concerning time management.

5. Resilience Evaluation

This section aims to evaluate the case study system's resilience. Therefore, we implemented a subset of the previously elicited resilience scenarios into resilience experiments using CTK. We compare the system's behavior against the expected behavior described in the scenarios' response part.

5.1. Experiment Setup

5.1.1. Examined Software System

Due to legal constraints and to maintain anonymity, our industrial partner provided us with a mocked version as a proxy for the real payroll accounting system. This version, shown in Figure 3, is used throughout this paper

ID	Short Name	Source	Stimulus	Artifact	Environment	Response	Response Measure
01	Peak(LinCo)/ Ser/Abr	User	Linear increasing load peak (cold start)	Service	Payslip calculation period	All requests are handled correctly and in time	Wage calculation $\leq$ 1 s, in 99 % of the cases, payslip calculation $\leq$ 20 s (300 Employees)
02	Peak(ExCo)/ Ser/Abr		Exponentially increasing load peak (cold start)				
03	Peak(LinCo)/ Ser/NoAbr		Linear increasing load peak (cold start)		Not during payslip calculation period		
04	Peak(ExCo)/ Ser/NoAbr		Exponentially increasing load peak (cold start)				
05	Failure(CF)/ Ins/Ber	Cloud Foundry	Instance terminates	Instance	During wage calculation	User unaware, calc. correct & in time, $\geq$1 instance running	
06	Bug/Ins/Ber	Bug				Developer gets notified	Developer gets notified within 5 min
07	Failure(MW)/ Bac/Ber	Middleware Operator	Middleware terminates	Backend	During wage calculation	Abort calculation, caller will be notified	Notification arrives within 1 s in 99 % of the cases
08	Failure(MW)/ Bac/Abr		Middleware terminates but recovers		Async. payslip calculation	Process is aborted but can be picked up	Restarts and SLOs are satisfied
09	Depl(GW)/ FroBac/NoIdle	Deployment	Gateway terminates	Front- and Backend	Not Idle	Frontend shows error, gateway restarts	Downtime of gateway instance is below 1 min
10	Failure(GW)/ FroBac/NoIdle	Technical Issue					
11	Failure(SerE)/ Ser/Ber	Receiving Service	Service terminates	Sending Service S, Receiving Service R	During wage calculation, no instance available	Error message and services R restarts	Downtime is below 1 min
12	Failure(SerE)/ Ins/Ber				During wage calculation, one instance not available	Calculation correct and in time	Wage calculation response time is below 2 s

Table 1: Scenarios created during the workshop

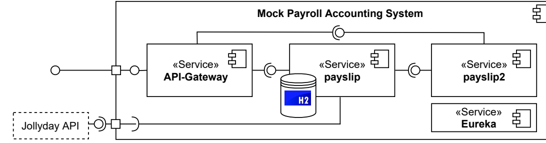


Figure 3: Mocked Payroll Accounting System

as the system under test. It implements a similar business logic but with less computational overhead. The system uses typical patterns of the microservice architectural style, i.e., *API-Gateway-service* as a central gateway that manages all incoming requests and *Eureka* [18] to provide service discovery. The *payslip-service* utilizes an H2 in-memory database and the third-party API *Jollyday*. It can forward requests to the *payslip-service2*. Requests can also be sent directly to *payslip-service2* using a different endpoint.

The following six endpoints are used during the experiments:

INTERNAL_DEP. — Calls the *payslip-service2* via *payslip-service*.

DB_READ — Reads an entry from the database of the *payslip-service*.

EXTERNAL_DEP. — Calls the third-party API *Jollydays* via *payslip-service*.

DB_WRITE — Writes an entry into the database of the *payslip-service*.

GATEWAY_PING — Checks whether the *API-Gateway-service* responds.

UNAFF._SERVICE — Sends a request directly to *payslip-service2*.

The actual payment accounting system is deployed to a paid CF. Due to financial constraints and legal issues, the mock system is deployed to a local CF environment [19], which has similar properties as a paid CF. As CF is a constraint given by the stakeholders, we did not consider other cloud providers.

5.1.2. Experiment Tools

Figure 4 shows our experiment framework comprising four tools, i.e., CTK, load generator, hypothesis validator, and dashboard. During an experiment, these tools interact with the system to monitor the experiments and provide detailed insights, e.g., response times of calls to individual endpoints.

To execute the experiments, we used CTK [20], which can execute and monitor chaos tests and has drivers for various PaaS solutions. We leveraged the CF driver to terminate a service instance at a specific point in time

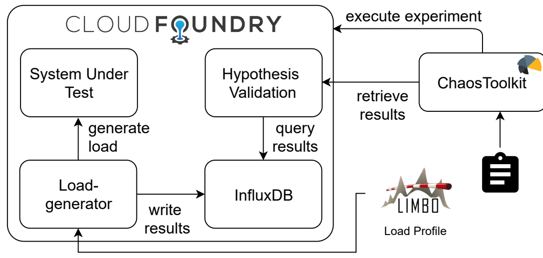


Figure 4: Used structure of the experiment framework

and validate the steady-state hypothesis. The load that the system receives is controlled by an adapted version of the load generator from the TeaStore microservices benchmark [21] that monitors response times, number of successful, dropped, and failed requests. The collected data is written into an InfluxDB [22] for a time series based evaluation. During the evaluation, a Spring service collects the necessary data from the InfluxDB and calculates whether a hypothesis holds. We also created a dashboard application that provides convenient features, like synchronized starting of CTK and the load generator, live monitoring, and automated CTK setup. Since the dashboard does not add functionalities in executing experiments, it is not part of Figure 4.

5.2. Experiment Execution

Based on Scenarios 04 and 05, we implemented three resilience experiments. The first experiment investigates a load peak with an exponential increase (Scenario 04), while the remaining two investigate instance termination due to an internal CF error for random instances (Scenario 05) and specifically the *payslip-service* (Scenario 05'). The selection of experiments is based on the industrial partner's preferences. In all experiments, the effect on all endpoints is examined. In the following, we will only discuss the results of a subset of endpoints for Scenario 05'. The residual results can be found in the supplementary material [10].

The design of the experiment related to Scenario 05' is given in Table 2. The target service of this experiment is the *payslip-service*, which holds the core business logic of the mock system. We use CTK to terminate running CF application instances to simulate the scenario's stimulus. The stimulus refers to an error that occurs in CF, which leads to a loss of an application instance. We assume that the blast radius only affects the *payslip-service* and that CF registers the loss of the *payslip-service* instance and starts a new instance. Our hypothesis is that the response measure of Scenario 05 still holds.

During the experiments, the system is exposed to an almost constant, synthetic load. We generated a load profile with a target load of 20 requests per second and some noise. The requests are evenly distributed over all six endpoints. To assess whether the system still responds correctly and in time, we measure response times of the requests and compute their success rate.

Target Service	*payslip-service*
Experiment Type	Terminate *payslip-service* application instance
Hypothesis	Response measure of Scenario 05 holds
Blast Radius	*payslip-service*

Table 2
Resilience experiment design for Scenario 05'

5.3. Experiment Results

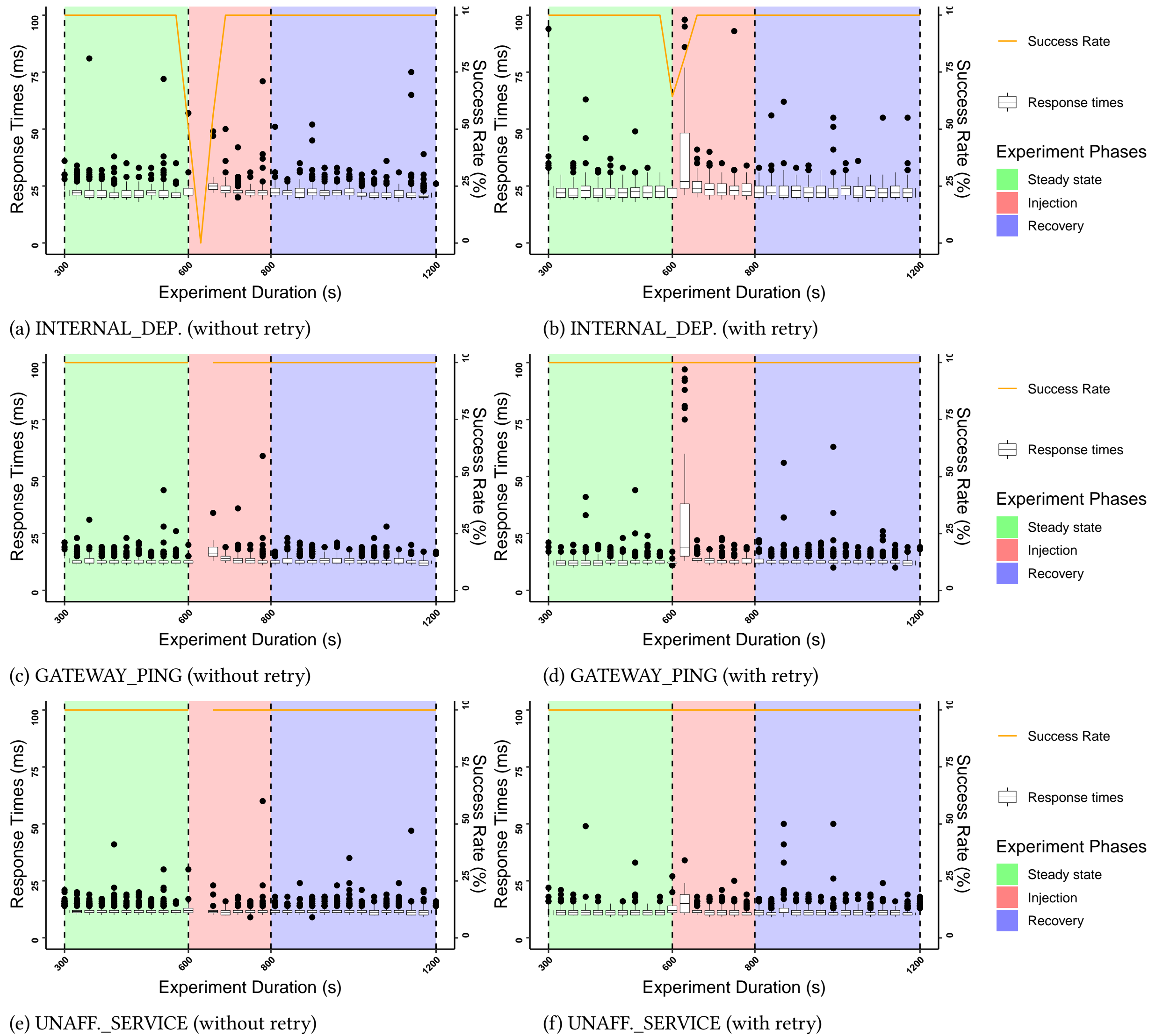
Figure 5 shows the steady-state, injection, and recovery phases of the experiment for endpoints INTERNAL_DEP., GATEWAY_PING, and UNAFF._SERVICE. In the steady-state phase, we assume that the system is working as expected, i.e., the response times satisfy the SLOs. In the injection phase, CTK terminates the *payslip-service* instance. In the recovery phase, we assume that the system recovers and returns to a steady state, i.e., the response times satisfy the SLOs. We omitted the load generator's warmup and cooldown phase due to readability and analysis purposes, which refers to the overall first and last 300 s. Further, a 30 s binning was applied, and extreme outliers (>100 ms) are not shown.

The success rates at the endpoints INTERNAL_DEP. (Figure 5a), DB_READ, EXTERNAL_DEP., and DB_WRITE drop to 0 % as the *payslip-service* is terminated after 600 s and rises back to 100 % as it recovers in about 1.5 min. During this downtime, no response times are recorded since no requests arrive at the *payslip-service*. During the steady-state and recovery phase, the response times are stable at around 20 ms and 15 ms, respectively. During the injection phase, there is a slight increase as the *payslip-service* has restarted. The results for GATEWAY_PING (Figure 5c) and UNAFF._SERVICE (Figure 5e) show a similar structure. However, the load generator did not record any successful or failed requests during the downtime. Therefore, no success rate could be calculated.

5.4. Discussion of Results

As visible in Figure 5 (left side), the response time and success rate values are almost identical in the steady state phase and the recovery phase. Furthermore, the increase in the success rate indicates that the *payslip-service* becomes available after 30 s to 60 s. Thus, the CF platform can re-instantiate the *payslip-service* quickly, leading to a quick recovery of the system.

Response times are slightly higher while the *payslip-service* is re-instantiated, which was expected as normal cold-start behavior. Endpoints GATEWAY_PING and



(a) INTERNAL_DEP. (without retry)

(b) INTERNAL_DEP. (with retry)

(c) GATEWAY_PING (without retry)

(d) GATEWAY_PING (with retry)

(e) UNAFF._SERVICE (without retry)

(f) UNAFF._SERVICE (with retry)

Figure 5: Comparison of experiment results at different endpoints with and without the implemented retry pattern

UNAFF._SERVICE should remain unaffected during the injection because the *payslip-service* is not required to answer the requests. Nevertheless, response times at endpoint GATEWAY_PING are affected, which indicates a propagation of the failure effects from the *payslip-service* to the *API-Gateway-service*.

After the injection started, the success rate drops to 0 % at the endpoints INTERNAL_DEP., DB_READ, EXTERNAL_DEP., and DB_WRITE. The CTK terminates the single *payslip-service* instance. The load generator flags all requests as failed, leading to a success rate of 0 %. The plots show neither successful nor failing request responses at the endpoints GATEWAY_PING and UNAFF._SERVICE during injection, which indicates no requests exist in the system. Another possibility is that requests have been dropped. Looking at the raw data tables disproves this argument as there are no dropped requests. Another explanation is that no requests arrived at the system, which leads to a lack of data in the time frame between approximately 600 s and 660 s.

We hypothesized that the response measure of Scenario 05 holds, i.e., requests are answered in time (99 % in less than 1 s) and correctly. As the response times are far below 1 s, our hypothesis regarding the response times

	Steady State								Injection								Recovery							
	w/o Pattern				w Pattern				w/o Pattern				w Pattern				w/o Pattern				w Pattern			
Endpoint	p_5	$\tilde{x}$	$\overline{x}$	p_{99}	p_5	$\tilde{x}$	$\overline{x}$	p_{99}	p_5	$\tilde{x}$	$\overline{x}$	p_{99}	p_5	$\tilde{x}$	$\overline{x}$	p_{99}	p_5	$\tilde{x}$	$\overline{x}$	p_{99}	p_5	$\tilde{x}$	$\overline{x}$	p_{99}
INTERNAL_DEP.	19	22	22.5	33	19	22	24.0	51	19	21	22.4	32	19	22	24.6	90	19	21	22.1	31	19	22	23.0	34
DB_READ	11	12	13.3	21	11	12	13.0	24	11	12	13.1	20	11	12	13.2	30	11	12	12.9	20	11	12	12.8	19
EXTERNAL_DEP.	11	12	12.6	21	10	12	13.3	23	11	12	12.5	21	10	12	14.1	31	11	12	12.3	19	10	12	12.2	19
DB_WRITE	11	13	13.4	21	11	12	13.1	22	11	12	13.1	20	11	12	13.3	27	11	12	13.0	20	11	12	12.7	19
GATEWAY_PING	11	12	13.3	21	11	12	13.3	24	11	12	13.1	20	11	12	13.6	32	11	12	12.9	19	11	12	12.9	21
UNAFF._SERVICE	10	11	11.9	19	10	11	11.6	19	10	11	11.7	18	10	11	11.6	19	10	11	11.8	18	10	11	11.5	19

Table 3
Statistical summaries of the three experiment phases. p_α: α-th percentile; $\tilde{x}$: median; and $\overline{x}$: mean. Values are given in ms.

is technically fulfilled. However, several requests are not answered at all, which is indicated by the dropped success rate. We consider these as incorrect response. Therefore, we assume that the hypothesis regarding correctness is not fulfilled.

6. Resilience Improvement

The previous section's experiments showed that the system does not respond as described in Scenario 05 to a failure of an instance of the *payslip-service*. While the response times are technically below 1 s in 99 % of all cases, requests are temporarily not answered at all, and thus, not correctly. Therefore, we aim to improve the system's success rate concerning Scenario 05 by applying resilience pattern(s). We then determine the efficacy of improvements to the system's resilience by re-executing the experiments.

6.1. Architectural Modifications

The system under test was fortified with a retry pattern [9], i.e., the *API-Gateway-service* sends another request to the *payslip-service* if a request fails or remains unanswered. The retry pattern seems to be a reasonable choice since response times are far below the threshold of 1 s, as indicated by the previous experiment. Due to its specific purpose, the system has to accept requests near real-time and always answer correctly. Thus, resilience patterns that rely on backup or restricting behavior, like circuit breakers or flow limiters, are unsuited. To avoid bad retry behavior, we configured the Spring-Retry as follows. We set the maximum number of retries of each *payslip-service* request to be 4, the initial delay to 10 ms, the factor for the exponential increase to 3, and the maximum delay to 150 ms — resulting in retries after 10 ms, 30 ms, 90 ms, and 150 ms.

6.2. Experiment Results and Discussion

Each plot on the right side of Figure 5 visualizes the system's response times and success rates with the retry pattern for an endpoint. As in the experiment presented in Section 5, each plot is divided into the steady state phase, injection phase, and recovery phase. Table 3 shows the associated statistical values.

In general, similar behavior can be observed at all the endpoints. Comparing the plots at left and right of the Figure 5, shows that the mean response times in the steady state phase do not vary significantly when the retry pattern is activated. Although, at the beginning of the injection phase, far more high response times can be observed. In addition, the boxplots show a slightly higher interquartile range in the plot where the retry pattern is integrated.

The plots also show that the success rate does not drop to zero anymore when the pattern is active. For the endpoints INTERNAL_DEP., the success rate drops to approximately 70 %. For the two endpoints GATEWAY_PING and UNAFF._SERVICE, requests are arriving and the success rate remains at 100 %.

The application of the retry pattern can explain the response time spikes during the injection (see the Figure 5). Requests sent shortly before the restart of the *payslip-service* fail, but are retried by the *API-Gateway-service* until the *payslip-service* recovered after approximately 10 s. However, as several retries have been aggregated, the *payslip-service* will have to handle a high amount of requests upon recovery, resulting in a visible spike in response times.

The endpoints UNAFF._SERVICE and GATEWAY_PING do not depend on the *payslip-service*. This explains the high success rate at these endpoints.

In contrast to the experiment without the retry pattern, the success rate does not drop entirely. Therefore, the retry pattern improves the scenario satisfaction as it increased the percentage of correct responses while keeping the response times below 1 s.

7. Discussion

7.1. Key Lessons Learned

Lesson 1: Elicitation of resilience requirements involves hazard analysis. It is essential to include stakeholders with different roles and particular expertise in

the business domain to quickly prepare a list of relevant hazards. Other roles, such as software developers and infrastructure engineers, help to identify causes of hazards that stem from software and its running environment.

Lesson 2: ATAM is a useful method to adopt resilience elicitation. Stakeholders of the software project were already familiar with scenario development for quality requirements. Therefore, the structure of the scenario template of Bass et al. [12] was intuitive for the stakeholders.

Lesson 3: Loose adoption of formalisms is already good enough. Researchers and practitioners have used fault tree formalism for both qualitative and quantitative hazard analysis in safety engineering. To identify the causes of a hazard, we did not have to comply with fault tree formalism rigorously. The informal way of constructing a fault tree was easy to understand for stakeholders.

Lesson 4: The ATAM workshop requires considerable refinement that can be done “offline”. The outcome of the well prepared one-day workshop needed further refinement. In particular, it was necessary to refine the stimulus and response measures parts of each scenario, e.g., we modeled the workload and tried to express the scenarios in temporal logic. This revealed that the initial requirements were partially ambiguous and imprecise, which was easy to resolve through clarification requests to the stakeholders. Therefore, we hypothesize that formalization benefits both validation and quantitative evaluation of resilience requirements and that an explicit (offline) formalization step could complement the proposed workshop well.

Lesson 5: A tightly planned one-day workshop is sufficient. We managed to collect resilience scenarios in a one-day workshop because it was well prepared (knowing the architecture description) and well-conducted (strict time management). Refinement can be done offline by more skilled engineers in formalizing stimuli and response measures (similar to writing SLOs). However, it is important to ask for feedback to check the validity of the requirements.

Lesson 6: The resilience elicitation helps to refine “classical” QoS requirements. All response measures are based on non-resilience specifications that make them imprecise. For example, maximum degradation and time to recovery was not specified. Thus, it is unclear whether experimentation shows acceptable or unacceptable degradation in performance or availability quality.

7.2. Threats to Validity

We discuss the threats to validity for the workshop and our experiment design.

7.2.1. Workshop

Conclusion validity One threat is the reliability of measures, which means repeating the workshop yields the same resilience requirements list. Elicitation of resilience requirements involves human judgment. Hence, it is a subjective measure. Therefore, we can not entirely rule out this threat.

Internal validity One threat is instrumentation, which means our tools and techniques were not suitable. We conducted a one-day structured workshop and used the scenario template of Bass et al. [12] for eliciting resilience requirements. We refined all the resilience requirements through several iterations after the workshop and validated them against the workshop participants.

Construct validity For us, the main threat in this category is mono-method bias, which means we did not use other elicitation methods. Therefore, there is a threat that elicited resilience requirements are biased. We can not entirely rule out this threat as we did not apply other methods and cross-check the results.

External validity The heterogeneity poses a threat, i.e., different roles and expertise of participants. Workshops with less heterogeneity in the stakeholders could lead to no resilience requirements. We can not entirely rule out this threat.

7.2.2. Experiment design

We used the mock system for quantitative evaluation of resilience requirements that are based on the actual system. There is a threat that evaluation results are inaccurate. However, the purpose of the experiments is to exemplary show how elicited requirements and derived experiments can help to improve the system — we do not claim the accuracy of the quantitative results. Furthermore, due to legal issues, we used CF Dev [19]. We faced instability, e.g., resource drainage of Dev nodes, in the environment during experimentation. There is a threat of a negative impact on results due to this instability. To counteract this threat, we re-executed experiments to gain insight into approximate measurements, ensuring reliable data with no unintended node or service crash.

8. Conclusion

The successful development of resilience scenarios depends on the outcome of the hazard analysis. Our approach to scenario-based resilience evaluation assumes a business domain expert to derive an initial list of hazards. FTA can then be a means to analyze the hazards and derive resilience scenarios. We plan to (1) extend our process with an explicit formalization step after the workshop for refinement of the scenarios, (2) formally verify

response measures of resilience scenarios, and (3) create processes for continuous hazard analysis when a system faces changes, e.g., updates and refinement/development of resilience scenarios.

Acknowledgments

This work has been supported by the Baden-Württemberg Stiftung (ORCAS — Efficient Resilience Benchmarking of Microservice Architectures) and the German Federal Ministry of Education and Research (Software Campus 2.0 — Microproject: DiSpel).

Data Availability

Our artifacts [10] comprise (i) the resilience scenarios and (ii) the data and R scripts as a CodeOcean capsule. We are working on making parts of the created/modified experiment tools available as open-source software. For confidentiality reasons, the system under test cannot be published.

References

[1] S. Newman, Building Microservices, O'Reilly, 2015.

[2] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, V. Sekar, Gremlin: Systematic resilience testing of microservices, in: Proc. 36th IEEE Int. Conf. on Distributed Computing Systems (ICDCS), 2016, pp. 57–66.

[3] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, C. Rosenthal, Chaos engineering, IEEE Softw. 33 (2016) 35–41.

[4] Chaos toolkit, 2020. URL: https://github.com/chaostoolkit.

[5] R. Miles, Learning Chaos Engineering – Discovering and Overcoming System Weaknesses through Experimentation, O'Reilly Media, Inc., 2019.

[6] N. G. Leveson, Safeware — System Safety and Computers: A Guide to Preventing Accidents and Losses Caused by Technology, Addison-Wesley, 1995.

[7] D. Kesim, A. van Hoorn, S. Frank, M. Häussler, Identifying and prioritizing chaos experiments by using established risk analysis techniques, in: Proc. 31st Int. Symposium on Software Reliability Engineering (ISSRE), 2020.

[8] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, The architecture tradeoff analysis method, in: Proc. 4th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS), 1998, pp. 68–78.

[9] M. T. Nygard, Release It!: Design and Deploy Production-ready Software, Pragmatic Bookshelf, 2018.

[10] S. Frank et al., Supplementary material, 2020. Artifacts: https://doi.org/10.5281/zenodo.5142006 (Scenarios); https://doi.org/10.24433/CO.0520280.v1 (Code Ocean capsule).

[11] K. Pohl, Requirements Engineering - Fundamentals, Principles, and Techniques, Springer, 2010.

[12] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, 2 ed., Addison-Wesley Longman Publishing Co., Inc., USA, 2003.

[13] J. Cámara, R. de Lemos, Evaluation of resilience in self-adaptive systems using probabilistic model-checking, in: Proc. 7th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012, pp. 53–62.

[14] J. Cámara, R. de Lemos, M. Vieira, R. Almeida, R. Ventura, Architecture-based resilience evaluation for self-adaptive systems, Computing 95 (2013) 689–722.

[15] J. Cámara, R. de Lemos, N. Laranjeiro, R. Ventura, M. Vieira, Robustness-driven resilience evaluation of self-adaptive software systems, IEEE Transactions on Dependable and Secure Computing 14 (2017) 50–64.

[16] R. Natella, D. Cotroneo, H. Madeira, Assessing dependability with software fault injection: A survey, ACM Computing Surveys (CSUR) 48 (2016) 44:1–44:55.

[17] K. Yin, Q. Du, W. Wang, J. Qiu, J. Xu, On representing and eliciting resilience requirements of microservice architecture systems, CoRR abs/1909.13096 (2020). URL: https://arxiv.org/abs/1909.13096v3. arXiv:1909.13096.

[18] Netflix Inc., Eureka, 2020. URL: https://github.com/Netflix/eureka.

[19] Cloud Foundry Foundation, Cloud foundry dev documentation, 2020. URL: https://github.com/cloudfoundry-incubator/cfdev.

[20] Chaos Toolkit, Chaos toolkit documentation, 2020. URL: https://chaostoolkit.org.

[21] 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: Proc. IEEE 26th Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2018, pp. 223–236.

[22] InfluxData Inc., InfluxDB website, 2020. URL: https://www.influxdata.com/.