Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Assessing Resilience of Software Systems by Application of Chaos Engineering – A Case Study

Dominik Kesim

Course of Study:	Softwaretechnik
Examiner:	Dr.-Ing. André van Hoorn
Supervisor:	Matthias Häussler (Novatec), Christian Schwörer (Novatec), Dr.-Ing. André van Hoorn.
Commenced:	März 1, 2019
Completed:	September 1, 2019
CR-Classification:	I.7.2

Abstract

Modern distributed systems more and more frequently adapt the microservice architectural style to design cloud-based systems. However the prevalence of microservice architectures and container orchestration technologies such as Kubernetes increase the complexity of assessing the resilience of such systems. Resilience benchmarking is a means to assess the resilience of software systems by leveraging random fault injections on the application- and network-level. Though current approaches to resilience benchmarking have become inefficient. Chaos engineering is a new, yet, evolving discipline that forces a change in the perspective of how systems are developed with respect to their resilience. The key idea is to apply empirical experimentation in order to learn how a system behaves under turbulent conditions by intentionally injecting failures. Solving the errors found by this approach in addition to repeating the same experiments, allows the system to build up an immunity against failures before they occur in production. In the scope of an industrial case study this work provides means to identify risks and hazards by applying three hazard analysis methods known from engineering safety-critical systems to the domain of chaos engineering, namely i) Fault Tree Analysis as a top-down approach to identify root causes, ii) Failure Mode and Effects Analysis as a component-based inspection of different failure modes, iii) and Computational Hazard and Operations as a means to analyze the system's communication paths. A dedicated number of the identified hazards are then implemented as chaos engineering experiments in Chaostoolkit in order to be injected on the application-platform-level, i.e., Kubernetes. In total, four experiments have been derived from the findings of the hazard analysis whereas three experiments have been executed and analyzed by applying non-parametric statistical tests to the observations. This work provides a generic approach to assessing the resilience of a distributed system in the context of chaos engineering illustrated by an industrial case study.

Keywords – Resilience, Chaos Engineering, Chaostoolkit, Hazard Analysis, Industrial Case Study, Kubernetes

Kurzfassung

Moderne verteilte Anwendung übernehmen immer öfter den Microservices-Architekturstil um cloud-basierte Anwendungen zu designen. Allerdings erhöht die weite Verbreitung von Microservice-Architekturen und Container-Orchestrierung Technologie, wie zum Beispiel Kubernetes, die Komplexität zur Bewertung und Verbesserung der Widerstandsfähigkeit (engl. resilience) von verteilten Anwendungen. Resilience Benchmarking ist eine Methode die Widerstandsfähigkeit von Anwendungen mit Hilfe von zufälligen Fehlerinjektionen auf Applikations- und Netzwerkebene zu bewerten. Jedoch sind aktuelle Resilience-Benchmarking-Ansätze ineffizient. Chaos engineering ist eine neue, sich entwickelnde Disziplin, welche ein Umdenken hinsichtlich der Entwicklung von Anwendungen in Bezug auf ihre Widerstandsfähigkeit erzwingt. Die Idee ist es, durch Ausführung empirischer Experimente mehr über das Verhalten eines Systems mit Hilfe von gezielten und kontrollierten Fehlerinjektionen zu lernen. Durch diesen Ansatz gefundene und behobene Probleme und Wiederholung der Experimente erlauben es dem System, eine Immunität gegenüber gefundenen Fehlern aufzubauen. Im Rahmen einer industriellen Fallstudie stellt diese Arbeit Techniken zur Identifikation von Risiken und Gefährdungen, in einem Chaos-Engineering-Kontext, zur Verfügung. Es wurden drei Gefährdungsanalyse Techniken vorgestellt, namentlich i) Fault Tree Analysis als top-down-Ansatz zur Identifikation von Grundursachen, ii) Failure Mode and Effects Analysis zur Komponenten-basierten Analyse von verschiedenen Fehlermodi, iii) und ComputationalHazard and Operations als Mittel zur Analyse der Kommunikationswege des Systems. Die identifizierten Gefährdungen werden anschließend mit Hilfe des Chaostoolkits als Chaos Engineering Experimente definiert und auf Plattform-Ebene (Kubernetes) ausgeführt. Insgesamt wurden vier Experimente aus der Gefährdungsanalyse abgeleitet und entworfen wobei drei dieser Experimente ausgeführt und durch statistische Tests analyisiert wurden. Diese Arbeit stellt einen generischen Ansatz zur Verbesserung der Widerstandsfähigkeit von Verteilten Anwendungen, veranschaulicht Anhand einer industriellen Fallstudie, im Kontext von Chaos Engineering bereit.

Schlüsselwörter – Widerstandsfähigkeit, Chaos Engineering, Chaostoolkit, Gefährdungsanalyse, Industrielle Fallstudie, Kubernetes

Contents

1. Introduction 1

2. Foundations 5
   2.1. Architecture and System Design 5
   2.2. Hazard Analysis 7
   2.3. Research Methods 11
   2.4. Technologies 16
   2.5. Software Resilience and Dependability 23
   2.6. Chaos Engineering 26
   2.7. State of the Art Tools and Methods 30

3. Related Work 39
   3.1. Search Strategy 39
   3.2. Chaos Engineering by Basiri et al. 40
   3.3. Professional Conference Contributions related to Chaos Engineering . . . 41
   3.4. Case Studies 43
   3.5. Summary of Findings 45

4. Case Study Design 47
   4.1. Objective 47
   4.2. The Case 48
   4.3. Preparation of Data Collection 48
   4.4. Data Collection Methodology 50
   4.5. Data Evaluation Methodology 52
   4.6. Interview Structure and Question Design 52

5. Cloud Service for Vehicle Configuration Files 55
   5.1. Findings from Interviews 55
   5.2. Domain Context and System Scope 56
   5.3. Architecture of the Legacy Application 58

5.4. Prototype of a Microservice based Architecture 61

6. Hazard Analysis 69
6.1. Failure Modes and Effects Analysis 70
6.2. Fault Tree Analysis 75
6.3. Computational Hazard Analysis and Operability 82
6.4. Summary of Hazard Analysis 87

7. Experiment Environment and Tool Configuration 89
7.1. Cloud Environment 90
7.2. Azure Kubernetes Service – Deployment 91
7.3. Simulating and Monitoring Environmental Conditions 92

8. Chaos Experiments Design, Execution, and Analysis 95
8.1. Steady State Description 95
8.2. Experiment Description 98
8.3. Hypothesis Backlog and Experiment Execution 99
8.4. Discussion 122
8.5. Threats to Validity 126

9. Conclusion 131
9.1. Summary 131
9.2. Discussion 132
9.3. Future Work 135

Appendices 136

A. Interview Resources 137
A.1. Question Catalogue 138
A.2. Consent Form 141

B. Incident Analysis Results 143
B.1. Failure Modes and Effects Analysis 144
B.2. Fault Tree Analysis 150
B.3. Computational Hazard and Operations Study 156

Bibliography 161

Online References 165

List of Figures

2.1. Intermediate fault event 9
2.2. Basic fault event 9
2.3. Input and output events 9
2.4. Logic gates 9
2.5. Organizational structure of a Kubernetes cluster [Fou] 18
2.6. Components of dependability [ALR+01] 24

5.1. Investigated system embedded in its domain context 57
5.2. Use Case diagram of the prototype and legacy system 59
5.3. Monolithic implementation of the case study application 60
5.4. Component diagram of the prototype system 61
5.5. Sequence diagram of the getAll request 63
5.6. Sequence diagram of the create request 63
5.7. Sequence diagram of the delete request 64

6.1. Fault tree with top-level hazard: "User does not receive configuration identifiers" 76
6.2. Left sub tree of Figure 6.1 77
6.3. Right sub tree of Figure 6.1 78

7.1. Experiment environment 90
7.2. JMeter test plan 93
7.3. JMeter user thread group HTTP request configuration 94

8.1. Response times for warmup, steady-state, injection and recovery phase with respect to the GET_CONFIG_ID operation 96
8.2. Steady-state phase of the first experiment 97
8.3. Response times of GET_CONFIG_ID operation after first experiment . 104
8.4. Response times of GET_CONFIG_DATA operation after first experiment 105
8.5. Empirical Cumulative Distribution Function (ECDF) of GET_CONFIG_ID operation after experiment no. 1 106

8.6. Empirical Cumulative Distribution Function (ECDF) of GET_CONFIG_DATA operation after experiment no. 1 106
8.7. Response times of of GET_CONFIG_ID after second experiment . . . 110
8.8. Response times of of GET_CONFIG_DATA after second experiment . 111
8.9. Empirical Cumulative Distribution Function (ECDF) of GET_CONFIG_ID operation after experiment no. 2 112
8.10. Empirical Cumulative Distribution Function (ECDF) of GET_CONFIG_DATA operation after experiment no. 2 112
8.11. Response times of GET_CONFIG_ID after third experiment 116
8.12. Response times of GET_CONFIG_DATA after third experiment 117
8.13. Empirical Cumulative Distribution Function (ECDF) of GET_CONFIG_ID operation after experiment no. 3 118
8.14. Empirical Cumulative Distribution Function (ECDF) of GET_CONFIG_DATA operation after experiment no. 3 118

A.1. Question Catalogue page 1/3 138
A.2. Question Catalogue page 2/3 139
A.3. Question Catalogue page 3/3 140
A.4. Consent Form page 1/3 141
A.5. Consent Form page 2/3 142
A.6. Consent Form page 3/3 142

B.1. Failure Modes and Effects Analysis of the Ingress component 1/3 144
B.2. Failure Modes and Effects Analysis Results of the Ingress component 2/3 145
B.3. Failure Modes and Effects Analysis Results of the Ingress component 3/3 145
B.4. Failure Modes and Effects Analysis Results of the Registry component 1/2 146
B.5. Failure Modes and Effects Analysis Results of the Registry component 2/2 147
B.6. Failure Modes and Effects Analysis Results of the data management component 148
B.7. Failure Modes and Effects Analysis Results of the Admin service 1/2 . . . 148
B.8. Failure Modes and Effects Analysis Results of the Admin service 2/2 . . . 149
B.9. Failure Modes and Effects Analysis Results of the User service 149
B.10. Fault Tree Analysis Results: User does not receive configuration identifiers 1/3 150
B.11. Fault Tree Analysis Results: User does not receive configuration identifiers 2/3 151
B.12. Fault Tree Analysis Results: User does not receive configuration identifiers 3/3 152
B.13. Fault Tree Analysis Results: Administrator cannot delete configuration data objects 153

B.14.Fault Tree Analysis Results: Administrator cannot create configuration data objects 154
B.15.Fault Tree Analysis Results: Administrator cannot load/unload configuration data objects 155
B.16.Computational Hazard and Operations Results: Ingress-Registry 156
B.17.Computational Hazard and Operations Results: Registry-Master 1/2 . . . 157
B.18.Computational Hazard and Operations Results: Registry-Master 2/2 . . . 157
B.19.Computational Hazard and Operations Results: Master-Data management 1/2 158
B.20.Computational Hazard and Operations Results: Master-Data management 2/2 159
B.21.Computational Hazard and Operations Results: Master-Config MS 159
B.22.Computational Hazard and Operations Results: Ingress-Adminservice . . 160
B.23.Computational Hazard and Operations Results: Ingress-Userservice . . . 160

List of Tables

2.1. Example FMEA for wind turbines [AOT10] 8
2.2. Example of the use of guidewords in case of a train telephone system [WJG01] 11
2.3. Examples of parametric and non-parametric hypothesis tests 16
2.4. Chaos experiment objects declared by ChaosToolkit 21
2.5. Chaos Tools used by Netflix also known as the Simian Army [IT11] . . . 31

4.1. Data sets directly related to the investigated system 49
4.2. Data sets not directly related to the investigated system 50

5.1. HTTP requests of the legacy and prototype system 60

6.1. Result of a FMEA analysis for the Kubernetes Ingress component 72
6.2. Description of basic events 79
6.3. Mapping of basic events to their occurence in the minimal cut set 80
6.4. Guidewords related to CHAZOP study 84
6.5. Result of a CHAZOP study of the interconnection between the Ingress and Registry 85

7.1. User thread group related property functions 94

8.1. Chaos experiment template 98
8.2. Experiment details of experiment no. 1 100
8.3. Standard statistics with respect to the observed phases of GET_CONFIG_ID operation, illustrated in Figure 8.3 104
8.4. Standard statistics with respect to the observed phases of GET_CONFIG_DATA operation, illustrated in Figure 8.4 105
8.5. Experiment details of experiment no. 2 109
8.6. Standard statistics with respect to the observed phases of GET_CONFIG_ID operation, illustrated in Figure 8.7 110

8.7. Standard statistics with respect to the observed phases of GET_CONFIG_DATA operation, illustrated in Figure 8.8 111
8.8. Experiment details of experiment no. 3 115
8.9. Standard statistics with respect to the observed phases of GET_CONFIG_ID operation, illustrated in Figure 8.11 116
8.10. Standard statistics with respect to the observed phases of GET_CONFIG_DATA operation, illustrated in Figure 8.12 117
8.11. Experiment details of experiment no. 4 121

List of Acronyms

AKS Azure Kubernetes Service

AWS Amazon Web Services

CHAZOP Computational Hazard Analysis and Operability

CLI Command Line Interface

ECDF Empirical Cumulative Distribution Function

FMEA Failure Modes and Effects Analysis

FTA Fault Tree Analysis

GUI Graphical User Interface

HAZOP Hazard Analysis and Operability

MTTF Mean Time to Failure

OS Operating System

URI Unified Resource Identifier

URL Unified Resource Locator

VM Virtual Machine

List of Listings

2.1. JSON based chaos experiment 22

6.1. HTTP POST request issued by a non-administrative user 73
6.2. Mutation of HTTP POST of Listing 6.1 73

7.1. Provisioning deplyoments for AKS cluster 91

8.1. Steady-state-hypothesis declaration according to Table 8.2 101
8.2. Method and rollback declaration according to Table 8.2 102
8.3. Kolmogorov-Smirnov test results related experiment no. 1 with respect to steady-state and injection phase of Figure 8.3 107
8.4. Kolmogorov-Smirnov test results related experiment no. 1 with respect to steady-state and injection phase of Figure 8.4 107
8.5. Kolmogorov-Smirnov test results related experiment no. 1 with respect to recovery phase of Figure 8.3 108
8.6. Kolmogorov-Smirnov test results related experiment no. 1 with respect to recovery phase of Figure 8.4 108
8.7. Kolmogorov-Smirnov test results related experiment no. 2 with respect to steady-state and injection phase of Figure 8.7 113
8.8. Kolmogorov-Smirnov test results related experiment no. 2 with respect to steady-state and injection phase of Figure 8.8 113
8.9. Kolmogorov-Smirnov test results related experiment no. 2 with respect to the recovery phase of Figure 8.7 114
8.10. Kolmogorov-Smirnov test results related experiment no. 2 with respect to the recovery phase of Figure 8.8 114
8.11. Kolmogorov-Smirnov test results related experiment no. 3 with respect to the steady-state and injection phase of Figure 8.11 120
8.12. Kolmogorov-Smirnov test results related experiment no. 3 with respect to the steady-state and injection phase of Figure 8.12 120

8.13. Kolmogorov-Smirnov test results related experiment no. 3 with respect to the recovery phase of Figure 8.11 120
8.14. Kolmogorov-Smirnov test results related experiment no. 3 with respect to the recovery phase of Figure 8.12 120
8.15. Potential method declaration of experiment no. 4 122

Chapter 1

Introduction

Improving the resilience of applications has become increasingly difficult due to the prevalence of microservice architectures. Resilience engineering or chaos engineering has been practiced by Netflix for the last eight to nine years based on their change of infrastructure [BBR+16]. In the last couple of years, chaos engineering has significantly increased in popularity which becomes evident by the increase of conference contributions and development of new platforms to leverage the principles of chaos [Greb]. Chaos engineering is expected to change the mindset of how systems should be developed with respect to their resilience.

Resilience is stated to describe a system's capability to recover from any kind of alteration particularly failures [VWAM12]. Although the term resilience does imply to constitute new techniques it does relate to already known engineering principles. Former approaches related to resilience are known as fault tolerance, fault avoidance, fault detection and are means to increase the dependability of a system. Dependability describes the interaction of multiple quality attributes such as availability, reliability, and safety. Thus, a system is called dependable if its available whenever its required, and it operates safely and reliably without any side effects [ORe17].

However, dependability increasing means yet fail to accurately adapt to continuous changes of a system. Software systems behave differently based on many criteria such as environment, technologies, developer skill and so on. Dependability increasing means are driven by a developer's capability of predicting different type of failures that may or may not happen. These methods yield high value in static environments which does not hold for any application running in a production-environment. Software application continuously experience changes especially in an agile environment, thus, making it difficult to predict failures. Hence, leading to the assumption that formerly stated techniques are insufficient to assess and increase the resilience of modern software applications.

Chaos engineering provides the possibility to address this issue by applying controlled and thoughtful experiments. Chaos engineering experiments are driven by the idea to intentionally inject failures into a system in a controlled manner in order to get a better understanding of a system's behavior [BBR+16]. The intentional injection of failures is reflected in the idea of performing fire drills. Fire men are continuously exercising their skills by exposing themselves to real-life situations within a controlled environment in order to be prepared for a real fire. Stated differently chaos engineering can be seen as a vaccine shot for a software system. By injecting harmful failures into a system the engineers have the ability to perform problem-solving tasks before failures become critical to the user thus building up a resistance against certain type of failures.

However, a major challenge is to decide which failures to inject into a system. Senior practitioners such as Netflix benefit from a rich pool of knowledge and a build up sensitivity against potential failures occurring in production. Beginners will struggle to benefit from an equal pool of knowledge, yet, existing solutions cannot be adopted easily since they are tailored to specific needs and systems. In context of an industrial case study we will evaluate the suitability of hazard analysis techniques, known from engineering safety-critical systems, as a means to identify potential risks on the architecture level based on an industrial case study application. After the risks and hazards have been identified the findings are implemented as chaos engineering experiments in Chaostoolkit [Chaa]. The chaos engineering experiments are then run against the case study application with the intention to surface vulnerabilities. At the end, the results are statistically analyzed by applying non-parametric hypothesis tests in order to observe potential deviations from the system's steady-state. In short, the objective of this case study can be stated as follows.

Analyze the planning and design of a Chaos Engineering experiment
for the purpose of providing a systematic approach to Chaos Engineering
respective to an industrial case study application
from the point of view of a Chaos Engineer
in context of a bachelor's thesis.

Summary of Method and Findings

In the following we will summarize the research method and findings of this work.

Method

- A case study plan has been elaborated following the principles by Wohlin et al. [WRH+12]. The case study plan comprises an objective, a description of the investigated case, a data preparation and collection plan, and a description of the data analysis.
- The information gathered in scope of this case study have been collected by applying semi-structured interviews as elaborated in the case study plan. The interview material has been provided as supplementary material. In total, three interviews have been conducted where each interview took approximately one hour and a half. The interviews were designed following the principles by Wohlin [WRH+12] and Runeson et al. [RH09]
- The interview findings were used to provide a clear interpretation and description of the investigated system's domain context. Further descriptions of the system include a detailed architecture description comprising component and sequence diagrams. The application's requirements were rephrased as user stories though being stated neutrally by the interviewees.
- Following the decision matrix for hypothesis tests provided by Field et al. [FH02] the appropriate hypothesis test would have been the Wilcoxon test, which is described in section 7.3 of [FH02]. However as suggested by Fielding et al. we decided to use the Kolmogorov-Smirnov test instead, as described in chapter 6 of [FH02].

Findings

- Based on the architecture description the system has been analyzed by applying three distinct hazard analysis methods, namely Failure Modes and Effects Analysis, Computational Hazard and Operations Study and Fault Tree Analysis. Due to the lack of missing quantitative information the probabilistic estimation of events have been left out. The analysis revealed the existence of multiple single points of failures implying that the system's architecture is considered rather fragile.
- A dedicated selection of the hazard analysis results were chosen by technical feasibility to be implemented as chaos engineering experiments in Chaostoolkit. In total, four experiments have been designed yet only three have been executed. The fourth experiment turned out to be technically infeasible but has been designed nonetheless. Therefore, the fourth experiment comprises only an experiment

description and possible future approaches to conduct the experiment. The steady-state description as well as the experiment results are a a set of response times measured by JMeter. During the experiments the response times were expected to change or not to change depending on the steady-state-hypothesis. The response times were measured for two interdependent HTTP requests, that represent the business logic of the investigated system.

- The response times were statistically analyzed by applying the non-parametric Kolmogorv-Smirnov test. The results were then discussed in the last chapter.

Supplementary materials are provided under the following link: https://doi.org/10.5281/zenodo.3374629

Thesis Structure

This thesis is structured as follows.

Chapter 2 – Foundations: – Covers the fundamentals required to understand the further content of this work.

Chapter 3 – Related Work: – Comprises related works in the area of chaos engineering.

Chapter 4 – Case Study Design: – Provides a case study plan that has been used to achieve the goal of this thesis

Chapter 5 – Cloud Service for Vehicle Configuration Files: – Covers an explanation of the domain context and a detailed architecture description comprising structural and behavioral diagrams with respect to the investigated system.

Chapter 6 – Hazard Analysis: – Covers a thorough analysis of the system by applying Hazard Analysis techniques, i.e., Failure Modes and Effects Analysis, Computational Hazard and Operations Study and Fault Tree Analysis.

Chapter 7 – Experiment Environment and Tool Configuration: – Describes the experiment environment where the chaos engineering experiments will be executed.

Chapter 8 – Chaos Experiments Design, Execution, and Analysis: – Comprises the implementation of identified risks and hazards as chaos engineering experiments as well as the execution and analysis of the experiments.

Chapter 9 – Conclusion: – Summarizes this work by discussing the findings and providing an outlook on future topics.

Chapter 2

Foundations

The content of this work combines methods of different software engineering domains such as hazard analysis and methodologies of case study research. In order to comprehend the methodologies, techniques, and taxonomy of this thesis, this chapter provides an introduction to these topics that are necessary to understand the further content of this work.

This chapter is structured as follows. Section 2.1 explains basic concepts of architecture and system design. Section 2.2 describes fundamental hazard analysis techniques. Section 2.3 summarizes research methods used throughout this work based on common case study research. Section 2.4 discusses the tools to perform data collection, data processing, execution of chaos experiments, and technologies that implement the investigated case study system. Section 2.5 explains the term resilience. Section 2.6 explains the chaos engineering methodology and its applications in software engineering. Section 2.7 provides an overview of the state of the art tools in the domain of chaos engineering.

2.1. Architecture and System Design

In the past years, authors such as Bass et al. [BCK03] and Perry et al. [PW92] argue that software architecture and design are not the same. Though Robert C. Martin points out that design and architecture should not be differentiated since both contribute to a system as a whole [Mar17].

Architecture can be explained as a system's description that encompasses its structure and behavior. An architecture comprises different views to comprehend different aspects of the architecture and guide communication between stakeholders. These views are meant to abstract away implementation details such as component attributes and functions

[BCK03]. For example a component diagram such as Figure 5.4 explains the structure of a system and its components as well as their relationships. This structural view is helpful for any developer involved in the development of the system but not beneficial for stakeholders who are not involved in the development process.

On the contrary design comprises decisions of more fine-grained details such as component modelling, attribute and method declarations, and so on [BCK03]. An example of design can be seen in Figure 5.7 illustrating message communication between different components of a particular system.

A more detailed definition of software architecture is given by the IEEE Standard 1471-2000 [MEH01]. According to this definition an architecture encompasses the fundamental organization of a system embodied in its components, their relationship to each other and to the environment [MEH01]. Hence, a software architecture commonly comprises of (1) a system's description of its architectural elements, (2) a set of different architectural views, and (3) a description of the system design comprising of fine-grained details such as component communication.

The architecuture of this case study's system given in Chapter 5 provides different architectural views and diagrams to explain the domain context and overall system structure, as well as design related views to emphasize different aspects of the system.

2.1.1. Architectural Styles and Patterns

Fielding et al. [FT00] describe an architectural style as a coordinated set of architectural constrains that narrows the possibility of how to align architectural elements and their allowed relationships, within a fitting architecture. Architectural styles encapsulate decisions and restrictions on its architectural elements and the relationships among the elements [FT00]. Examples of architectural styles are pipes and filters, client-server, microservices, and monolithic architectures. The latter two are referenced in Chapter 5 distinguishing the same system implemented with two different architectural styles. A monolithic architecture describes the structuring of components in such a way that the system is written as a single executable or single deployment file [Ric17]. This kind of architecture implies a tight coupling of single components.

On the contrary Newman et al. [New15] describes microservices as small pieces of code that autonomously work together. Newman's definition, intensively follows Robert C. Martin's single responsibility principle [Mar02]. The single responsibility principle states that multiple pieces of code that change for the same reason, should be grouped together. In contrast, pieces of code that do not change for the same reason, should be separated [Mar02]. For example, consider a web shop application implemented

with a microservice oriented architectural style. Within the web shop application there exist components responsible for user and product management, pricing, and order management. According to Martin's single responsibility principle, the user management component should not have to change if a new product is added to the web shop.

Patterns provide generic solutions that have been proven reliable to common problems that emerge from developing systems in practice [FLR+14]. Architectural styles as well as patterns can be categorized according to the problems they intend to solve. For example, the architectural styles client-server and microservices can be labeled as styles for distributed systems. Categories for patterns are typically distinguished on a higher-level such as architectural patterns, design patterns, cloud computing patterns, structural patterns, and behavioural patterns [FLR+14]. The design patterns by Gamma et al. [PG95] have become a well-known standard in the domain of Object-Oriented-Programming (OOP). The patterns given by Gamma et al. provide solutions for common problems that emerge in the development of system that use OOP languages such as Java [Oraa].

2.2. Hazard Analysis

With the expansion of systems and technologies to various aspects of our environment it becomes more important to consider the safety of systems and technologies [Eri99]. Sooner or later systems will experiences failures that potentially lead to hazards. The MIL-STD 882 [Sys] standard defines hazard as the following:

> A real or potential condition that could lead to an unplanned event or series of events (i.e. mishap) resulting in death, injury, occupational illness, damage to or loss of equipment or property, or damage to the environment.

Reasonably it is important to identify hazards of systems as soon as possible. A hazard analysis is given as a step within a process to determine a systems safety given by the MIL-STD 882. This section introduces three techniques to identify hazards in a system namely Failure Modes and Effects Analysis, Fault Tree Analysis and Computational Hazard and Operations.

2.2.1. Failure Modes and Effects Analysis

The following explanation is based on the work by Teoh et al. [TC04].

Failure Modes and Effects Analysis (FMEA) is used to identify failure modes of systems, sub systems and potential system effects of failures. A thorough FMEA comprises of

Part	Failure mode	Local effect	Next effects	End effect	Severity

Table 2.1.: Example FMEA for wind turbines [AOT10]

different sets of attributes that are captured during the analysis whereas some approaches extend the attributes by a quantification in terms of the criticality of failures. Commonly a FMEA is conducted by a team of experts of various departments who analyze single components of a system by the previously mentioned attributes failure modes, potential sub system and system effects and the criticality of these effects. In order to capture the findings, FMEA is commonly documented within a table where each column corresponds to one attribute. For example consider Table 2.1.

In this particular case the FMEA is done according to the analysis of wind turbines [AOT10]. The first column selects the part that is going to be analyzed. A part can be any component that system, in this case a wind turbine, comprises. The second column defines a potential failure mode. A failure mode is a description of a potential failure that may occur in the system. Any subsequent column refers to the failure mode determined in the second column. In the given examples a failure mode of a wind turbine is determined to be that the system is not able to cool down a specific amount of oil [AOT10]. The following three columns local effects, next effects and end effects describe effect the failure mode may have on the system on different levels. The last column, severity, defines a metric that determines the criticality of the failure mode based on its effects.

In conclusion the FMEA is a team-based qualitative approach to identify failure modes and their potential effects on systems. Documentation of a FMEA is commonly done as shown in Table 2.1. However it has to be considered that labels of columns may deviate based on the needs, use cases and the system. With a thorough FMEA it is possible to identify failure events that can later be used for a fault tree analysis.

2.2.2. Fault Tree Analysis

The following explanation is based on the work by Leveson et al. [LH83].

Fault Tree Analysis (FTA) is considered to be a technique to analyze a software system's safety. The result and goal of a Fault Tree Analysis is to identify environmental relationships and failures between events that may cause a safety failure of the software system. During the analysis a series of hazardous events connected by boolean expressions are constructed in a top down manner. The root of the tree describes the top level event to be analysed which is usually explained to be the loss event or the undesirable event. Any subsequent level of the tree contains nodes which describe events that are causes

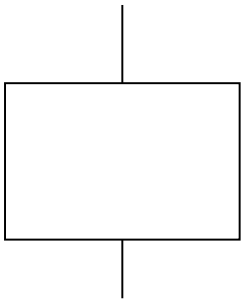

Figure 2.1.: Intermediate fault event

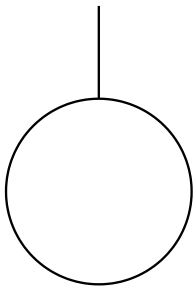

Figure 2.2.: Basic fault event

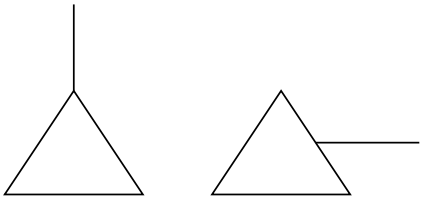

Figure 2.3.: Input and output events

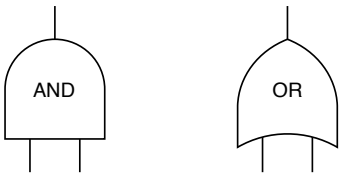

Figure 2.4.: Logic gates

for their parent node whereas each node is expanded by identifying possible causes. Each node is connected by an AND or an OR connection describing the relationship of two or more nodes. For example, if two nodes are connected by an AND connector, each node's event must occur at the same time in order for the parent event to happen. At the bottom of the tree are the leaf nodes which describe events of calculable probability or are unable to be further deconstructed for some reason.

In order to construct fault trees the event representation by Lee et al. [LGTL85] is used which is illustrated in Figure 2.1, Figure 2.2, Figure 2.3 and Figure 2.4. It has to be considered that there exists more representations to construct a fault tree but the objects previously mentioned are the only ones used throughout this work.

Figure 2.1 illustrates an intermediate fault event. Such events occur due to the combination of other fault events connected through a logic gate [LGTL85]. Figure 2.2 shows a basic event. Usually basic events occur on the bottom of any FTA describing events that are not required to be further analysed for some reason or describing events of calculable probability [LH83]. Figure 2.3 show two input events. The left event is called an input event that takes some kind of input that may later be used in the analysis, e.g., a subtree. The right event is called an output event denoting that some input may used at this particular place, e.g., a subtree [LGTL85]. Lastly Figure 2.4 illustrate logical gates that connect events. Events connected by an AND gate occur if both child node events occur at the same time. On the other side, events connected by an OR gate occur if either of the child node events occur.

Next to the graphical modelling approach any fault tree can also be analyzed qualitatively by determining minimal cut sets or minimal paths that lead to the top-level hazard [LGTL85]. Hence, a minimal cut set is the minimum number of events that are allowed to happen in order for the top-level hazard to occur. There exist various approaches to determine minimal cut sets such as monte carlo simulation procedures and other deterministic methods [LGTL85]. Other than that, due to the logical gate connection of each event, minimal sets can also be determined by application of basic boolean algebra. The goal is to apply these principles until an expression is reached that looks similar to this:

$$(A_1 * A_2 * ... * A_i) + (B_1 * B_2 * ... * B_j) + ...$$

where A_i and B_j with $i, j \geqslant 1$ describe basic events in a fault tree. Following the convention of boolean algebra, multiple events connected by an AND gate are expressed as:

$$(A_1 * A_2 * A_3 * ... * A_i),\ i \geqslant 1$$

Multiple events connected by an OR gate are then expressed as:

$$(B_1 + B_2 + B_3 + ... + B_j),\ j \geqslant 1$$

The minimal cut sets in this work are entirely determined by the application of boolean algebra. A quantitative evaluation of any fault tree labels basic events with probabilities. Therefore, leading to a probabilistic estimate of the basic event's occurrence. Such an evaluation requires knowledge about the statistical expectation or probability of each basic event and is not further used in this work due to the lack of necessary data.

2.2.3. Computational Hazard and Operability

Computational Hazard Analysis and Operability (CHAZOP) is an extension to the Hazard Analysis and Operability (HAZOP) study known from the chemical process industry [FH94]. HAZOP refers to a systematic study which identifies deviations from a given design specifications of a system and whether these deviations may result in hazards [WJG01]. Conceptually the HAZOP study is similar to the FMEA approach based on the fact that it is a qualitative team review activity. It is recommended that the team conducting the study should have sufficient experience with assessing a system's safety

Interconnection	Attribute	Guideword	Cause	Consequences
Train radio station to train radio exchange server	alteration	other than	train ID is altered by manipulation of communication links	Call information is wrong

Table 2.2.: Example of the use of guidewords in case of a train telephone system [WJG01]

as well as a deep knowledge of the studied system [WJG01] [DFVA10]. The knowledge and experience of the team therefore decides the quality of the study.

During a HAZOP study a interconnection or flow between two units is selected that is to be analyzed. This interconnection is then further analyzed by applying a combination of attributes and guidewords [DFVA10]. Common guidewords are (1) no, (2) more ,(3) less, (4) as well as, (5) part of, (6) reverse, (7) other than, (8) early, (9) late, (10) before and (11) after [WJG01] [DFVA10]. Attributes and guidewords vary based on the use case and the people who conduct the study. For example consider Table 2.2.

Table 2.2 illustrates a small example of a HAZOP study by using the (7) guideword in combination with the attribute alteration. In this case the interconnection is selected to be between two components which illustrate the communication between a train radio station and a train radio exchange server. The attribute alteration describes briefly what occurs during the communication process between both parts. The guideword other than describes that the delivered action is other than the design intent which is further described in the cause column. The last column indicates possible causes.

HAZOP provides meaningful insight in interconnections and data flows between small units of a given system to identify hazards and design deviations. In case of assessing the safety of software system's the list of guidewords may be extended or changed to some degree. Additionally some attributes are considered to be more meaningful, e.g., using the attribute data flow in combination with the (1) guideword to identify deviations in the data flow between two units. A thorough application HAZOP is given in Chapter 6. The list of guidewords used in Chapter 6 is equal to the guidewords explained in this section.

2.3. Research Methods

The definition of case study research varies among authors such as Yin, Wohlin, and Runeson et al. [Yin98] [WRH+12], [RH09]. However, Robson [RM16], Yin [Yin98]

and Benbasat et al. [BGM87] agree, that a case study can be described as an empirical method that aims at investigating a contemporary phenomena in its natural context [WRH+12]. A case study is referred to as being an empirical research method which is 1 out of 4 essential research methods identified by Wohlin et al. [WRH+12], namely scientific, engineering, empirical, and analytical. Therefore, a case study leverages empirical research strategies to achieve its study objective.

2.3.1. Exploratory and Explanatory Research

Empirical research strategies further divide into two paradigms, i.e., exploratory research and explanatory research [WRH+12]. Exploratory research studies objects and phenomena in their natural context combined with a flexible research design in order to adapt to the changes, that occur in a natural setting [WRH+12]. Moreover, qualitative research and inductive research are strategies which are associated with exploratory research. Explanatory research is concerned with identifying cause-effect-relationships between two observed objects or groups of people [WRH+12], [RH09]. In contrast to explanatory research, a fixed research design is applied during explanatory research. Commonly controlled experiments are conducted during such research approaches. Quantitative research and deductive research are associated with explanatory research. According to these explanations, a case study is associated as a exploratory research utilizing qualitative and quantitative approaches to gather information about a specific case.

2.3.2. Case Study Planning

Before a case study is executed, it is suggested to provide a case study plan. A case study plan comprises of (1) an objective, (2) a description of the investigated case, (3) an explanation of the data that needs to be collected, (4) which data collection strategies are used and (4) how the data is going to be evaluated [RH09], [WRH+12]. The objective is formulated to point out the focus of the case study. A set of research questions indicate, what goals have to be fulfilled in order to achieve the study objective. The objective can either be exploratory, explanatory, descriptive, or improving [WRH+12]. Stated by Runeson et al. [RH09] a case is an object or a phenomena to be studied for an identified reason in its natural context. A description of the case is necessary in order to understand the phenomenon that is investigated and its context.

The data collection strategy must fit the data that has to be collected. For example, if a system is analyzed as part of the study and the only source of information are the stakeholders, than it is the researchers obligation to use meaningful qualitative strategies

such as interviews or focus groups to obtain the data. Moreover, as part of the data evaluation it has to be considered that a case study does not provide a conclusion with statistical significance [WRH+12]. Though a clear chain of evidence is provided by collecting evidence from multiple sources of information. This information may either be collected qualitatively or quantitatively with respect to the data. The triangulation of multiple information sources leads to a less biased and stronger conclusion, than drawing a conclusion from a single information source [WRH+12]. A thorough case study plan is created during the course of this thesis. The case study plan can be found in Chapter 4 comprising of the previously mentioned contents of a case study plan.

2.3.3. Qualitative Research Strategies

Qualitative research refers to any type of research that concerns about findings that are not derived by statistical procedures or any other means of quantification [SC98]. Typically such findings include research about person's lives, experiences from projects, tracking emotions and feelings or organizational movements and social changes [SC98]. This work applies qualitative research strategies in order to gather data and information about the investigated software system introduced in Chapter 5. In the following the major strategies are explained and evaluated.

Interviews

During an interview, the researchers asks a series of questions to subjects or interviewees. The goal of an interview is to collect information about the areas that are of interest for the case study [WRH+12]. Interviews can either be performed with one subject or multiple subjects. The latter being a group interview. Furthermore interviews can have different structures respective to the sequence of questions asked, and the structure of the questions. In general, interview questions are derived from the research questions but not stated as precisely [WRH+12].

Interview questions can be open or closed. Open interview questions allow a more broad range of answers by not limiting the subject to predefined answer options [Poh10]. Closed interview questions, provide a predefined set of equal or different answers for each questions. Hence, the subject can select one of the available options and is thus limited in his or her response [Poh10].

In general there are three types of interviews namely fully-structured, semi-structured, or unstructured interviews [WRH+12]. Fully-structured and semi-structured interviews have their questions prepared in advance [WRH+12]. Though during a fully-structured interview questions may have a pre-defined sequence where each participant is asked

the same questions in the same sequence. In contrast semi-structured interviews do not require the interviewer to ask questions in the same sequence. Moreover, the interviewer has the possibility to deviate from his or her interview questions. This allows a more fluent and broad conversation, where the interviewer has also the possibility to direct the conversation towards a more specific topic, that may not be covered by a prepared set of interview questions [WRH+12].

Unstructured interviews have interview questions stated as major interests of the interviewer of general topics of concern, rather than strictly specified questions. Based on this, the conversation develops towards the interests of the researcher and does not follow any questions catalogue.

Focus Groups

Focus groups are a special composition of subjects in terms of size, purpose, composition, and procedures [KC14]. The goal of focus group is to raise information about how the subjects feel or think about a specific item, service, idea, or product [KC14], [Poh10]. Krüger et al. [KC14] identifies five essential characteristics of focus groups. Therefore, a focus group comprises of a small number of people (1), who share a certain characteristics in common (2). The people provide qualitative data (3), within a focussed discussion session (4), to help understand the topic of interest (5). The characteristics identified by Krüger et al. [KC14] imply that, the most important source of data during a focus group session, is the discussion among participants.

However, focus groups and group interviews share the same weakness. Both methods are affected by the groupthink effect. Janis et al. defines the groupthink effect as a mode of thinking that overshadows people's capability of realistically evaluate one course of action taken over another by their strive of unanimity [Jan08]. This especially occurs if people are deeply involved in cohesive groups.

Hence, the groupthink effect lowers the quality of ongoing or expected discussions within group interviews and especially in focus groups. If chances are high that the groupthink effect might occur, it is recommended to separately interview less dominant participants from more dominant participants, or choose another elicitation method [Poh10].

Questionnaires

Questionnaires have the advantage that, researcher and participants are not required to meet in person. The questionnaire can be delivered and evaluated via an appropriate

online platform or provided by e-mail. As in interviews, the questions can either be closed or open [Poh10]. Additionally, there is no strict time constraint for questionnaires. Within a given time frame where the questionnaire has to be finished, this can be several days or even weeks, the participants are free to choose the location and time when they want to answer the questions. This gives the participants a higher leverage of control when to answer the questions and more time to reconsider answers to more complex questions. However, in contrast to interviews or focus groups the researcher has little to no control over the quantity of information. For example, during a semi-structured interview or focus group, the researcher has the possibility to direct the discussion or conversation into more specific direction respective to his or her interests. This is not given within a questionnaire since the researcher and participants are decoupled from each other.

Summary

The methods discussed in this section have different goals next to collecting information from people. Focus groups heavily rely on the aspect of continuing discussions among the participants in order to let new insights emerge. Questionnaires restrict the participants in such a way that, a limited sets of answers is provided, leaving the participants no chance to further elaborate their thoughts. Interviews enable a more broad, flexible, and exploratory approach to raise information during a one to one session with participants.

2.3.4. Quantitative Research

The following explanations are based on the work of Wohlin [WRH+12] and Runeson et al. [RH09].

Quantitative research concerns about the quantification of data typically by application of descriptive statistics. This includes correlation analysis, hypothesis testing, calculation of mean values and standard deviations. Commonly as stated by Wohlin and Runeson et al. a quantitative research approach assume a fixed research design due to data becoming impossible to interpret if changes to the data happen as common in a flexible research design. However case studies make use of a combination of qualitative and quantitative research strategies. For example by qualitatively gathering data about a investigated case and then conducting a quantititave analysis that comprises hypothesis testing.

Hypothesis testing aims to answer the question if a null hypothesis will be rejected or not. A null hypothesis H_0 is commonly stated negatively in contrast to the alternative

Test name	Description
t-test	Most often used parametric test to compare two sample means
Mann-Whitney	Non-parametric test alternative to the t-test
Analysis of Variance (ANOVA)	A set of parametric tests for designs with more than two levels of a factor

Table 2.3.: Examples of parametric and non-parametric hypothesis tests

hypothesis H_1 whereas H_1 describes the initial hypothesis that is going to be investigated. Furthermore H_0 describes properties of some statistical distribution from which a sample is drawn where the researcher investigated whether to reject that the properties a true with respect to a given significance α.

In general hypothesis tests are classified into parametric and non-parametric tests. Parametric tests often require the model to be given as a specific distribution. For example, the parametric t-test requires the model to be normally distributed. Non-parametric tests are more general than parametric tests and do not require the data to be given as a specific model. They can be used whenever a parametric test is not applicable or suitable to the investigated purpose. For example, chi-2 is a non-parametric tests that can be used if the data is provided as frequencies. In this case a parametric test is not applicable since a parametric test requires the data to be measurable on an interval scale in addition to be normally distributed. Table 2.3 shows some hypothesis tests commonly applied in practice.

The significance level α of a test, often stated as the $p - value$, describes a probability where if any measured value occurs within the given $p - value$, H_0 will be rejected in favor of H_1. In most cases the $p - value$ describes an area below a given distribution. Common values for $p - value$ are $\alpha = 0.05$ or $\alpha = 0.01$ dependent on the investigated case and the trustworthiness of measured data.

In this work a non-parametric test, the Kolmogorov-Smirnov test, is applied in order to analyze two distributions with respect to measured response times of the investigated software system. The test was selected by following Field's et al. decision matrix [FH02].

2.4. Technologies

This section gives an introduction to the technologies that have been used for the implementation of the case study application, which is described in Chapter 5. Furthermore,

two measurement instroments will be introduced, namely Apache JMeter and Chaostoolkit. These instruments are going to be applied in Chapter 7. The application is build upon three core technologies namely Spring Boot, Docker for the usage of container virtualization, and Kubernetes as a deployment platform and container orchestration.

2.4.1. Spring Boot

Spring Boot provides the possibility to create stand-alone and production-grade ready applications without concerning about any underlying configuration details [Piva]. Spring Boot has become the de-facto standard for implementing JVM-based microservices. Additionally it takes control of the generation of typical boilerplate code so that developers can solely focus on integrating business process features [Piva]. Some of its features are:

- Provide production-ready features such as metrics, health checks and externalized configuration
- Create stand-alone Spring applications
- Embedded Tomcat, Jetty or Undertow

Spring has grown to become an entire platform for building enterprise applications. Spring Boot as well as Spring Cloud, and Spring Cloud Data Flow are also projects build on top of the Spring framework. Driven by the 'convention-over-configuration' paradigm, Spring makes the development process easier by leaving configuration work to Spring and thus reducing the amount of boilerplate code. This enables developers to entirely concentrate their effort on the development process [Piva]. Spring builds on top of three core features [Piva]:

- Dependency Injection
- Aspect-Oriented-Programming
- POJO-based programming model

The Spring framework contains many more projects which focus on different application aspects such as Message Queuing, Data Persistence, Cloud Deployment, and Cross-Cutting-Concerns.

2.4.2. Docker

Container virtualization exists since a very long time but was not easily accessible for everyone. In 2014 Docker provided an engine to easily isolate applications in containers and make the roll-out process quicker. Since then, the concept of container isolation has become more popular and Docker has become the de-facto standard for container virtualization [Doc]. Containers work extraordinary well within microservice environments since each service can be developed and isolated in its own container supporting the mind-set of the single-responsibility principle of microservices. Each service runs in its owner environment and thus, makes the development and deployment process easier since each container is isolated from other containers.

2.4.3. Kubernetes

Kubernetes is a container-orchestration tool to efficiently manage containerized applications [Fou] . Moreover, Kubernetes enables engineers to scale and deploy their applications more efficiently. But Kubernetes requires applications to be containerized, namely running inside a container. This can be achieved by tools such as Docker and rkt (spoken Rocket) [Fou].

Kubernetes provides a cluster which is maintained and run by Kubernetes itself. A Kubernetes Cluster is a combination of multiple Virtual Machines or physical devices that act as a single unit of computation. A Kubernetes Cluster is visualized in the Figure below.

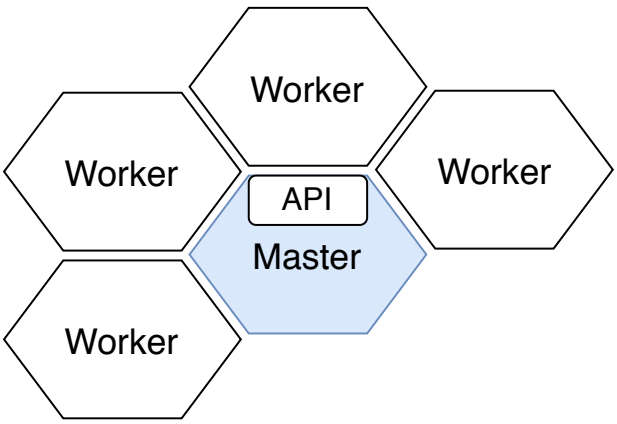

Figure 2.5.: Organizational structure of a Kubernetes cluster [Fou]

The Kubernetes Cluster Services are responsible for running the cluster and maintaining its state. To do so, the Cluster Services permanently verify that the current cluster state

matches the desired cluster state as specified within a deployment file. This allows Kubernetes to provide a self-healing mechanism since the Cluster Services detect if a worker node goes down automatically initialize actions to get back to a working cluster state. A deployment contains information for the Cluster Services such as the number of nodes and pods inside the cluster. A pod is the smallest, scaleable unit in Kubernetes and contains either a single or multiple containers [Fou]. Whereas a node is either a Virtual Machine (VM) or a physical device that is responsible for running an application inside a pod [Fou].

Kubernetes was developed in order to automate deployment, scaling and management of containerized applications. Additionally, Kubernetes provides self-healing mechanisms in such a way that it continuously verifies that the current cluster state is equal to the desired state. Therefore, if critical nodes, pods or containers terminate, Kubernetes re-starts the terminated objects to restore the desired cluster state.

2.4.4. Apache JMeter

The following explanations are based on the official Apache JMeter documentation [The19b] and homepage [The19a].

Apache JMeter is a java based application that allows to load test functional behavior and simulate user workload of applications. As stated by Vieria et al. [VWAM12] a workload represents an event the application is exposed to such as database queries. Apache JMeter feature set allows to performance test applications, servers and protocols of different kinds such as REST web services.

As a means to load and performance test an application, JMeter provides a graphical user interface to generate test plans and a command line mode to execute them. A test plan comprises elements which allow the customization of work load. A minimal test comprises a test plan, a thread group and one or more samplers. The test plan describes the test scenario. During the execution of the test plan, a thread group and multiple samplers are executed in a defined order to generate the specified work load. For example, a thread group may contain 5 concurrent users that execute multiple HTTP []requests with respect to a web service in assistance of samplers.

Number of Threads (users) is the total amount of users that will be created during the test period and execute commands that are configured by different test plan objects, e.g., samplers. The *Ramp-Up Period (in seconds)* declares a time-interval that JMeter uses to start all previously defined threads. For example if the *Ramp-Up Period (in seconds)* is set to 100 seconds and the *Number of Threads (users)* is set to 10 then JMeter will take $\frac{100}{10} = 10$ seconds to start all threads within the given thread group.

A thread group defines the number of concurrent users that execute different tasks according to the test goal. For example, if a database is required to be load tested, a thread group may set up a total of twenty users where each user executes a specific database query. A thread group and any other element must be assigned to a test plan object. Moreover, a thread group assigns one or multiple controllers to execute different tasks. Samplers are a special kind of controllers. JMeter provides an additional logical controller element to enable a combination of multiple samplers for more complex scenarios.

Samplers provide access to send requests to a server, e.g., sending an HTTP request to a remote server. If multiple samplers are defined within a thread group, JMeter executes them in the order they appear in the test plan. Each sampler can be modified with respect to its number of repetitions by a controller. For example, a controller can be used to define that sampler is repeated 5 times. A sampler can be combined with an assertion element. An assertion can be used to verify responses from a server, e.g., status code sent inside a response header is equal to HTTP status code 200.

In this work, a test plan will be created to create environmental conditions similar to the production environment of the prototype system explained in Chapter 5. The environmental condition simulates the expected user behavior of the application based on the interview findings discussed in Section 5.1.

2.4.5. ChaosToolkit

The following descriptions and explanations are based on the chaostoolkit documentation [Chab] and homepage [Chaa].

ChaosToolkit is a python based, open source platform provided by ChaosIQ [Cha19] which is extensible through its open API. The chaostoolkit aims to easily adopt chaos engineering by providing full integration of chaos experiments. In addition to its rich feature set of executing chaos experiments, the chaostoolkit provides different drivers to inject chaos into different abstraction levels, e.g., a driver for Kubernetes to create chaos within a Kubernetes cluster.

In the scope of chaostoolkit experiments are defined by creating YAML or JSON based files which contain different objects that make up the experiment. It is distinguished between must-have and optional objects. As the name implies must-have objects must occur in each experiment declaration, e.g., a steady-state-hypothesis. Optional objects must not occur inside an experiment declaration, e.g., rollbacks. The different objects are shown in Table 2.4. The chaostoolkit further describes objects that may be included in an experiment. However these objects are not sufficient for this work and are hence

MUST	
Object name	Description
version	Version of the current experiment file
title	Experiment title
description	Description of the experiment
method	Comprises of actions and probes to change the system state and verify the hypothesis
SHOULD	
Object name	Description
steady-state-hypothesis	Defining the expected working behavior of the application under test
rollbacks	Actions to reverse the actions of the method block

Table 2.4.: Chaos experiment objects declared by ChaosToolkit

not included in Table 2.4. Note that the order of the objects given in Table 2.4 is the same order the objects must appear in the document.

Changes in the system are created by providing actions inside the method block. An action is a JSON object which performs specific operations against the system, e.g., terminating a Kubernetes pod that hosts an application. An action must declare the properties (1) type, (2) name, (3) provider and (4) controls.

In order to verify the steady-state-hypothesis a chaos experiment may include probes. A probe is another JSON object that collects information from the system during an experiment. Probes do not change the system like actions, they merely collect information based on their provider. Like the action a probe must declare the properties (1) type, (2) name and (3) provider.

Probes can be included in the steady-state-hypothesis block in order to define a working behavior, e.g., requesting a service by issuing a http request to a given URL. The method block may as well include probes in order to collect information about the system state.

Listing 2.1 illustrates a sample experiment declared as a JSON file. If executed the chaos experiment will first verify the steady state hypothesis by applying the probes declared in the steady-state-hypothesis block. The probe inside the block makes a http GET request to localhost and expects that the application responds with a status code 200. In this case this serves as a verification that the application is still up and running.

Listing 2.1 JSON based chaos experiment

```
{
      "version": "1.0.0",
      "title": "Example experiment",
      "description": "Application is still available despite termination of a pod",
      "tags": [
      "thesis",
      "kubernetes"
   ],
      "steady-state-hypothesis": {
            "title": "The app is up and running",
            "probes": [
         {
            "name": "app-responds",
            "type": "probe",
            "tolerance": 200,
            "provider": {
               "type": "http",
               "url": "http://localhost:8080/hello"
            }
         }
            ]
      },
      "method": [
            {
         "type": "action",
         "name": "terminate-pods",
         "provider": {
            "type": "python",
            "module": "chaosk8s.pod.actions",
            "func": "terminate_pods",
            "arguments": {
               "label_selector": "app=spring-app",
               "ns": "default"
            }
      },
      {
            "ref": "simpleweb-responds"
      }
      ],
      "rollbacks": []
}
```

Next the method block is executed. Actions that change the system state are applied inside this block. The action defined in Listing 2.1 uses a python module provided by the chaostoolkit Kubernetes plugin. The action will look for a specific pod inside the cluster and terminate it. After the action has been applied, the steady-state-hypothesis block is executed again in order to verify that the hypothesis still holds despite the applied changes in the system state.

Since there are no rollbacks defined, the experiment will stop and print the result to the console. Based on the result, chaostoolkit will either respond with the experiment being successful or not. In case of a successful experiment, the system worked as expected.

The experiments created in this work follow the explanation in this section. All experiments created during this work will be declared as JSON files and explained in Chapter 7. In addition to the core functionalities of chaostoolkit, the Kubernetes driver will be used in order to inject chaos to a Kubernetes cluster. The respective functionalities are going to be explained in Chapter 7.

2.5. Software Resilience and Dependability

Resilience has grown to become a widely adopted term in various research areas. The following section shows, that the meaning of the term resilience varies in relation to different domains such as psychology and software engineering. Thus, it is necessary to understand its usages as well as its relation to other engineering concepts such as dependability, in order to know which measures appropriately characterize resilience.

2.5.1. Resilience and Dependability

Resilience is considered as a quality attribute such as availability, maintainability and durability which describe quality aspects of a software system. In order to measure or assess the resilience of a system it is required to clearly understand what resilience is about and what it actually means [VWAM12]. In a non-technical context resilience is described as the power or ability to return to the original position after being bend, compressed, or stretched [Ran]. Another definition is given by Vieira et al. [VWAM12] where resilience is described as an object's ability to recover or returning to a normal state after being altered or deformed by some kind of pressure.

Viera et al. [VWAM12] argue that the increasing usage of the term resilience creates a doubt whether it is just a new term to highlight already known concepts such as fault tolerance, reliability, availability, maintainability and security or if it actually identifies

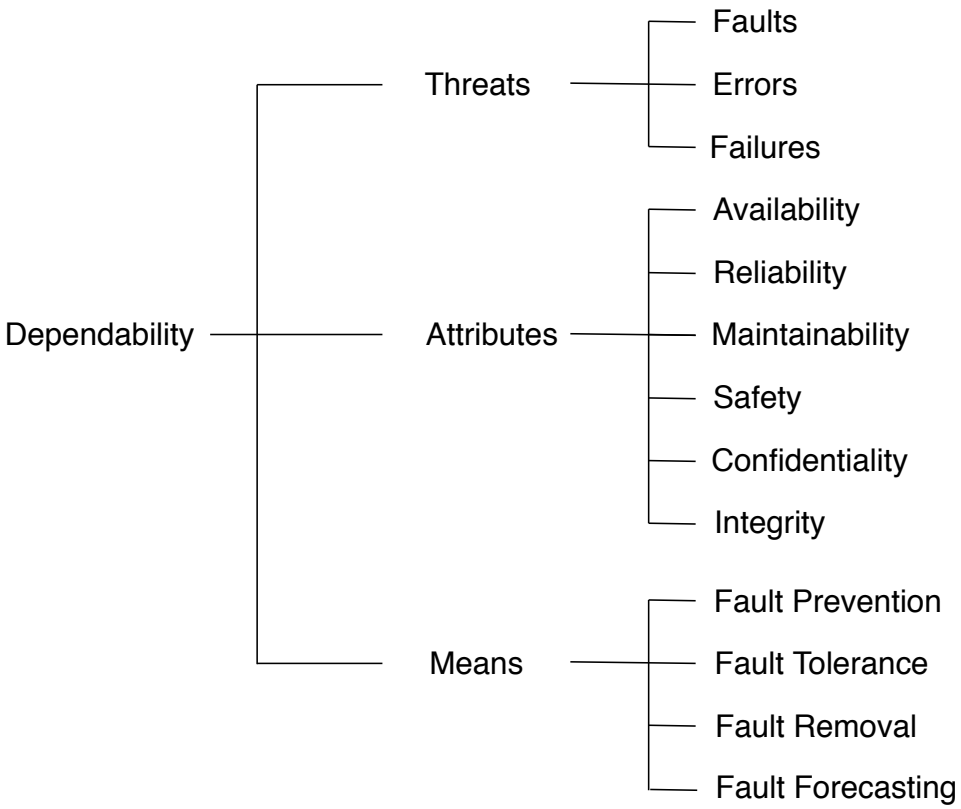

Figure 2.6.: Components of dependability [ALR+01]

new concepts. These concepts relate to dependability. A system is called dependable if its available whenever its required, and it operates safely and reliably without any malicious side effect [ORe17]. Figure 1.1 shows a systematic representation of the dependability concept.

It is illustrated that dependability has three main components: the threats to dependability, the attributes of dependability, and the means by which dependability can be achieved [ALR+01]. Availability and reliability are quality attributes users unknowingly experience every time they use an application and thus affecting the user experience. This structure implies that by changing the attributes, the dependability is also affected and vice versa. Whereas threats negatively affect the dependability but can be dealt with the described means. Availability and reliability are quality attributes engineers are looking to improve next to performance and resilience. Therefore, it is needless to say that both of these are of major importance respective to improving an application's dependability.

2.5.2. Fault Avoidance and Fault Tolerance

With respect to the threats shown in Figure 1.1, fault tolerance and fault avoidance have been mentioned by [VWAM12] where it was claimed that resilience is related to these concepts. Fault tolerance and fault prevention or fault avoidance are techniques where each of them comprises of different strategies with capability to increase the Mean Time to Failure (MTTF) of a system. MTTF describes the fraction of time from a working state of a system until an error occurs [SI03].

Fault avoidance aims to reduce the number of possible defects in software programs by various approaches such as precise specification practices, information hiding and encapsulation [Sah11]. Whereas fault tolerance means to make the system able to tolerate the effects of faults occurring in the system [VWAM12]. Dependability increasing techniques such as fault avoidance or fault prevention are applied during the software development process in order to reduce the number of faults introduced during construction [Pul01]. Avoidance or prevention techniques address specific issues respective to specification, software design methods, system requirements and reusability [Pul01].

Strigini [Str12] explains that these techniques and strategies might become inadequate due to the appliance of patterns such as loose coupling and microservices, making computing systems growing and changing as time progresses. These changes are prone for introducing new faults and errors which can not been foreseen most of the time by designers which are responsible for applying fault avoidance strategies.

2.5.3. Fault Injection

Fault injection is a common and well-known technique for researching the dependability of an application. Fault injection can either be applied to the software level or the hardware level [HTI97]. Both approaches differ in their type of injection or level of injection. Fault injection is used to introduce failures into the system in order to investigate the dependability of the system. As mentioned in previous sections, dependability comprises of different attributes, means and threats. Fault injection is one mean to research dependability as illustrated in Figure 1.1.

Moreover, fault injection has been used as an experimental approach by various engineers. Chaos Engineering applies fault injection during production time. Therefore, it can be implied that fault injection is an important technique used by Chaos Engineering. However, compared to the appliance of fault injection within a chaos experiment, common fault injection approaches are only applied during conception, design or testing stages [HTI97]. Whereas chaos experiments explicitly apply fault injection during the production stage.

Nonetheless, fault injection is a valuable technique used within a chaos experiment. Whereas fault injection can either be applied to a hardware- or software level, chaos engineering solely injects failures on a software level to simulate real world scenarios in order to get a better understanding of the application's behavior. Chaos engineering tools use the concept of fault injection to inject failures into a system in a controllable way, e.g., injecting delayed service calls to a component. The main purpose of fault injection is preserved but it's application area has shifted from an isolated environment to the more lively production environment.

Resilience is related to dependability and must therefore be treated as a dependability affecting technique or concept. Furthermore, Date et al.[Dat08] identifies a meaningful extension of fault tolerance by describing resilience as the ability of a system to deliver, maintain and improve service when facing threats and future changes. Consequently, improving a system's resilience makes the system more fault tolerant when facing threats and future changes. The affiliation with dependability leads to the assumption that other dependability affecting measures may be of interest when assessing resilience. The following Section explains the principles of Fault Injection as a means to inject faults and failures into a system.

2.6. Chaos Engineering

This section explains the main concern of this thesis. Chaos Engineering is a powerful method to improve the resilience and the confidence of large scale distributed systems. Chaos Engineering, formerly known as Resilience Testing, has been established and pushed forward by Netflix growing into a large culture of efficiently assessing resilience of large scale distributed systems.

The previous section explained that benchmarking and resilience benchmarking as well, is executed in an experimental-like manner. Chaos Engineering in particular adapts the experimental nature by executing chaos experiments during a system's production stage. Chaos Engineering can be viewed as an approach of resilience benchmarking and is therefore a methodology to effectively assess the resilience of a system. Chaos Engineering as well as the term "Chaos" are often misinterpreted as randomly injecting chaos into a system in order to break it. This is not necessarily true. Moreover, Chaos Engineering is viewed as an empirical approach to systematically surface weaknesses by exposing the system to turbulent situations [BBR+16]. The exposure of these situations takes place within controllable experiments that are conducted most efficiently in a production environment [BBR+16]. More precisely, chaos engineering can be defined as a principle which describes the execution of thoughtful, planned experiments designed to reveal weaknesses in a system [Med18].

Chaos Engineering grew in popularity since common benchmarks are mostly ad hoc and lack the possibility to efficiently assess a distributed system's resilience [HADP18] [VWAM12]. Although, they can be used to assess a software system's resilience, benchmarks struggle with one of a kind software systems by lacking domain-specific benchmarks as it was discussed by Gray [Gra93]. Thus, chaos engineering fills the gap by providing a meaningful possibility to give information about a system's resilience as well as surfacing weaknesses that put the system's dependability attributes at risk.

2.6.1. Chaos Experiments

Essentially, chaos engineering is about executing chaos experiments in production. During a chaos experiment, chaos variables are created and purposely injected to the system under test. These variables represent real events such as server crashes, service malfunctions, delayed response times between service calls and so on. The number of variables and their values are unlimited. The goal is to find possible weaknesses within a system before they cause serious harm in production and thus possibly affecting the user. Therefore, reducing the number of preventable outages and improving confidence in a system. Basiri et al. [BBR+16] state that, each chaos experiment is driven by the advanced principles for chaos engineering experimental design. The following steps have to be taken:

- Hypothesize about steady state
- Vary real-world events
- Run experiments in production
- Automate experiments to run continuously
- Minimize blast radius

A steady state refers to a state of the system in which the system is considered working as expected. Therefore, the system provides its implemented functionality to their full extent. The steady state of a system varies for each system. A system's steady state can be expressed as a set of metrics or attributes of a system such as the number of incoming HTTP requests, the number of running Docker containers, and so on. In order to execute a meaningful experiment, a hypothesis has to be stated about the steady state of the system. Usually, these hypothesis are stated in a form like "The events we are injecting into the system will not cause the system's behavior to change from steady-state" [BBR+16].

The core of each experiment is the injection of chaos variables or real-world failure scenarios. The events injected are driven by the motivation of the chaos experiment,

meaning that they depend on the weakness or the part of the system that is under observation. Failure scenarios could be realized as server crashes, service malfunctions, delayed service calls or injected latency. Each failure scenario should represent an event that is likely to occur in production, thus there exists a large number of scenarios that can be injected during the experiment. However, it is not necessary to enumerate all possible events but merely injecting those events that are frequent and harmful to the system.

Just running a failure scenario without performing any kind of monitoring or measurement is meaningless. Each experiment aims to measure some kind of effect either supporting the stated hypothesis or disproving it. In order to know if the hypothesis about the steady state holds or not, it has to be measured if the steady state deviates from its values. The deviation of the metrics considered for the steady state, provide good evidence that the failure scenario injected, has some kind of impact on the system. In the end, this impact on the system is the effect that engineers aim to measure. If the failure scenario injected has a serious impact on the system, then there should be a large deviation of the steady state which means, that there might be a potential weakness in the system.

One core aspect of chaos engineering is running experiments in production. This might sound contrary since during a chaos experiment the system will intentionally be exposed to turbulent situations. Therefore, it is important to minimize the blast radius of an experiment. The blast radius is an area of effect which a chaos experiment causes on the system, e.g., injecting latency to a service call causes multiple components to go down and thus shutting down the entire system [BBR+16]. Minimizing the blast radius refers to the technique of identifying weaknesses of the system without accidentally breaking the entire system [BBR+16]. This should always be kept in mind during the appliance of chaos engineering.

The last step in applying chaos engineering is to automate experiments in the long run. Automation plays a huge role in software engineering and makes uniform, repeatable tasks easier by executing them autonomously. At first it is suggested to manually run small chaos experiments but as the size of the experiments and the confidence of the engineering team increases, chaos experiments should be automated.

2.6.2. Chaos Engineering at Netflix

With their migration to the cloud in 2008, Netflix realized that they have to design a cloud architecture where individual components can fail without affecting different dependability attributes of the system [BBR+16]. As a solution Netflix has adopted resilience testing in order to keep improving the resilience of their cloud infrastructure.

Over the years Netflix's approach to resilience testing has become to be known as chaos engineering. With a large microservice landscape that resides in the cloud, Netflix and any other company that adopts the microservice-architecture pattern, gains velocity and flexibility but at the cost of human understandability. This lack of understandability creates the opportunity for chaos engineering [BBR+16]. Later, Netflix invented and introduced the Chaos Monkey [IT11] tool which, at first, randomly selected Amazon Web Services (AWS) and shuts them down. Over the years, Netflix extended their Simian Army with further tools where each is at its core an extension to the Chaos Monkey. Next to Chaos Monkey, Netflix also invented the Failure Injection Testing (FIT) platform. FIT is a platform to inject failures in a more controllable way other than the Chaos Monkey. The figure below illustrates the work flow of the FIT platform [Net14].

The FIT service pushes failure configuration data to Netflix' API Gateway Zuul. Selected requests are then decorated with the failure configuration metadata. Afterwards, the requests are proxied to their destination services. Every service that is touched by the infected request, checks the requests failure context. If the failure context matches the corresponding component, the service simulates a failure as defined in the failure configuration data [Net14]. By using these tools in combination with the steps described by Basiri et al. [BBR+16], Netflix is able to mitigate possible outages before they have a serious impact on customer services.

To summarize, chaos engineering aims to execute experiments on a distributed system in order to expose weaknesses and possible failure sources. Early failure detection enables engineering teams to mitigate the impact on the production system by initiating actions to do so. This increases the confidence in the system's resilience. Large distributed systems are complex and chaotic in itself. This is reinforced by the fact that such systems are dynamic and changing constantly. Hence, it is unavoidable that during the development process, failures are unintentionally introduced into the system or emerge due to the interconnection of many services. Instead of waiting until an outage happens, possible outages should be tackled in advance by executing controlled experiments and by observing the behavior of the system and thus gain new knowledge about the system. Tools such as Chaos Monkey and Chaos Toolkit, make the execution of chaos experiments easier. On the basis of newly gained knowledge about the system, engineers have the possibility to take actions to remove possible weaknesses that otherwise would have a harmful impact on the system.

This work will focus on the execution of a chaos experiment applied to a case study application. The case study application is described in detail in Section 1.3. Chaos experiments have single components, services or even entire subsystems as targets. Attack areas can be separated into the application level, container level and infrastructure level. For example, making an HTTP call fail for a specific service or slowing down a data base call for a specific table is considered to be an application level attack [Greb].

Application level attacks solely target the application and its dependencies. Container attacks are targeted towards containers that contain an application. For example, shutting down a running Docker container is considered to be a Container level attack. Chaos Monkey is a good example of infrastructure level attacks. The Chaos Monkey tool shuts off entire Amazon Web Service regions which host the application.

Within the scope of this work, the chaos experiment will be executed on three-different levels, thus targeting the application-, container- and infrastructure level. The technologies used within each level are explained in Section 1.1.7.

Running a chaos experiment in production is required in order to provide meaningful insight of the application's behavior. Though this is out of scope of this thesis. Since the case study application is a proof-of-concept implementation, the experiment is conducted within a test environment. The data to define the steady state are obtained by running load tests against the case study application in order to simulate production data. Each step discussed in this Section, other than the automation of an experiment, will be performed as a means to make the experiment as efficient as possible.

2.7. State of the Art Tools and Methods

In the following I will analyze different tools and methods that are currently state of the art for the execution of chaos engineering.

2.7.1. Netflix's Simian Army

Netflix maintains a highly complex distributed system based on thousands of microservices that are constantly changing and improved by many engineers. Netflix established a unique way to conquer the chaos within their distributed system in order to expose possible weaknesses and improve their systems resiliency. At Netflix, engineers push new code into production on a daily basis. More than other services, Netflix heavily relies on its service availability. If a customer is not able to watch a video because of a service outage, he or she might not be a customer for long [BBR+16].

As it was discussed in the first section, it is not enough to just implement fault tolerance mechanisms. The system must also be able to withstand disruptions as effectively as possible. Moreover, a changing production environment implies the presence of harmful failure outages that could negatively affect the system. These outages are not foreseeable by a human since the complexity of the system implies unknown behavior. To comply with the chaos inherent to these systems, Netflix invented the Chaos Monkey tool [IT11].

Netflix advances the view that in order to handle hazardous situations one have to be confronted with it frequently. As an example, the same holds for changing a tire. In order to know how to do it, one has to do it frequently [BBR+16]. That's where the Chaos Monkey comes into effect.

Chaos Monkey is an internal service, which randomly selects virtual machines (VMs) instances and terminates them. As in the previous example illustrated, the purpose for this exercise was to drill engineers to design resilient components which could handle outages instead of trying to prevent them. This is achieved by frequently confronting engineers with situations where outages happen. In order to prevent seriously harming the system by these random outages, Chaos Monkey is only active during common working hours so that engineers can respond quickly to any failures [IT11].

Over the years Netflix has recruited more simians to their service in order to improve the resilience and availability of their distributed system. The so called Simian Army comprises of the services shown in Table 1.1.

Tool name	**Description**
Chaos Monkey	Randomly terminates virtual machine instances and containers
Chaos Gorilla	Simulates outages in Amazon availability zones
Latency Monkey	Induces artificial delays in RESTful client-server communications
Doctor Monkey	Provides healthchecks and monitors external signs of health
Janitor Monkey	Removes unused resources inside a cloud environment
Conformity Monkey	Shuts down instances that don't comply with best-practices
Security Monkey	Extension to Conformity Monkey and finds security violations
10-18 Monkey	Detects configuration and run time problems

Table 2.5.: Chaos Tools used by Netflix also known as the Simian Army [IT11]

Summarized, the Simian Army is a set of tools that Netflix uses in order to improve their systems infrastructure resilience as well as availability. Recently, other platforms and companies have adopted the concept of Netflix's Chaos Monkey to improve the resilience and other dependability attributes of their systems.

Conspicuous, is the mechanism of randomness that Netflix uses to select VMs or Amazon Web Service regions. This refers to the nature of failure appearances in real world scenarios. The failure scenarios Netflix and other engineers try to tackle during a chaos experiment may also appear in a production environment where failures could have a serious impact on the users. Moreover, such events are not foreseeable when and where they happen. By adding random selection such as the Chaos Monkey tool does, Netflix is able to almost perfectly simulate those events by not knowing when and which

components are going to be shut down. As it was illustrated in the subsection Chaos Engineering, the precise simulation of real world events is required by chaos engineering in order to get meaningful results to improve resiliency.

However, the act of randomly selecting components or VM instances seems counter intuitive to the definition of chaos engineering given by Medina [Med18]. Therefore, chaos engineering is driven by executing thoughtful and planned experiments. Nonetheless, Netflix's Chaos Monkey approach of performing chaos engineering is indeed an effective approach but one must not adapt to a random failure injection in order to do effective chaos engineering.

2.7.2. Failure-as-a-Service - Gremlin

Gremlin is a tool build to make chaos engineering easier to apply on systems. It provides two fault injection levels, namely: infrastructure-level and application-level. The first level of faults focuses to reproduce issues with machines the application runs on such as CPU being consumed, disk filling up and network hosts becoming unreachable or slow. The second level concentrates to simulate issues with the application and its dependencies such as making an HTTP call fail for a specific user or make a database call slow for a specific table. Moreover, Gremlin provides a complete application programming interface (API) to interact with, making it unnecessary to use a Web UI which is also provided by Gremlin [Greb].

Gremlin can be run on different platforms such as Azure, Amazon Web Services (AWS) or Kubernetes and so on. Gremlin divides its attacks in different types of "gremlins" such as network gremlins, state gremlins and resource gremlins each attacking different parts of the system. Maintainability is preserved by allowing to define an attack schedule where the attacks can be set to specific times of a week or a day. The possibility to integrate Gremlin as a slackbot helps to have a better overview of when attacks are executed and what the actual results are [Greb].

In contrast to Netflix's Simian Army, Gremlin takes on the roles of multiple "simians". Whereas, Netflix provides different "simians" for different attack points, Gremlin merges multiple functionalities in one tool, thus reducing setup and integration effort. Gremlin is open-source and easy to use due to a Web UI that enables complete configuration of single gremlin attacks. Different to Netflix's services, Gremlin gives a better intuition of how chaos engineering should be done. Therefore, Gremlin provides a control of attack points, thus supporting the aspect of executing thoughtful and controlled chaos experiments rather than rely on complete randomness such as Netflix's Chaos Monkey tool. This inevitably shows, that current tools provided for chaos engineering have different levels of granularity such as Netflix's Chaos Monkey and Gremlin. Different

aspects of a system may have another level of granularity which makes some tools more useful than others. This means that the application of more than one chaos engineering tool may be beneficial according to the system under test.

2.7.3. Chaos testing for Docker with Pumba

Pumba is a chaos testing and network emulation tool for Docker created by Alexei Ledenev and is also available on GitHub. It's name refers to a character from Disney's animated film The Lion King and has its origin in the Swahili dialect. Pumba is able to disrupt running Docker environments by injecting different failures. Among other things Pumba's skill set contains commands for killing, stopping, removing or pausing Docker containers. Additionally, it is able to serve as a network emulation with the possibility to simulate different network failures such as delay, packet loss, bandwidth rate limits and so on [Led17].

Interestingly, one of Pumba's features is the possibility to either pass a list of containers that should be affected or tell Pumba to randomly select a number of containers. If none of the previous is provided, the tool will try to disrupt all running containers. To get a better control over the chaos, it is possible to schedule time intervals and duration parameters. As for the portability of Pumba, the tool is available as a single binary file for Linux, MacOS and Windows or it can be run inside a Docker container [Led17].

Pumba adds another level of granularity namely on a container level. Whereas the Chaos Monkey merely focus on the task to shut down AWS instances, yet Pumba solely focuses on the chaos testing of the Docker environment. This, in combination with Gremlin for example, adds a high level of flexibility to an chaos experiment by the possibility to disrupt each level of abstraction of a system namely on the application-level, infrastructure-level and container-level. In addition, the ability of both tools to control time intervals and attack points allows a better focus of attacks on selective parts of a system such as combining container- as well as application-level disruption. However, Gremlin also provides the ability to run chaos experiments on docker containers. But using two different tools for two different concerns may improve the overall maintainability of the tool chain. Therefore, it is possible to use Gremlin or Chaos Monkey on the application- or infrastructure-level and Pumba as a chaos testing tool for a Docker environment.

2.7.4. Docker Chaos Monkey

Docker Chaos Monkey is an adoption of the Netflix Chaos Monkey provided via a shell script. Docker Chaos Monkey, if executed, has the purpose of killing of Docker swarm services but only those that are specifically labeled as such.

The purpose of the Docker Chaos Monkey is to kill or shut down docker swarm services. In order to identify services for an attack, the tool provides a labeling mechanism where services can explicitly be labeled as disposable. Therefore, the Chaos Monkey for Docker does not randomly kill docker swarm services but merely those who are labeled as candidates for an attack [Pat].

Docker Chaos Monkey is a simple alternative to use but provides much less functionality than e.g., Pumba or Gremlin. However, it is easily configurable over the Docker CLI where merely Docker swarm services have to be flagged with a unique command.

2.7.5. ChaosToolkit

Chaos toolkit is a popular open-source and easy to use chaos experimenting tool. It has been developed to provide a free-to-use possibility to easily run chaos experiments on different levels within a system. Experiments are stated within a JSON or YAML configuration files, where each step is executed consecutively. As monitoring is a very important aspect of executing chaos engineering, Chaos toolkit provides meaningful information after each experiment in order to perform better post experiment analysis to get the maximum value out of each experiment run. Furthermore, Chaos toolkit comes with already built-in extensions for different levels of application such as platform/infrastructure-level injections, application-level injections and monitoring. Extensions for Kubernetes, Azure, Google Cloud, Spring, Gremlin and integration with Slack are also available. Lastly, Chaos toolkit has Netflix's Chaos Monkey as well as Chaos Monkey for Spring Boot already built-in and ready to use but if necessary it can also be disabled [Chaa].

If the provided functionality is not enough to run chaos experiments for a system, Chaos toolkit provides an open API to extend its functionality in order to make it suitable for any application [Chaa]. Chaos toolkit can therefore be considered a powerful and easy to use tool which already comes with a lot of functionality built-in. In contrast to the previously mentioned tools, Chaos toolkit merges functionality from different tools and thus providing them all in one place.

2.7.6. Chaos Monkey for Spring Boot

Chaos Monkey for Spring Boot is yet, as the name implies, another adaption of Netflix's Chaos Monkey. Netflix has been pushing chaos engineering in the past years, at first by their release of the Chaos Monkey tool. Chaos Monkey for Spring Boot is a small library that aims to attack running Spring Boot applications. The library can easily be integrated in a POM file. The Monkey does not start attacking the application upon initialization. Since the latest release it is possible to enable or likewise disable the Monkey dynamically during runtime [Wil18].

The Chaos Monkey will scan the application and look for specific annotations, namely:

- @Controller
- @RestController
- @Service
- @Component
- @Repository

Each service is connected with a watcher that looks for the respective annotation. Currently, the Chaos Monkey provides three attacks: Latency Assault, Exception Assault and KillApp Assault [Wil18]. The level of granularity (which annotation to scan for and which assault to launch) can be entirely configured within the Chaos Monkey properties. Much like Pumba and Kube Monkey, Chaos Monkey for Spring Boot focuses merely on the assault of application-level parts of a system. Additionally, in combination with Pumba and Kube Monkey, this set of tools provides functionality to cover all levels of a system to perform chaos engineering. This results in a broader toolchain but keeps concerns to each tool. Nonetheless, as Chaos toolkit comprises Chaos Monkey for Spring Boot, it might be worth considering to use Chaos toolkit since it keeps the toolchain smaller and keeps functionality in one tool which makes maintainability and monitoring easier.

2.7.7. Kube Monkey

Kube Monkey is an adaption of Netflix's Chaos Monkey especially for Kubernetes clusters. Kube Monkey randomly shuts down a pod with a cluster much as Chaos Monkey shuts down random AWS instances. If there are pods which should not be considered by Kube Monkey, because the blast radius might be to high, a blacklist of pods can be created which describes the pods inside a cluster that should not be attacked by Kube Monkey. Kube Monkey, as well as Pumba, can be scheduled and is by default configured to be

running early in the morning each day. Additionally, Kube Monkey builds a schedule of deployments that will be affect during the week but is not providing information about when those deployments will be affected. Moreover, the number of pods to be killed can be configured in various ways such as telling Kube Monkey to only kill one pod (by default) or killing a certain percentage of running pods inside a cluster. Highly valuable is the possibility to set a value for the Mean Time Between Failures (MTBF) inside Kube Monkey. For example if set to "3", Kube Monkey will shut down a pod every three days. This gives a better and fine-granular control over timing a chaos experiment and provides a good solution for automation chaos experiments on a Kubernetes cluster [Sob].

2.7.8. Chaos Engineering Game Days

Chaos Engineering Game Days are a common strategy for introducing and applying chaos engineering in a company. A game day typically consists of a preparation or planning phase and an execution phase. The preparation phase is needed in order to prepare the setting and information needed for the execution phase. Common information include [Grec]:

- Target system, subsystem or component
- Time and place and
- Goal for the Game Day

A game day is a workshop that takes approximately two to four hours. The goal is to gather ideas for possible chaos experiments and running them live, to observe the outcome. A thoughtful preparation is needed because it is a hands on meeting were a lot of people get together. Defining the target in the planning phase, implicitly gives information about the people that should attend the game day. These people include engineers, managers or any other stakeholders that have a good understanding of the target system. The goal of a game day may differ from time to time. A possible goal could be to state relevant test cases for a chaos experiment or investigating if a new subsystem has all the right monitoring tools integrated that are necessary for the deployment to production. During the game day, it is considered as a best practice to model the current state of the system under test. This way, everyone attending the meet up is up to date with the current state of the system and also it makes possible failure points visible [Grec].

After the identification of possible failure points, it is time to think of possible failure scenarios. It is important to consider the blast radius during the game day. This implies that it is necessary to have a abort plan if a failure has a unexpectedly large blast radius.

Additionally, to run live experiments efficiently during a game day, the right tools have to be integrated such as monitoring, metrics and so on.

Establishing metrics for a future game day and chaos experiments can also be stated as a goal for a game day. The metrics are necessary in order to know what to look out for during the execution phase. Otherwise it is not possible to observe if a specific test case has a meaningful impact or if a test case causes the expected effect on the system [Grec].

To summarize a game day is a popular method to start out with chaos engineering. However, planning a game day takes time and has to be done faithfully. The core aspect of a game day are the people attending the game day. If non of the people attending has deeper knowledge about the system under test, it becomes dangerous and difficult to draft the current architecture of the system and create test cases. This might result in unknown outcomes that have a large blast radius. Thus, it is important to assure that at least some of the people from a team responsible for the system under test, are present.

Chapter 3

Related Work

The term chaos engineering has been characterized by Netflix's way of improving their cloud infrastructure's resilience [BBR+16]. Intentionally inducing chaos into the production environment, has now become a part of Netflix's development culture. Due to the novelty of chaos engineering and its unusual nature in tackling resiliency, there is an absence of scientific resources. Moreover, currently available resources are very generic in terms of how chaos engineering should be applied to any kind of distributed system or executed within a project's life cycle.

At the time of writing, scientific research resources in terms of case studies, scientific papers, and such are limited. Therefore, this chapter will also include non scientific research resources which are referred to as grey literature. Grey literature comprises of material that is produced by academics, business, and industries either in a printed format or in an electronic format but is not controlled by commercial publishing interests [FF98].

This chapter is structured as follows. Section 3.1 explains the search strategy that has been used to sight literature and other resources. Section 3.2 introduces the most relevant literature to chaos engineering. Section 3.3 discusses different professional conference contributions. Section 3.4 provides information of other related case studies. Section 3.5 summarizes the findings of this chapter.

3.1. Search Strategy

Initiating research at resources provided by Netflix, a couple of keywords stood out that seemed to be promising for further investigation. Among others these keywords have been used during further research:

- chaos engineering
- resilience engineering
- chaos engineering netflix
- incident management netflix
- site reliability engineering
- chaos engineering case study/field study
- conference papers chaos engineering

The above itemized keywords were the most dominant terms mentioned in grey literature regards chaos engineering. On top of using different keywords within different search engines, different mentions by contributors in the field of chaos engineering have been followed, e.g., case studies mentioned by companies, specific conference contributions, and so on. Most contributions in the field of chaos engineering converge to professional conference talks given by well known people in the field of chaos engineering. Netflix as a main contributor to the chaos engineering culture yielded high value for providing a high-level picture to chaos engineering, but provided little to no value for deeper understanding of chaos engineering.

3.2. Chaos Engineering by Basiri et al.

Probably the most referenced and iconic work related to this thesis is Chaos Engineering by Basiri et al. [BBR+16]. In their book, the authors focus on explaining the principles of chaos engineering as well as its benefits, drawbacks, and how to apply chaos engineering on a distributed system. In their research, Basiri et al. explain different terms related to chaos engineering such as steady state, hypothesis, failure scenarios, and blast radius. Additionally, the work gives detailed information about case examples of incidents that have happened at Netflix and other companies, that probably could have been avoided if chaos engineering was applied.

Furthermore, Basiri et al. identified five key steps to successful chaos engineering. These steps include [BBR+16]:

- Form steady state hypothesis
- Real-world event variation
- Run experiments in production environment
- Experiment automation

- Mitigation of blast radius

The different steps are thoroughly explained by the authors and how they contribute to the motivation behind chaos engineering. Nonetheless, the content of the work by Basiri et al. is as generic as its title. Though, it provides a meaningful insight into chaos engineering processes, its motivation, and benefits, as well as useful insights into Netflix's experiences with chaos engineering. Therefore, the work by Basiri et al. is a great summary of chaos engineering, what it comprises of, when to apply it, and what its benefits are. However, it is clearly missing a practical reference, e. g. forming an example hypothesis, or describing a steady state with real-world metrics. Despite the thorough explanations, it is not clear on how to practically apply or execute a chaos engineering experiment on a distributed system.

3.3. Professional Conference Contributions related to Chaos Engineering

In the past few years, the number of conference contributions in terms of keynotes has increased in the chaos engineering community. The Chaos Conference [Grea] is an event solely dedicated to contribute new insights, techniques, and methods to improve chaos engineering as a practice.

The discussed topics range from explaining the principles of chaos engineering to changing the mindset of engineers to faciliate these principles. Probably, the most popular speakers are Nora Jones and Kolton Andrus. At the time of writing, Nora Jones is currently the Head of Chaos Engineering and Human Factors at Slack, whereas formerly being a Senior Software Engineer in the Resilience (Chaos) Team at Netflix. Nora Jones' contributions primarily consist of explaining chaos engineering as a practice, its benefits, how to design microservices for a chaos engineering approach, and how chaos engineering differs from a traditional testing approach. She was also involved in the writing and publishing of Chaos Engineering by Basiri et al [BBR+16]. Most of her contributions provide a distinction of traditional testing approaches to chaos engineering. This is followed by an explanation of the principles of chaos as well as an introduction to different types of chaos failures within Netflix. Nora Jones' contributions are a great possibility to get a better overview of chaos engineering and its applications as well as its benefits. Though, her contributions lack the possibility to get a better understanding of how to design a chaos experiment for a specific application.

Another, yet increasing in popularity, well-known person is Kolton Andrus. Kolton Andrus, formerly Senior Software Engineer at Netflix's Resilience (Chaos) Team, and now CEO of the Gremlin platform [Greb], is known for its contributions towards a

more controllable approach to chaos engineering. Chaos engineering is known for its randomized approach to cause chaos. Andrus' approach with Gremlin is more controllable by providing precise definitions of attacks to specific levels of an application. In general, Gremlin differs between application level attacks and infrastructure level attacks [Greb]. Nonetheless, there is still a randomization within the process in order to avoid bias. But by providing a detailed guide to certain chaos attacks, Andrus' allows a more in depth view into the chaos experiment process.

Another approach to non-randomized chaos engineering is given by van Hoorn et al. [HADP18]. The ORCAS project aims to reduce the overall number of chaos engineering experiments that are necessary to find vulnerabilities. The key idea is to leverage architectural knowledge of the system that is given to a decision engine, which then runs a dedicated number of appropriate chaos engineering experiments. The chaos engineering experiments are selected based on the given architectural information, e.g., number of microservices and existing resilience patterns and their configuration. Consequently, the ORCAS project is yet another contribution, which does not leverage randomization as a part of chaos engineering experiments, such as the Simian Army [IT11].

There are many more contributors who continuously pushing the field of chaos engineering. Lorin Hochstein, who also participated in the writing of Chaos Engineering by Basiri et al., provides a cumulated list of important contributors to the field of resilience engineering [Hoc].

Altogether, the resources in terms of gray literature is strictly saying, quite generic or very specific. For example, most talks given by several known chaos or resilience engineers are about the necessity of chaos engineering, its principles, and how its benefits. Other talks given explain very specific topics such as designing microservices for failure, changing the mindset of engineers for chaos engineering, pitfalls to avoid and so on. A great example of this, is a new approach to optimize the process of finding failures during the chaos experiment execution, established by Kolton Andrus and Peter Alvaro [ARH15].

Lineage Driven Fault Injection (LDFI) describes a backwards reasoning mechanism which tries to follow different paths down to the root of an outcome in order to find failures, that could have prevented the outcome from occurring [ARH15]. The novelty of this approach is non debatable but has little to no purpose for an organization or group of people who want to adopt chaos engineering in the first place. Nonetheless, this approach seems to be promising for advanced chaos engineering applications.

Summarized, contributions in the field of chaos engineering referred to as gray literature is a good place to start but lacks a detailed guide to designing chaos experiments for specific system types.

3.4. Case Studies

Profound case studies in the field of chaos engineering are scarce or non existent. At the beginning of chaos engineering, most companies shied away from applying chaos engineering due to the lack of knowledge of the art and missing guidelines. This was also mentioned during a talk by Nora Jones given at the AWS re:Invent in 2017 [Jon]. Yet, a lot of companies started to adopt chaos engineering in the past years such as Facebook, Amazon, Microsoft, Dropbox, and several others of different industry branches. A mind map of companies who adopted chaos engineering as well as the tools they used, can be extracted from the appendix.

Despite the increasing number of companies adopting the principles of chaos, there is still a lack of profound case studies. Nonetheless, a few have been conducted and are even available to the public.

3.4.1. Case Study at IBM: Systematic Resilience Testing of Microservices

The first case study has been conducted at IBM and published in 2016. Heorhiadi et al. [HRJ+16] stated that most modern internet applications are implemented as loosely coupled microservice architectures. Usually, these loosely decoupled services are updated and deployed multiple times a day. Therefore, a new approach to test the resilience of such applications is required.

Heorhiadi et al. continued by summarising recent incidents of popular and highly available internet services such as Spotify. They indicated that these incidents refer to a missing or a lack of failure-recovery logic [HRJ+16]. Moreover, they state that unit and integration tests do not have the possibility to avoid those outages.

Other than Netflix, who state that a randomized approach is more valuable due to the non biased approach of failing components [BBR+16], Heorhiadi et al. argues that a systematic testing approach is more valuable. It is explained, that a systematic approach to chaos engineering yields a better opportunity for developers to re-run specific failure cases and receive immediate feedback [HRJ+16]. It is argued that the feedback loop is more valuable than a randomized approach, since developers are able to quickly locate and fix certain failure cases [HRJ+16].

The idea for a systematic approach to chaos engineering has also been proposed by Kolton Andrus, who created Gremlin. This case study shows, that chaos engineering is now moving on a scale between two endpoints. One being a randomized approach, whereas the other one being a more systematic approach. As outlined in Chapter 1,

more tools emerge on the market that allow a more fine-grained selection of which components to attack such as Pumba for Docker [Led17].

However, the case study focuses on providing an explanation on how to use Gremlin for fault injection attacks on a microservice based system. Considering the different steps involved in chaos engineering, it is reasonable to say that this case study merely covers the step of executing a chaos experiment. The steps before and after are neglected. Hence, it is not stated how the steady state is described nor how a hypothesis is formed. Heorhiadi et al. provide a very focused and specific view on executing a chaos experiment by showcasing, how Gremlin can be used to systematically test the resilience of a microservice based architecture.

3.4.2. Case Study at Oracle: Lessons Learnt of approaching Chaos and Resilience in a Mid-Sized Organization

The paper by Nazareno [Fei18] is not necessarily considered to be a profound case study but rather close to an experience report on applying chaos engineering. The key points of the paper outlines a couple of pitfalls according to the organizational appliance of chaos engineering and an approach to evaluate different chaos engineering tools.

Important to note is that, in his paper Nazareno explains that leveraging the architecture of an application yields a lot of potential when applying chaos engineering. Neglecting the core statement of chaos engineering which states, experiments must be run in production [Fei18]. Nazareno took another approach by slowly moving from an architectural proof-of-concept level to the testing stage, leaving the production state chaos experiment as a final destination.

Nazareno states in his paper, that a lot of resources related to chaos engineering are available but none of them take an approach to leveraging the architecture level to design the experiments. However, the paper does not provide insight on how chaos experiments are designed and applied in detail but how tool evaluation is taken care of at Oracle. Additionally, organizational changes are considered and explained in detail but the analysis of these changes is out of scope of this thesis.

3.4.3. Case Study at HomeAway

Another case study has been conducted at the company HomeAway [Mil18]. HomeAway is an online booking platform for people who travel a lot around the world and allows them to quickly find places to stay in their favourite countries. The following information were available as an interview posted on Medium by Russ Miles of ChaosIQ.

During the interview the contributors Eduardo Solis and David Sumera explain how applying chaos engineering to their business and extending the ChaosToolKit for their needs, has evolved their perception around testing resilience [Mil18]. To summarize, the contributors integrated the ChaosToolKit, and therefore an approach to chaos engineering, into their Continuous Integration (CI) and Continuous Delivery (CD) pipeline. Additionally, they extended the tool by functionalities such as monitoring and dynamic injection of failures into their system [Mil18].

Unfortunately, the contributors did not provide information about how chaos engineering is applied to their processes or how they design chaos experiments or even how they select appropriate failures to inject by assistance of the chaos toolkit.

3.5. Summary of Findings

To summarize, resources available that provide a detailed overview of every step in the chaos engineering process are scarce or none existent. The above described case studies are in general not profound case studies but yield insights as experience reports with chaos engineering. The grey literature comprises of a lot of information about chaos engineering, its benefits, what tools to use and how to use them, designing microservices for failure, and what to focus on during chaos engineering. Though, most contributions are quite generic in terms of the field of appliance such as designing components for failure or experience reports on chaos engineering. On the other side, contributions become too specific by providing approaches to further optimize the finding of potential outages such as by Alvaro et al. [ARH15].

To fill the gap, this thesis shall provide a better understanding of the entire chaos engineering process by thoughtful planning and execution of a chaos engineering experiment on a large scale distributed system.

Chapter 4

Case Study Design

The usage of case study research has been increasing in multiple research areas over the past years [HKD14]. Case studies in particular have been proven to be valuable to investigate phenomna of software systems in their natural context. Conducting a case study comprises multiple steps and techniques that have been explained in Section 2.3.2. This case study design is based on the work of Wohlin [WRH+12] and Runeson et al. [RH09].

This chapter is structured as follows. Section 4.1 describes the case study objective. Section 4.2 explains the software application under observation referred to as the investigated system. Section 4.3 provides an overview of the data that will be collected throughout this work. Section 4.4 provides the methods and techniques used for data collection. Lastly, Section 4.5 describes how the data defined in Section 4.3 will be analyzed.

4.1. Objective

In this case study the objective is descriptive and can be stated as follows. The objective is to analyze the planning and design of chaos engineering experiments, for the purpose of providing a systematic approach to chaos engineering, respective to an industrial case study application, from the point of view of a chaos engineer in the context of a bachelor's thesis.

The principles of chaos and various elements of chaos experiments have been elaborated in Section 2.6. The expected result of this case study is a profound understanding for conducting chaos engineering experiments on a distributed system. The process is supported by application of hazard analysis methods in order to create meaningful experiments.

4.2. The Case

The investigated system implements a service which is responsible for efficient handling of automotive configuration data. In its domain context, different systems obtain configuration data for further analysis and processing, e.g., Manufacturing Execution Systems (MES), configuration tools provided as web interfaces, and such like. Given the fact, that the goal of these systems is to sell cars to customers, it is reasonable to imply that the provisioning of automotive configuration data has to be efficient in terms of performance, availability, and resilience. The investigated system is provided by a third-party company that is currently responsible for the system's development and maintenance. Currently the system is stated to be a prototype of an already existing application which does not meet the requirements of the stakeholders. A set of technologies has been selected by the stakeholders in addition to a deployment platform the application has to be deployed to in the future. By the time of writing this thesis the application has not been deployed to production or a production-like environment. Thus there is no field data available with respective to the system's performance, available or resilience. However rough statements given by the stakeholders are available with respect to expected response times and user behavior which will be used as a threshold for this work.

4.3. Preparation of Data Collection

In the following, multiple data sets will be identified which are necessary to further analyze the investigated system. The data sets are divided into data directly related to the investigated system and data not directly related to the investigated system. Data directly related to the investigated system comprises information such as architecture description, and structural as well as behavioral modelling of the system. Data not directly related to the investigated system comprises information that is collected from the stakeholders, e.g., name and profession of participants. The different data sets are summarized in Table 4.1 and Table 4.2.

The data sets outlined in Table 4.1 are further divided into the groups architecture and design , steady state, and resilience assessment. Architecture and design is separated into two sub groups technical data and non-technical data. Technical data comprises information that describes the application by leveraging technical details such as architecture descriptions and views, sequence diagrams, used technologies, and so on. The architecture description will result in a high-level architecture diagram that explains the structure and composition of the investigated system. Behavior and structure serves the purpose to get a better understanding of the interactions between components and

Architecture and Design	
Technical data	Non-technical data
• Architecture description • Application behavior and structure • Technology stack • Motivation to migrate architecture	• Non-functional requirements • Functional requirements • Design decisions • Decomposition into services
Steady State	
• Response Times	
Hazard Analysis	
• Fault Tree Analysis • Computational Hazards and Operability • Failure Modes and Effects Analysis	

Table 4.1.: Data sets directly related to the investigated system

how users interact with the system. Explaining the technologies is crucial for the chaos experiment with respect to selecting the appropriate plugins and injection levels. The motivation to migrate to a different architecture explains the system's domain context and requirements.

Non-technical data references information that may lead to the creation of technical data such as functional and quality requirements, design decisions with respect to selecting appropriate technologies and component design. Further information about design decisions and microservice decomposition lead to a better understanding of the system's design.

Steady state data describes measured response times of the application as a technical metric that is used for observing the system's behavior during a chaos experiment.

Hazard analysis data comprises of the results collected from applying hazard analysis methods. Chapter 6 applies three hazard analysis method to identify potential hazards and failures.

Demographical data	
Participant-related data	Project-related data
• Name • Age • Profession • Work experience • Period of employment	• Current responsibilities in project • Retrospective changes • Benefits of current implementation • Drawbacks of current implementation

Table 4.2.: Data sets not directly related to the investigated system

Data not directly related to the investigated system is summarized in Table 4.2 and comprises demographical data. The demographical data is further divided into two groups namely participant-related data and project-related data. Due to anonymization of subjects, the participant-related data will not be provided in the Appendix. The demographical data is a means to ease the data collection process according to the funnel model [RH09]. During the data collection process it is possible that more data is raised than described in Table 4.1 and Table 4.2.

4.4. Data Collection Methodology

In order to know from which sources the data sets of Section 4.3 can be collected a process to identify the information sources needs to be explained. In this thesis, the investigated case is a given. The people who develop the case, are the primary information source. Nonetheless, there might exist other potential information sources.

4.4.1. Identification of Information Sources

From a requirements engineering perspective, the procedure to identify different information sources is similar to the identification of requirements sources. Pohl [Poh10] suggests an iterative approach for requirement elicitation which, can also be considered to raise additional, potential information sources for this case study. The people implementing the investigated system, will serve as the primary information sources. The iterative process to raise further information sources is described as a three steps process [Poh10]:

1. Ask already identified stakeholders for additional information sources.
2. Investigated identified archival data for further, potential information sources.

3. Analysis of existing systems for additional, potential information sources by examining interactions with other systems or users.

However with respect to the investigated system the elicitation of information is limited according to non-disclosure of customer-related information. Therefore, the information sources stated in step two might not be accessible. The third step includes information sources that are not available since any external services that interact with the investigated case are not maintained by the system's developers.

This leads to the conclusion that the stakeholders developing the investigated case and existing archival data are the only available information sources to raise the data according to Table 4.1 and Table 4.2.

4.4.2. Qualitative Data Collection Methods

Due to the limited number of previously identified information sources few qualitative methods to collect the necessary data are worth considering. In the scope of qualitative information elicitation given that merely stakeholders serve as a meaningful information source it is possible to consider interviews, focus groups, questionnaires and surveys as a means to collect the required information.

The benefits and drawbacks of each method has been described in detail in Section 2.3.2. In conclusion conducting interviews seems to be promising for gathering information from people about a case where no other information is available. This is supported by the fact, that the required information is not available anywhere else than in the minds of people who are interested in the investigated case [RH09]. Hence, a semi-structured interview approach is used to raise the information from the stakeholders.

In order to identify and create failure models that can be injected into the investigated case, three hazard analysis methods known from the domain of safety assessment will be applied. The methods suggested are (1) Failure Modes and Effects Analysis, (2) Fault Tree Analysis and (3) Computational Hazard and Operability.

- Failure Modes and Effects Analysis
- Fault Tree Analysis
- Computational Hazard and Operability

These methods are described in Section 2.2 whereas a thorough application of each method is done in Chapter 6.

4.4.3. Quantitative Data Collection Methods

In the scope of this case study quantitative analysis will be restricted to evaluate the results of the chaos engineering experiments. This is based on the fact that there exists no previous information that justifies application of descriptive statistics.

As it is mentioned in Section 4.3, there is no field data available that could serve as a potential description of a meaningful steady state. Therefore Apache Jmeter is going to be used to provide threshold measurements of expected user behavior.

In order to simulate user behavior a load test is run against the investigated system in assistance of Apache JMeter. The expected values for the user behavior are based on the findings of Section 2.1 and will be used as a threshold to define a steady-state and create appropriate environmental conditions for the chaos experiments.

4.5. Data Evaluation Methodology

The interviews are going to be transcribed and compared to observe any repetition of information. Moreover, the findings are going to be edited, e.g., diagrams drawn by participants are illustrated with software.

The results of each chaos experiment are evaluated by applying hypothesis tests in order to verify if the defined steady state hypothesis' still hold or not. Other calculations in context of descriptive statistics are mostly performed by JMeter, e.g., calculating the average response time.

4.6. Interview Structure and Question Design

This section provides an explanation of the interview structure and the participant selection process. The interview structure includes length of interviews, environment selection, tooling, as well as question design.

As a means to raise the information according to Table 4.1 and Table 4.2, the interviews are going to be semi-structured. The participants will be invited to one-on-one interview sessions where each interview takes place at a specific meeting room provided by Novatec Consulting GmbH in Echterdingen. Occasionally one interview will be performed remotely in consequence of the participants location. The duration of each interview was estimated to be approximately one hour and thirty minutes.

Two out of three interviews will be recorded with the app EasyVoiceRecorder for iPhone 8. The third interview will be recorded with the built-in recording tool of Skype. The consent form includes a permission to take notes and and a permission to record the interview.

4.6.1. Selection of Participants

As defined in Section 4.4.1, a few potential participants are available as potential interview participants. In total there are seven potential participants, where each of them is currently involved or has been involved in the development of the investigated system. According to Wohlin et al. [WRH+12] and Chapter 2, participants are going to be interviewed until saturation occurs. The saturation serves as good evidence of enough participants being interviewed.

4.6.2. Planning the Interview Questions

As explained in Section 2.3.2, semi-structured interviews do not strictly follow a prepared question catalogue. Nonetheless, a question catalogue will be prepared in order to provide a structure that sufficiently covers all necessary information about the investigated system. The question catalogue might change during an interview, depending on the direction of a conversation or if a specific topic has to be further developed. This includes that some questions might be removed or added during an interview.

The following interview structure and sequence of questions is proposed, where the questions are arranged within topics:

1. Demographical data
2. Project related data
3. Requirements
4. Architecture
    a) Structural View
    b) Behavioural View

The interview question catalogue can be found in Appendix A. The sequence of the topic questions refer to the funnel model explained by Runeson et al. [RH09].

At arrival, the participants are going to be asked kindly to read and sign a consent form which can be extracted from Appendix A. Afterwards, the participants are given an introduction comprising detailed information about the goal and motivation of the interview. The questions of each topic will be asked consecutively whereas the sequence of the topics is the same as in the list above.

After given an introduction, the interview begins with questions according to the demographical data, comprising of profession, working experiences, age, and name of the participants. Afterwards, the focus of the interview narrows down to project related questions comprising motivation, planning, and task distribution of the project. Next, the participants are going to be asked questions about the application's requirements and how the requirements were established. The last topic focuses on the architecture of the application. The architectural questions have been further separated into two topics, (1) structural view and (2) behavioural view. As the topic labels imply, structural view comprises questions about the structure of the application such as the number of components, architectural style, component dependencies, and connections. Behavioral view comprises questions about the behaviour of the application and its component such as communication type, and the general workflow.

At the end, the interviewees receive information about the next steps, which comprises of (1) generating a transcript of the interviews and (2) analysis of the qualitative information by creating a detailed description of the investigated system. The interviewees will receive a copy of the transcript in order to review their statements and perform corrections in order to assure a higher quality of the collected information.

Chapter 5

Cloud Service for Vehicle Configuration Files

This chapter provides a detailed description of the system's business domain, a description of the legacy application, and a detailed explanation of the new prototype system. The explanation of the prototype system includes a description of the system's architecture and its components, a description of implemented patterns, and a modelling of the system's behavior.

Hence, this chapter is structured as follows: Section 5.1 outlines the sources of information as a foundation for this chapter's content. Section 5.2 describes the reason for the development of the investigated system, including the system's requirements. Section 5.3 provides an explanation of the legacy application's architecture. Section 5.4 explains a new architecture for the prototype system including a description of its components, implemented patterns, and modelling of the system's behavior.

5.1. Findings from Interviews

This sections summarizes the findings from the interviews. These findings are the main source of information relevant for this chapter's content. All information referring to the analyzed legacy and prototype system have been raised during the interviews.

As outlined in Section 4.6, a series of semi-structured interviews has been conducted. The objective of the interviews was to get more information about the investigated system with respect to the data listed in Table 4.1 and Table 4.2. In total, three interviews have been conducted. On average, each interview took approximately 1 hour and 17 minutes, whereas 1 hour and thirty minutes have been initially estimated for each interview. In

total, the interviews took 3 hours and fifty-one minutes. Before the beginning of each interview, the participants were asked kindly to read and sign a consent form to ensure data integrity and anonymity. After the successful conduct of each interview, a transcript has been created and sent to the participants in order to review their statements and increase data quality.

To summarize, the interviews provided a detailed view on the domain context of the project as well as its architecture. During the third interview almost every information started to repeat. Hence the interview process has been stated to be saturated and no further interviews were necessary to be conducted [WRH+12]. Diagrams, models and such like that were drawn during the interview, have been edited. The requirements provided have been rephrased as user stories.

5.2. Domain Context and System Scope

In this section the context of the investigated system is clarified by explaining the motivation for developing such a system. The motivation is emphasized by giving an explanation of the domain context as well as describing the system's scope by providing a set of requirements.

There exists a large amount of different car types that are currently available for purchase. To itemize a few, a car can be of type passenger car, truck, bus or electronic. The manufacturing process differs for every car type since each car type consists of different parts with respect to their size, functionality, and individual preferences such as the inclusion of interior gadgets. The composition of different car parts result in a set of car configurations.

Each set of configuration data is used by different systems where each system describes a different business domain. Common systems that may obtain configuration data are Enterprise Resource Systems (ERP), Manufacturing Execution Systems (MES), or configuration tools which enable users to configure a car to their needs. Figure 5.1 shows a high-level view of the described environment. The KBV component illustrates the investigated system which is connected to a database that stores configuration data. Third-party systems comprise ERP, MES, or other systems that may need to obtain configuration data for further processing. The end user component describes administrators or other stakeholders who need to request configuration data from the system for further analysis. Given the above described context, it is reasonable to state, that the configuration data is crucial for the entire manufacturing and sales process.

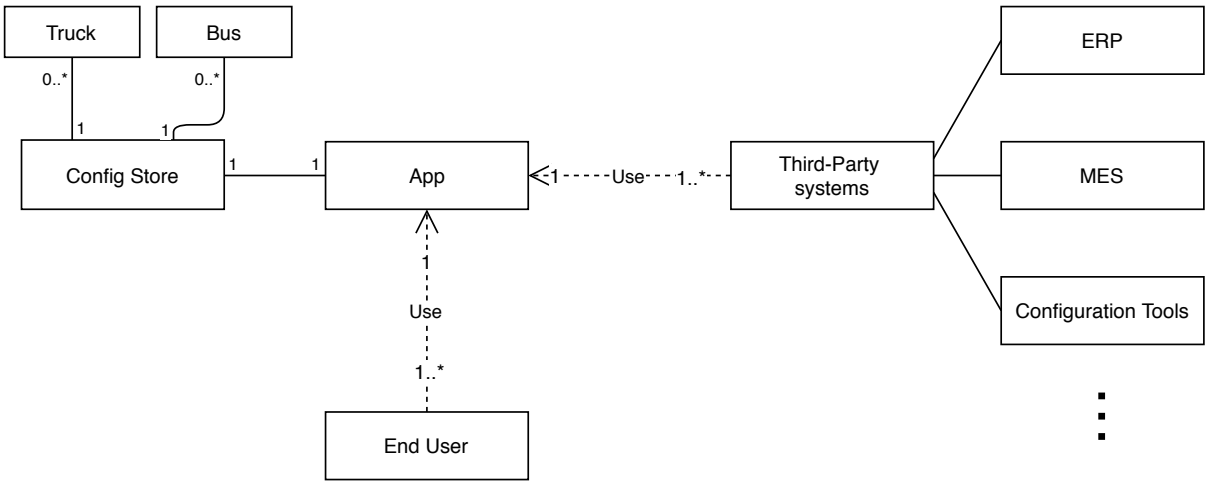

Figure 5.1.: Investigated system embedded in its domain context

The strict entanglement in different processes, requires configuration data to be highly available as well as being handled efficiently. Additionally, systems that rely on this data, must have quick access to it.

To achieve a high availability and an efficient handling of configuration data a set of requirements must be defined. The stakeholders have defined a set of core requirements the prototype has to fulfill. The requirements can be summarized as follows:

Functional requirements

- As a third-party system, I want to retrieve configuration data by accessing a RESTful API that is similar to the API implemented by the legacy application, so that I do not have to change my implementation.
- As an administrator, I want to create new configuration data by accessing a RESTful API, so that I can add new configurations to the database.
- As an administrator, I want to delete unused configuration data by accessing a RESTful API, so that I can clean up the database from unused data.

Technology requirements

- As a stakeholder, I want the prototype to be deployed on Kubernetes, to enable easier and faster deployments, and a better container orchestration.
- As a stakeholder, I want the application level logic implemented by using the Spring Framework in assistance of Spring Boot, so that I can leverage all features of the Spring Framework.

- As a stakeholder, I want the application to be containerized with Docker, so that I do not have to learn a different technology for managing my containers.

Quality requirements

With respect to the quality requirements the developers were not given detailed information but mere statements, e.g., the system's availability has to be at least as high as the legacy application's availability or higher. This also applies to the resilience, scalability, performance. Therefore, phrasing a set of dedicated user stories to define those requirements is not possible.

The functional requirements can be summarized to provide an equal RESTful API to access the configuration data. However, given the fact that a legacy application already exists, it's insufficient to meet the requested quality and technological requirements given above. Hence, the legacy application should be improved with respect to its availability, scalability, performance, and resilience. The technological requirements were given as a means to achieve the quality requirements.

Therefore, the goal is to create a prototype which handles car configuration data efficiently, and provides equal APIs as the legacy application. On top of that, the prototype's availability should be at least as high as the legacy application's availability, and being able to scale efficiently in a cloud environment. Additionally, the performance should be at least as good as the legacy application's performance, and being as resilient as possible.

5.3. Architecture of the Legacy Application

The legacy system in its current state is insufficient to meet the quality and functional requirements described in Section 5.2. To understand the system's structure and domain context, the following Figure 5.2 illustrates all possible use cases.

Note that each use case is also implemented by the prototype system. Hence, the use cases illustrated in Figure 5.2 also reference the use cases for the prototype system. Figure 5.2 shows 3 actors who participate in the use cases. The first actor is the end user who can only participate in the use case to request configuration data. The second actor is a third-party system which can participate in the same use case as the end user. The third actor is the administrator. An administrator can participate in two use cases. The first use case initiates the creation of configuration data, whereas the second one deletes configuration data. In total, these are the only use cases currently implemented in the system for both the legacy and the prototype system.

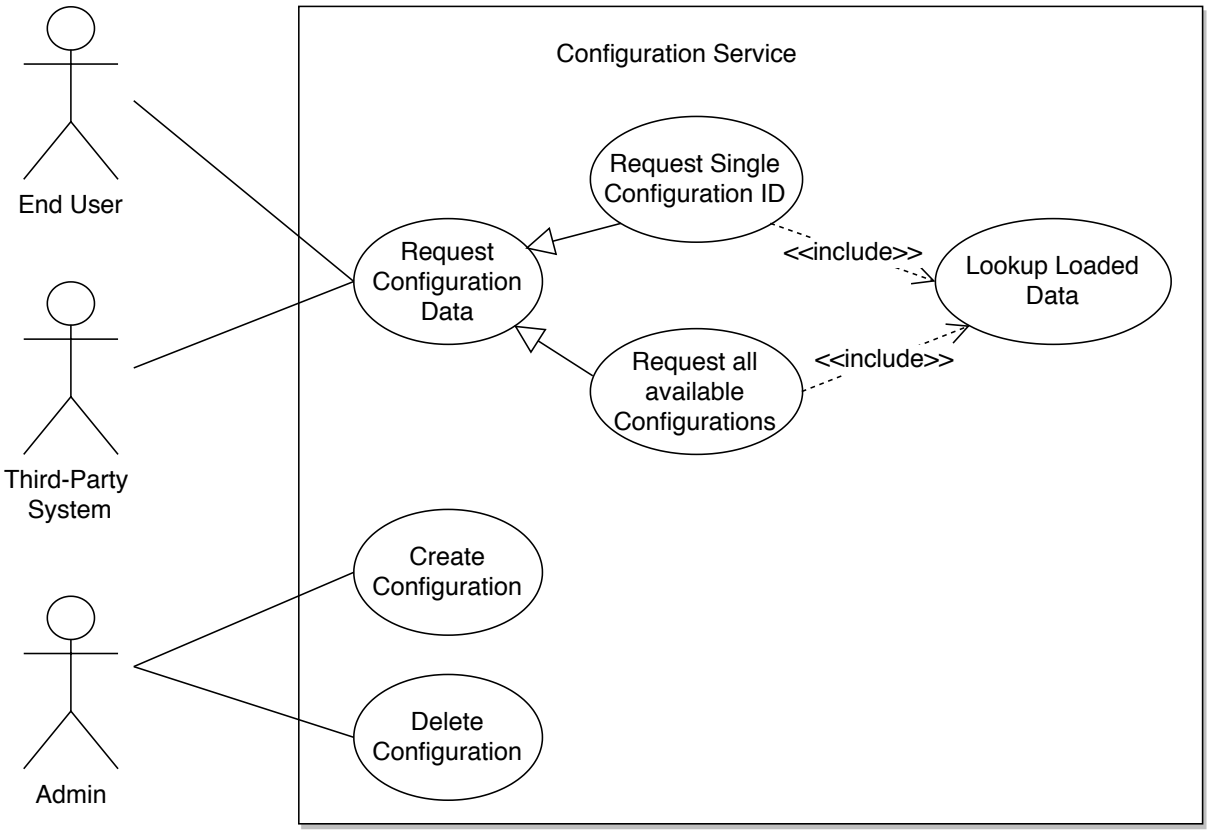

Figure 5.2.: Use Case diagram of the prototype and legacy system

The legacy application itself is a Java Enterprise Edition [Orab] application, deployed on two WebSphere Application Server [IBM] instances, running inside a private data center. Each WebSphere instance hosts the same legacy application. Beyond that, the developers have not been provided with much information about the legacy application. Though, due to the cutting of the services, the monolithic approach allows an assumption of how the system might be structured. Figure 5.3 shows a high level architecture of the legacy application.

The application, being a pure backend application, does not contain any front end components and therefore no graphical user interfaces (GUI). Communication within the application is done by accessing application programming interfaces (APIs). The APIs are implemented based on the architectural style Representational State Transfer (REST). APIs exposed for third-party systems are also based on REST. In total there are 2 external APIs which can be accessed, namely the end user and administrator API. Third-party systems access the same API as the end user. Each API allows exactly two Hypertext Transfer Protocol (HTTP) requests whereas each request can directly be mapped onto the use cases illustrated in Figure 5.2.

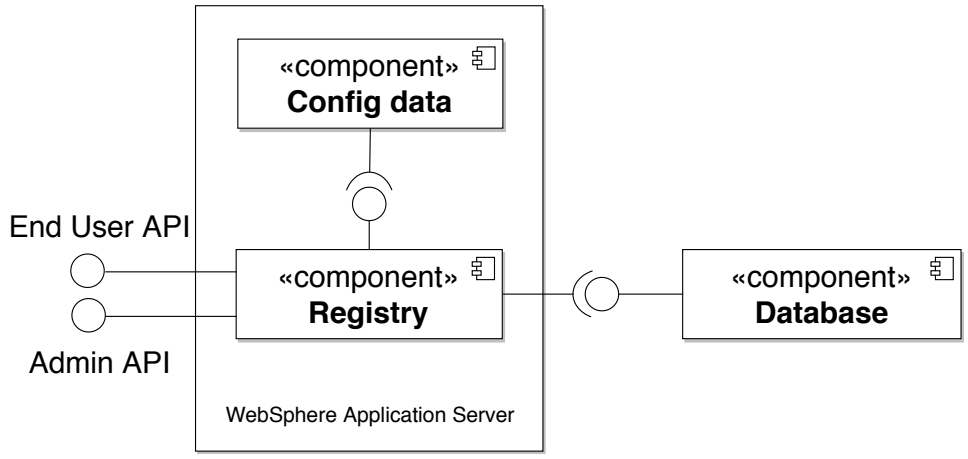

Figure 5.3.: Monolithic implementation of the case study application

HTTP Request	Request Method	Role
getAll()	GET	end user/third-party system
getById()	GET	end user/third-party system
create()	PUT/POST	administrator
delete()	DELETE	administrator

Table 5.1.: HTTP requests of the legacy and prototype system

The configuration data is stored in a database of unknown type. It is assumed that the data is stored in a document-based database such as Cassandra [Thea], MongoDB [Mon], or CouchDB [Theb]. The configuration objects are stored as binary long objects (BLOB). In Java a BLOB Object is the representation of a serialized SQL BLOB value, that stores a BLOB as a column value in a row of a database table [Orac].

The registry is responsible for accepting requests by the administration and the end user API. Additionally, the registry keeps track of the amount of configuration data objects that are currently persisted in-memory.

Table 5.1 shows the HTTP requests that are currently permitted by the given APIs. The end user API accepts two requests of type GET, namely getAll() and getById(). The administrator API accepts two HTTP requests, one of type PUT or POST and the second of type DELETE, namely create() and delete(). All requests are accepted by the registry.

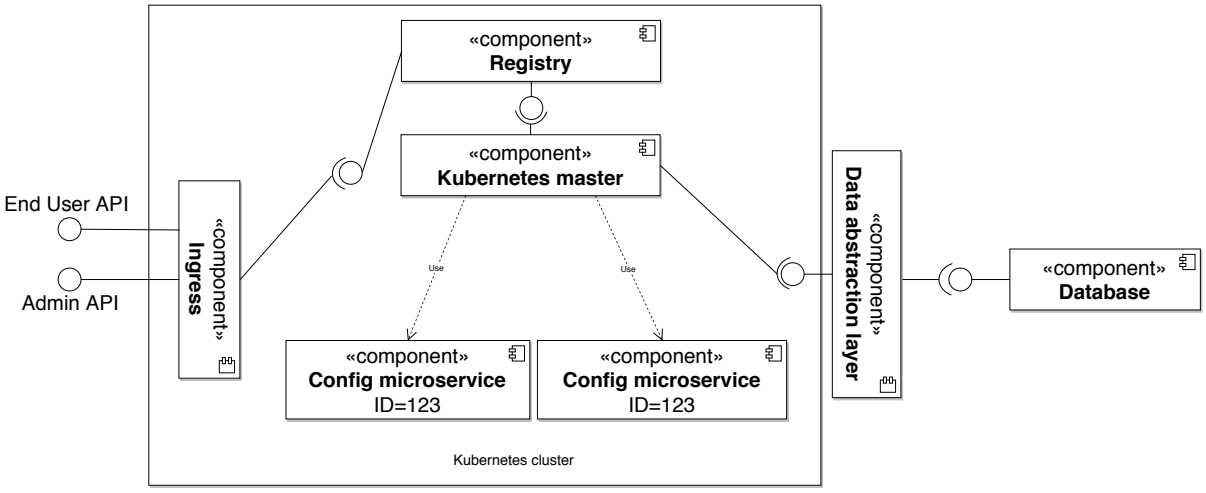

Figure 5.4.: Component diagram of the prototype system

5.4. Prototype of a Microservice based Architecture

This section will describe a microservice based architecture which reflects the prototype system. The architecture discussed in this section will serve as the system under test for chaos experiments.

Figure 5.4 illustrates a high level view of the prototype's system implemented as a microservice based architecture. The functional requirements from Section 5.3 have not changed but the architecture does now sufficiently satisfy the specified technological requirements. In the following the structure of the architecture is explained in more detail as well as the patterns that have been used to implement the prototype.

5.4.1. Structure

Based on Figure 5.3 the components (1) registry, (2) database, (3) end user API, and (4) admin API have been adopted. However, the registry was formerly implemented manually whereas the registry in Figure 5.4 is provided by Kubernetes. Additionally, the database component has been determined to be a Postgres SQL database. PostgresSQL is an open source object-relational database system [Thec]. The usage of PostgresSQL was requested by the stakeholders. The components (5) Kubernetes ingress, (6) Kubernetes master, (7) config microservices, and (8) data abstraction layer where added to the system's architecture. Such like the registry component, component (5) and (6) are provided by Kubernetes. The Kubernetes cluster is not referred to be a component in the architecture but illustrates that the application is deployed on a Kubernetes cluster.

The REST requests permitted by (3) and (4) have not been changed nor have any requests been added to the APIs. Therefore, Table 5.1 still holds for the prototype system.

Component (1) refers to the Kubernetes registry. The registry is a service provided by Kubernetes which is responsible for keeping track of the currently persisted configuration data. The registry is communicating with the (6) in case of querying additional configuration data or executing database queries [Fou]. Additionally, the registry knows the service names of each microservice in the cluster. If a user queries for a set of configuration data and provides a list of id's, the registry is able to look up the corresponding services.

Component (5) refers to the Kubernetes ingress object. As pods and services inside the Kubernetes cluster are not accessible from outside the cluster, the ingress object manages the external access to cluster objects [Fou]. The ingress provides a couple of functionalities, Among others, it can serve as a load balancer [Fou].

Component (6) refers to the Kubernetes master object. The master is a dedicated process that is part of Kubernetes' control pane [Fou]. The master itself runs a sequence of process in order to verify that current cluster state matches the desired state [Fou]. In this case, the master does know the amount of pods deployed on the cluster, whereas each pod holds an instance of component (7).

Component (7) refers to a microservice implementing the core functionality of the system. Each microservice references to a dedicated identifier which references a row in the database. Thus, each microservice is responsible for persisting exactly one configuration file. A microservice instance is deployed inside a Kubernetes pod, which is referred to as the smallest unit of deployment within Kubernetes [Fou]. There exists at least 2 instances of each microservice as required by the stakeholders. In general, Kubernetes enforces a self-healing mechanism by restarting components in its cluster that have been terminated. The master component (6) implements Kubernetes' self-healing, e.g., by initiating actions to recover the cluster state if any changes occur inside the cluster [Fou]. Kubernetes' self-healing mechanism and the microservice redundancy given in Figure 5.4 lead to an increased resilience to outages.

Component (8) refers to a data abstraction layer implemented with Spring Data Java Persistence API (JPA). The abstraction layer is responsible for decoupling the database from the application. Hence, it is possible to change the database based on the requirements, without changing the behavior or API of the application.

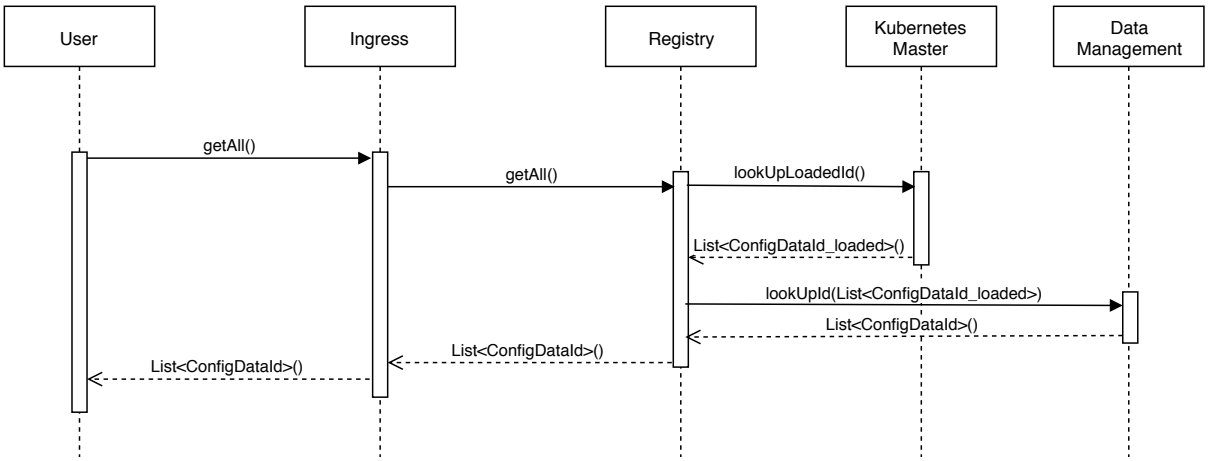

Figure 5.5.: Sequence diagram of the getAll request

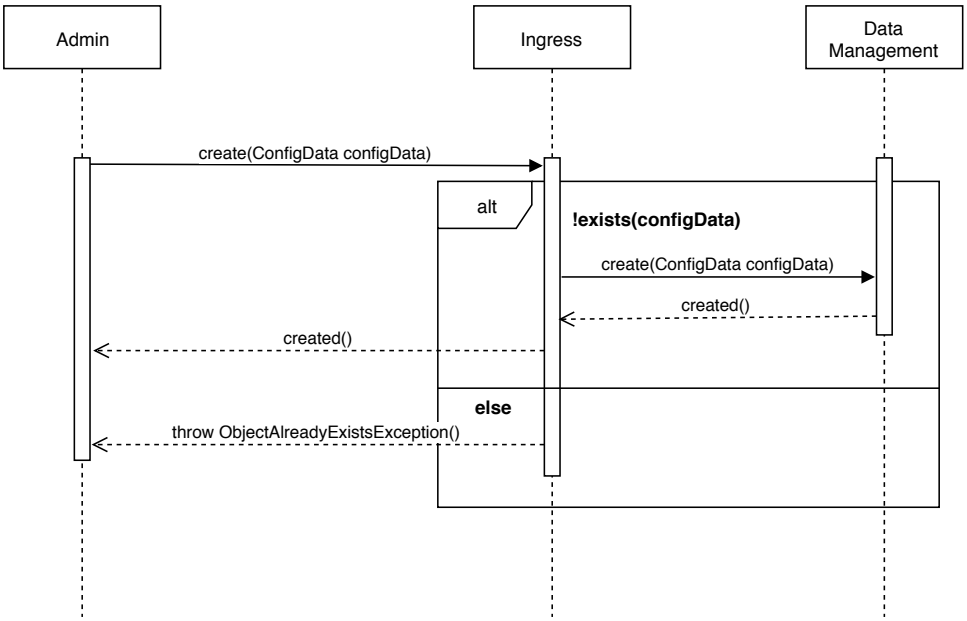

Figure 5.6.: Sequence diagram of the create request

5.4.2. Workflow and Behavior

Illustrating the behavior of the system supports the explanation of the application's structure outlined in Section 5.4.1. The following use case [Obj] and sequence diagrams [Obj] describe the behavior of the system illustrated in Figure 5.4.

Figure 5.5 consists of (1) the user, (2) the Kubernetes ingress object, (3) the Kubernetes registry, (4) the Kubernetes master object, and (5) the data management. This diagram illustrates the behavior and exchanged messages of the getAll() request defined in

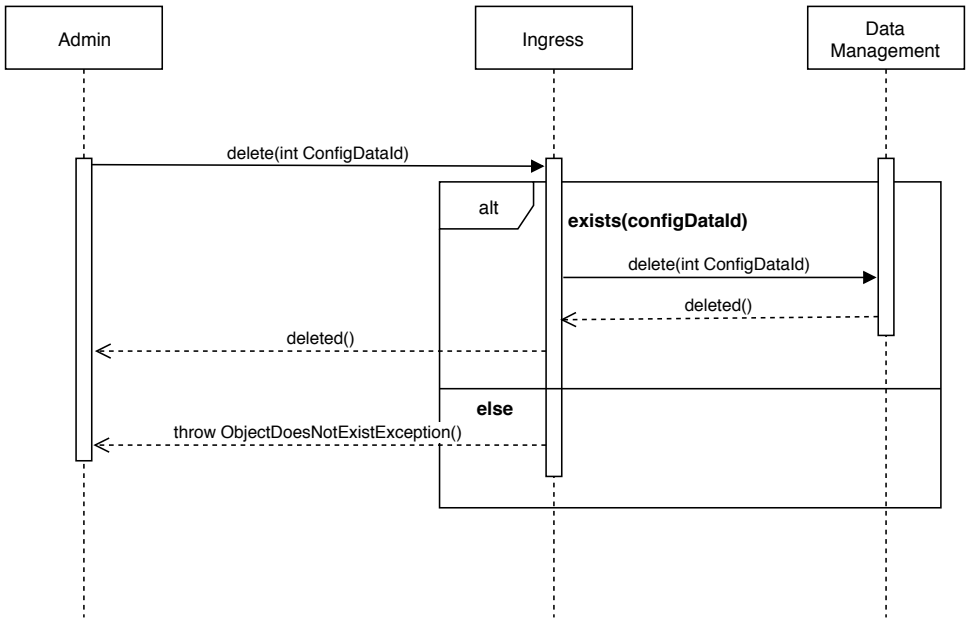

Figure 5.7.: Sequence diagram of the delete request

Table 5.1. The sequence diagrams illustrating the getAll() and getById() requests, are similiar and merely differ in returning different objects. Hence, Figure 5.5 serves as a reference for the getAll() and getById() request. For simplification, the data management actor in each subsequent diagram is a compound object of the data layer and data base component illustrated in Figure 5.4.

HTTP GET – getAll()

A GET requests to the corresponding end user API, calls the getAll() method. The (2) Kubernetes ingress object receives the request with appropriate parameters and forwards the request to the (3) registry object. Firstly, the registry queries the (4) master object. Then the master object, looks up the amount and type of currently persisted configuration data. In the case of a getAll() request, the response holds a list of configuration data IDs that are currently persisted in-memory. Next, in addition to the list of id's retrieved from the master, the registry queries the data management. The data management takes a list of configuration data id's as a parameter and looks up the corresponding objects in the database. The data management compares the retrieved list of available id's with the id's in the data base. In the end, the list's are merged as a list of all available configuration data id's and returned to the user.

HTTP PUT – create()

Figure 5.6 illustrates the permitted admin create() requests of Table 5.1. Additionally, Figure 5.6 and Figure 5.7 are missing the object (3) and (4) since the administrator's requests are not required to communicate with them. The admin defines the data content to be saved in the database and sends it along with the PUT request to the API. As before, the ingress object receives the request and forwards it directly to the data management. In this case, there is no communication with the Kubernetes registry object. The data management checks if a data object with the received content already exists in the database. If so, then the admin will receive an appropriate exception, stating that the object he wishes to persist already exists. Otherwise, the object is created and the admin receives an appropriate response code.

HTTP DELETE – delete()

Figure 5.7 shows the admin delete() request. Note that this request is almost identical to the create() request illustrated in Figure 5.6, except that the queried object will be deleted, if it already exists. In the case, that the object already exists, the admin will receive an appropriate exception as well.

To summarize, the described behavior of the prototype system is modelled in Figure 5.5, Figure 5.6, and Figure 5.7. Figure 5.5 described the behavior of the system, if a user requests to receive a list of configuration data id's or a single configuration data id. Figure 5.6 and Figure 5.7 explained the HTTP requests that are exclusively permitted to the administrator of the system, namely to create and delete an object. Note that every request, except delete(), communicates with the Kubernetes ingress object instead of communicating directly with the registry, enforcing loose coupling [FLR+14] in the architecture.

5.4.3. Architectural Styles

Representational State Transfer – REST

Unlike most monolithic applications, distributed systems communicate over a network, thus it is required to determine how messages are exchanged over the network. The prototype system in Figure 5.4 implements REST [FT00] as an architectural style to design the system's APIs. REST imposes a set of architectural constraints to its elements, e.g., providing a uniform interface between components to decouple services from each other [FT00]. The prototype's APIs are designed with REST without exception.

Client Server Model

Most distributed system logically adopt a common set of styles such as the client server model [Ben13]. In the case of the prototype system, a client server model has been adopted as well. A client server architectural style completely separates concerns of client and server implementation [FT00]. In the context of the prototype system, the client implementation is not important for this thesis. However, the prototype system modelled in Figure 5.4 can be referred to as the server implementation within a client server model.

Microservice based Architecture

Figure 5.3 describes a monolithic based architecture, where the entire application logic, database management, and graphical user interface is cumulated into one deployment. Figure 5.4 defines a microservice based architecture [New15] where each service is only responsible for one particular set of tasks. In the case of the prototype, the services have been identified by the stakeholders by clearly defining a business boundary for each service, e.g., a config microservice is only responsible for providing and persisting configuration data but not for querying data from the database. Microservices describe isolated units of work that can be developed and deployed independently [New15].

Kubernetes as a Platform

Kubernetes requires an application or microservice to be containerized. Meaning the application should be running inside a container and therefore providing some kind of container image, independent of the used container virtualization technology. Additionally, these images are distributed to pods which are deployed on a Kubernetes cluster. Another mechanism provided by Kubernetes is its ability to know the internal state of a cluster. For example if a pod in a cluster is shut down for some reason, Kubernetes recognizes the change in its cluster state, and performs actions to recover its state. The Kubernetes master object [Fou] is responsible for verifying that the current cluster state matches the dedicated deployment.

Kubernetes as a platform choice is commonly not referred to as an architectural style. However, by definition of Fielding et al. [FT00] given in Section 2.1.1, Kubernetes does impose a restriction on the relationship between architectural elements and how to encapsulate these elements. Hence, it is reasonable to state that Kubernetes as a platform choice can be labeled as an architectural style.

5.4.4. Design Patterns

As opposed to architectural decisions, the system illustrated in Figure 5.4 does not integrate design patterns that have specifically been chosen by the developers. Though, most design patterns are forced by the chosen technologies such as Spring Boot. Spring Boot integrates a lot of design patterns, therefore they will not be explained in detail. However, the most important ones that significantly change the implementation style will be briefly explained as well as other design patterns.

Aspect Oriented Programming – AOP

Aspect Oriented Programming (AOP) is viewed as the complement of Object Oriented Programming (OOP). Within OOP a class is the key unit of modularity, whereas within an AOP approach, the key unit of modularity is an aspect [Pivb]. Aspects allow to modularize crosscutting concerns such as transaction management and provide a different view of structuring classes and modules [Pivb]. AOP is integrated in the entire Spring ecosystem and not restricted to Spring Boot.

Inversion of Control

Inversion of Control (IoC) or Dependency Injection (DI) describes the reversal of dependencies of objects. Instead of leaving it to the dependent object's responsibility to find its dependencies, the dependent object is provided with the necessary dependencies by injecting them, e. g. through constructor arguments [Pivb]. DI drastically increases testability and readability of components.

External Configuration

Spring Boot enables the support of external configuration [Pivb]. This means, that configuration properties about an application, e. g. by using YAML files or command-line arguments can be decoupled from the application code. The configuration of an application is then extracted from the environment rather than from hard coded properties [Pivb].

Twelve Factor App Manifesto

The last design pattern complies with one factor of the twelve-factor app manifesto by Wiggins et al. [Wig11]. Wiggins et al. describes the twelve-factors as a methodology to build and deliver modern software-as-a-service. Commonly known factors are [Wig11]:

- Dev/prod parity – dev, test and deliver environment should be as similiar as possible
- Codebase – an application should only have one codebase tracked by a version control system, but many deploys
- Build, release, and run – Each stage, build, release, and run should be strictly separated from each other

Each of the twelve factors have been strictly followed for the prototype system. Architects of the prototype emphasized on the 3 factors external configuration, stateless, and dev/prod parity.

Chapter 6

Hazard Analysis

Running a chaos engineering experiment requires to know the type of experiment to be executed, i.e., it has to be decided which failures or turbulent conditions to inject into the system. Creating experiments for software systems results in a great amount of effort and should be planned accordingly in order to be certain about actually executing the experiments. Therefore, a sensitivity to failures has to be established by addressing chaos engineering in terms of identifying incidents in a system before it is pushed to production. A major obstacle is to identify such incidents in a meaningful way. Conducting a hazard analysis on a given system is a means to systematically identify potential hazards. According to the MIL-STD 882 [Sys], a hazard analysis is part of assessing a system's safety and by applying methods to identify incidents such as fault tree analysis, failure modes and effects analysis or computational hazard analysis [Sys]. These methods belong to a larger group of methods and techniques which are associated with safety assessment of software systems. Hence, the use of hazard analysis methods allows to make predictions and assertions of the prototype system with respect to its architecture. The fundamental principles of the methods used in this chapter have been explained in Section 2.2.

This chapter is structured as follows. Section 6.1 conducts a Failure Mode and Effects Analysis. Section 6.2 applies a Fault Tree Analysis. Section 6.3 applies a Computational Hazards and Operations Study. All results of the hazard analysis are attached in Appendix B.

6.1. Failure Modes and Effects Analysis

6.1.1. Summary of FMEA Findings in Appendix B.1 and Procedure

In this chapter, we will discuss only one result out of all FMEA results. The results of the FMEA can be found in Appendix B.1. The failure modes of the FMEA study have been elaborated qualitatively by thinking of different events, that might lead to an undesirable outcome. Usually, these failure modes and subsequent system effects are discussed in reviews with the stakeholders and developers of the system. This ensures that the elaborated findings have higher quality and are complete. In this case, thorough reviews have not been conducted due to time constraints. However, the results were reviewed by the lead architect and one developer of the prototype system during an one-hour session. Though, the review did not comprise of thorough discussions of the results but merely discussing the procedure and some of the results. Therefore , it is possible that the outcome of the FMEA is incomplete.

It becomes obvious that all components that have been investigated, share a common set of failure modes. As the prototype system is deployed on a Kubernetes cluster, all components are threaten by the loss of different Kubernetes objects, e.g., termination of pods or replicasets. Furthermore, since the prototype is very likely to be deployed on a managed Kubernetes cluster, most of the failure modes are left out of consideration by the developers, as stated by a developer of the system. However, as Kubernetes is configured to scale out pods as necessary, the termination of pods and unavailability of services may cause a whole new level of failures. As the severity levels of these failures imply, a loss of different Kubernetes objects leads to highly severe effects on the system and should be addressed accordingly. This issue could be addressed, e.g., by increasing the amount of replicasets and pods available in the cluster. Though, this requires more memory space since more objects have to be stored in side the cluster. Moreover, this might also lead to a negative impact on the performance of the cluster.

Other failure modes included modification of requests, e.g., changing the data content of requests, modifying request headers, changing the number of request parameters, and so on. Most of these issues should be addressed by the service's respective Spring Boot implementation, e.g., a not permitted number of request parameters without a correctly implemented Spring Boot controller, should return a corresponding error message to the user. Despite being undesirable outcomes, such events are claimed to not severely damage the system in such a way, that a manual operation of the cluster will be required. However, the severity of these events is still considered very high, since users will have a bad user experience.

6.1.2. Definition of Severity Levels and Documentation Style

Table 6.1 illustrates one result of the FMEA specific to the prototype system. The results were documented in a table which comprises five columns. Note that the number of columns and labels of the columns may change, depending on the use case and the people responsible for conducting such an analysis. The architectural elements, i.e., the investigated components, referenced in this section are illustrated in Figure 5.4.

The columns reference (1) the participating component, (2) a potential failure that affects the participating component, (3) sub system effects, (4) system effects, and a (5) severity level. The severity levels are set specific to the prototype system and outlined as follows:

- low – the potential failure will not have any harmful effects on the user or the system
- middle – the potential failure slightly affects the system in an imperceptible way
- high – the potential failure will very likely have a harmful effect on the user or the system
- very high – the potential failure will definitely have a harmful effect on the user and might damage the system

In Section 2.6 it has been discussed that chaos engineering focuses on surfacing potential weaknesses of a system with respect to prioritizing identifying conditions that might have a critical impact on the user of a system. Therefore, Table 6.1 and the other results of Appendix B.1, focus on events that might have harmful impacts on the user experience. Thus, the severity levels are defined from a user experience perspective, where each level is described as an intensification of the impact the event might have on the user experience. For example, a failure with a low severity level has almost no impact on the user experience, whereas a failure with a severity level of very high will result in receiving error messages or a total outage of the system. The administrator is considered to be a user of the system. Thus the severity level's semantics remain as they are for administrator related cases.

As FMEA is a qualitative approach, the assigned severity levels may deviate from the actual severity of a failure mode. Thus, the impact a potential failure has on the system might turn out to be higher than initially estimated. For example, a chaos engineering experiment that focuses on a failure mode that has an assigned severity level of low, might as well turn out to have severity level of very high.

No.	Components	Failure mode	Sub system effects	System effects	Severity level
1	Ingress	Pod that hosts Ingress is terminated	Registry does not receive any requests by the Ingress service	User cannot request the registry to handle his or her requests	very high
2	Ingress	Replicaset that hosts the Ingress is terminated	Registry does not receive any requests by the Ingress service	User cannot request the registry to handle his or her requests	very high
3	Ingress	Service that exposes the Ingress is removed	Ingress is still accessible inside the cluster but not from the outside	Users cannot access the system since the Ingress is not exposed	very high
4	Ingress	Deployment that hosts the Ingress is deleted	Kubernetes Master cannot check cluster state for the Ingress component	Kubernetes Master cannot heal the cluster state for the Ingress component if deviations of cluster state occur	high
5	Ingress	Clusternode that hosts the Ingress is terminated	Registry does not receive any requests	User cannot request the registry to handle his/her requests	very high
6	Ingress	Ingress receives an HTTP requests paired with a not allowed HTTP method	Ingress responds with status code 405 (method not allowed)	User receives an error message in addition to no data	very high

Table 6.1.: Result of a FMEA analysis for the Kubernetes Ingress component

6.1.3. FMEA of the Kubernetes Ingress Component

Table 6.1 shows one result of the FMEA with respect to the Ingress component. The Ingress service, as seen in Figure 5.4, accepts user and admin requests and forwards them to the appropriate services. Table 6.1 illustrates six failure modes labeled with a severity level of middle, high, and very high. The failure modes with an assigned severity level of high and very high are therefore considered to have a harmful impact on the user and the system. For example, row number 1 in Table 6.1 describes a failure mode that terminates the Kubernetes pod that is responsible for running the Ingress service. Since internal routing of requests is not visible to the user, the sub system effects are not of concern to the user. Though, the system effects will be visible to the user by receiving an error message. The event results in a system state were the user will not receive any data. Rows 2 to 5 describe similar events, where different Kubernetes objects which run the Ingress service are terminated or shut down.

Row 6 describes the mutation of the HTTP method of incoming requests. The Ingress service receives requests on a dedicated Unified Resource Identifier (URI) with the intention to route them to internal services. Before the request is routed to another service, the HTTP method of the request will be changed. Administrator-related endpoints, i.e., create and delete configuration objects, only allow HTTP POST, GET, and DELETE requests. If a request is sent with an HTTP method that is not defined on the URI, the service will return a status code 405 which states that the HTTP method is not allowed. Thus, the user will not receive any data. Consider the HTTP request in Listing 6.1.

Listing 6.1 HTTP POST request issued by a non-administrative user

```
POST /api/registry/getConfigurationIdentifiers HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.54.0
Accept: */*
Content-type: curl/7.54.0
Content-Length: 8
body: {}
```

Listing 6.2 Mutation of HTTP POST of Listing 6.1

```
DELETE /api/registry/getConfigurationIdentifiers HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.54.0
Accept: */*
Content-type: application/json
Content-Length: 2
```

Listing 6.1 shows HTTP header information of a POST request issued by a non-administrative user. The endpoint's permitted methods and access rights are shown in Table 5.1. Usually, as illustrated in Figure 5.5, the request is received and accepted by the Ingress service routed to the registry service where the data is further processed. Listing 6.2 illustrates the mutation of Listing 6.1 by changing the HTTP method from POST to DELETE. This event references row 6 of Table 6.1, where the HTTP method is changed to a method that is not permitted on the given URI.

Appendix B includes all results from the FMEA conducted on the prototype system. Considering that FMEA is a component-based approach the participating components in Appendix B are respective to the components in Figure 5.4 whereas every component has been considered in the analysis. All rows of the tables provided in Appendix B constitute to a specific event, e.g., a user will not receive data from the system whereas multiple rows may constitute to the same event. These events are further analyzed by conducting a fault tree analysis. It has to be considered that such events are not explicitly documented during a FMEA but emerge as a by-product of such an analysis. However, it is possible to extend the documentation by an additional column describing potential events.

6.1.4. Summary of Findings

In total, the FMEA provided a thorough analysis of individual components of the studied system's architecture where each component has been considered except the Kubernetes master and the configuration microservices. The Kubernetes master, as

well as the Ingress and Registry, are managed services provided by the Kubernetes platform. However, the Kubernetes master is a service majorly beyond the control of the developers. The configuration microservices were excluded from the analysis based on the little information available about the services. Hence, it was considered to be not meaningful to analyze the Kubernetes master and configuration microservices with respect to a FMEA. The findings of the FMEA revealed critical components that can be considered as single points of failures. This becomes more evident considering the number of severity levels with a label of *very high* versus the number of severity levels with a label of *low* or *middle* concerning a single component. Naturally the more failure modes result in a harmful event the more fragile the component is to the system. This ratio is arguable ,considering the fact that in most cases the goal of a FMEA is to surface only harmful events rather than non-harmful events. Though it is worthwhile to keep an eye on the ratio between different severity levels.

In total, the severity level ratio with respect to the Ingress component lead to 22 failure modes which are equal to 22 rows within a FMEA table. In the range of these 22 failure modes, four were labeled with a severity level of *middle*, four with *high* and 14 were stated to be *very high*. There is a narrow margin between a severity level of *high* and *very high*, thus, these two may not necessarily be viewed as two separate labels if the number of harmful events is considered. This results in 18 out of 22 failure modes, or approximately 82% of failure modes, that may have a harmful effect on the user and the system. In contrast, the registry shows a severity level ratio of approximately 94% which is almost equal to the data management's, administrator's and user's component ratio.

In consequence, of almost every component being a single point of failure it becomes quite difficult to minimize the blast radius of a chaos engineering experiment. Additionally, it has to be considered that FMEA as well as FTA and CHAZOP is a qualitative approach. Though the findings have been reviewed by the developers once, they still may be incomplete. This inevitably leads to the assumption that even if the blast radius is as minimal as possible with respect to the findings there might occur unconsidered failure modes. This leads to a false perception with regards to the blast radius where it is larger than initially expected. Nevertheless, the findings provide examples where the blast radius can be considered small, e.g., failure modes with a severity level of *low*. Likewise failure modes that have a severity level of *very high* but merely result in a decrease of the user experience, may be considered suitable for a chaos experiment. These experiments may not lead to a critical outage of the system but are still considered to be harmful due to a decrease in user experience.

6.2. Fault Tree Analysis

6.2.1. Summary of FTA Findings of Appendix B.2 and Procedure

Section 2.2 explains that the FTA is a deductive technique originating of a top-level hazard, down to its root causes, therefore, emphasizing on finding cause-effect relationships between faults within a system. Any fault tree begins with a top-level hazard that is the result of a preliminary hazard analysis such as FMEA. Afterwards, top-level hazards are further expanded by thinking of events that are a direct cause of their parent event. This continues until an event is reached, which can not be further analyzed. Hence, the event should be illustrated as a basic event. This results in the leaf nodes being basic events, which are connected by logical gates. The basic events are then used to describe the fault tree as a boolean expression to calculated the minimal cut set, i.e., the minimum amount of basic events that need to occur in order for the top-level hazard to be triggered. The trees are constructed with the logical gates and events given in the figures of Section 2.2.2.

The results of the FTA, found in Appendix B.2, obviously look similar to each other. This becomes more evident, by reference the findings of Section 6.1, where it has been explained, that each investigated component of the prototype system, shares a common set of failure modes. These failure modes exist in the fault trees at the bottom of the tree, i.e., the leaf nodes which are stated to be basic events. As fault trees do not merely consider single components as in the FMEA, but rather comprise of events that result from multiple interactions between components. Further, it can be the case that multiple fault trees with distinct top-level hazards comprise the same components or component interactions in terms of events. Thus, fault trees with distinct top-level hazards may look similar to each other with respect to the leaf nodes and minimal cut sets. In this case, the fault trees of the prototype system look almost similar to each other.

Appendix B.2 shows two more fault trees considering top-level hazards with respect to the administrator. In case of administrative tasks, different top-level hazards have been identified which correspond to the administrator use cases identified in Figure 5.2. However, among the administrator fault trees, the results are equal to each other except the top-level hazard. This results from the implementation of the use cases. Considering the sequence diagrams Figure 5.5 and Figure 5.6. The work flow of the individual requests concerning the administrator are equal to each other. Thus, implying that the underlying events that may result from these interconnections result, must be equal. In the end, this also leads to an equality of the minimal cut sets. The cut sets revealed, that the system must comprise multiple single points of failure. This becomes evident by emphasizing multiple occurrences of single entity cut sets, such as in Equation (6.3).

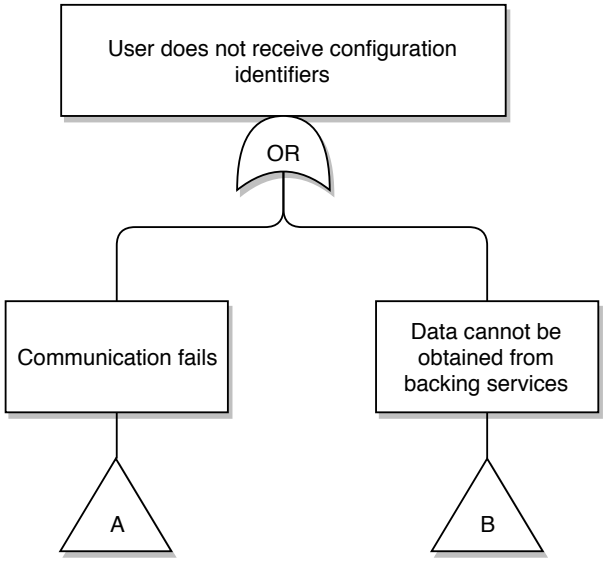

Figure 6.1.: Fault tree with top-level hazard: "User does not receive configuration identifiers"

This has been discovered to be a major issue in the current prototype system and will be further investigated during the chaos engineering experiments.

6.2.2. Top-Level Hazard: User does not receive configuration identifiers

In the following, a dedicated example of the FTA results of Appendix B.2 is discussed. Figure 6.1 shows a fault tree where its top-level hazard results of the failure modes in Table 6.1. The top-level hazard describes a scenario, where the user will not receive any data from the system.

In Figure 6.1 two child nodes are connected by an OR gate to the top-level hazard. Thus, if either event occurs, the top-level hazard will also be triggered. The left child node represents the loss of communication between any component responsible for implementing the use case of a non-administrative user, shown in Figure 5.2. This affects the Ingress component, the registry component, the Kubernetes master node, configuration services that hold configuration objects of the requested identifiers, and the database. The right child node represents an event, where the Registry is not able to retrieve the requested data. Note that each child node represents an intermediate event as explained in Section 2.2. Thus, being events that have to be further analyzed.

The input symbols A and B are optional operators that allow subtrees to be taken as input in order to increase the readability of fault trees. In this case, A and B receive sub

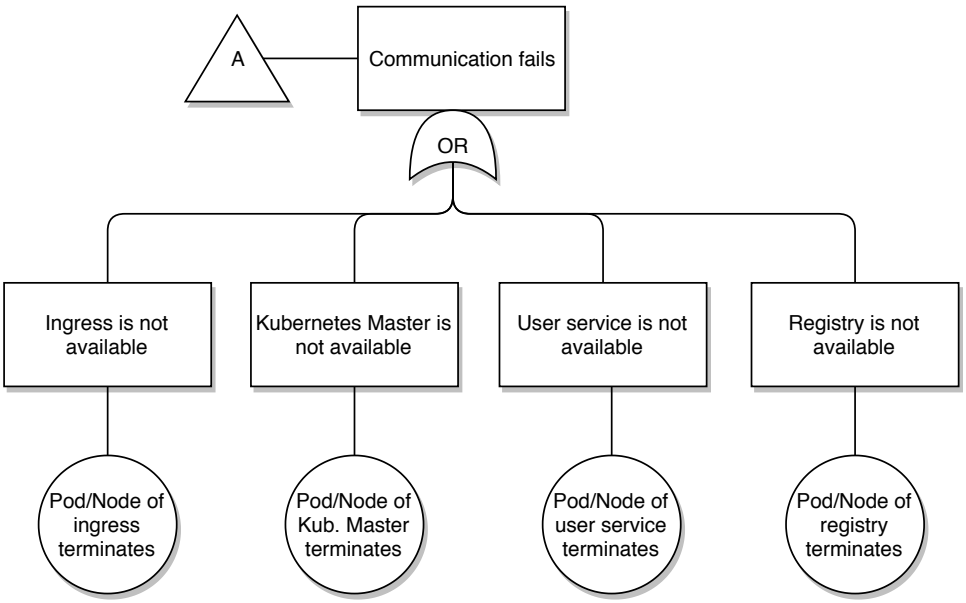

Figure 6.2.: Left sub tree of Figure 6.1

trees as an input, where A receives the fault tree of Figure 6.2 as input and B receives the fault tree of Figure 6.3 as input.

6.2.3. Analysis of Subtree A: Communication fails

Figure 6.2 shows the sub tree of the left child node of Figure 6.1. The child nodes of the top-level hazard, illustrated in Figure 6.2, are connected by an OR gate. Furthermore, every child node represents an intermediate event.

Further analysis of the intermediate events leads to the conclusion, that each basic event describes the termination or unavailability of a Kubernetes object that is responsible for running the corresponding service. For example the basic event of the left child node states that in order for the Ingress to become unavailable the termination of a pod or node of the Ingress component must occur. The leaf nodes in Figure 6.2 are thus the root causes of the top-level hazard of this sub tree. Since they are connected by an OR gate the occurrence of either event will lead to a loss of communication.

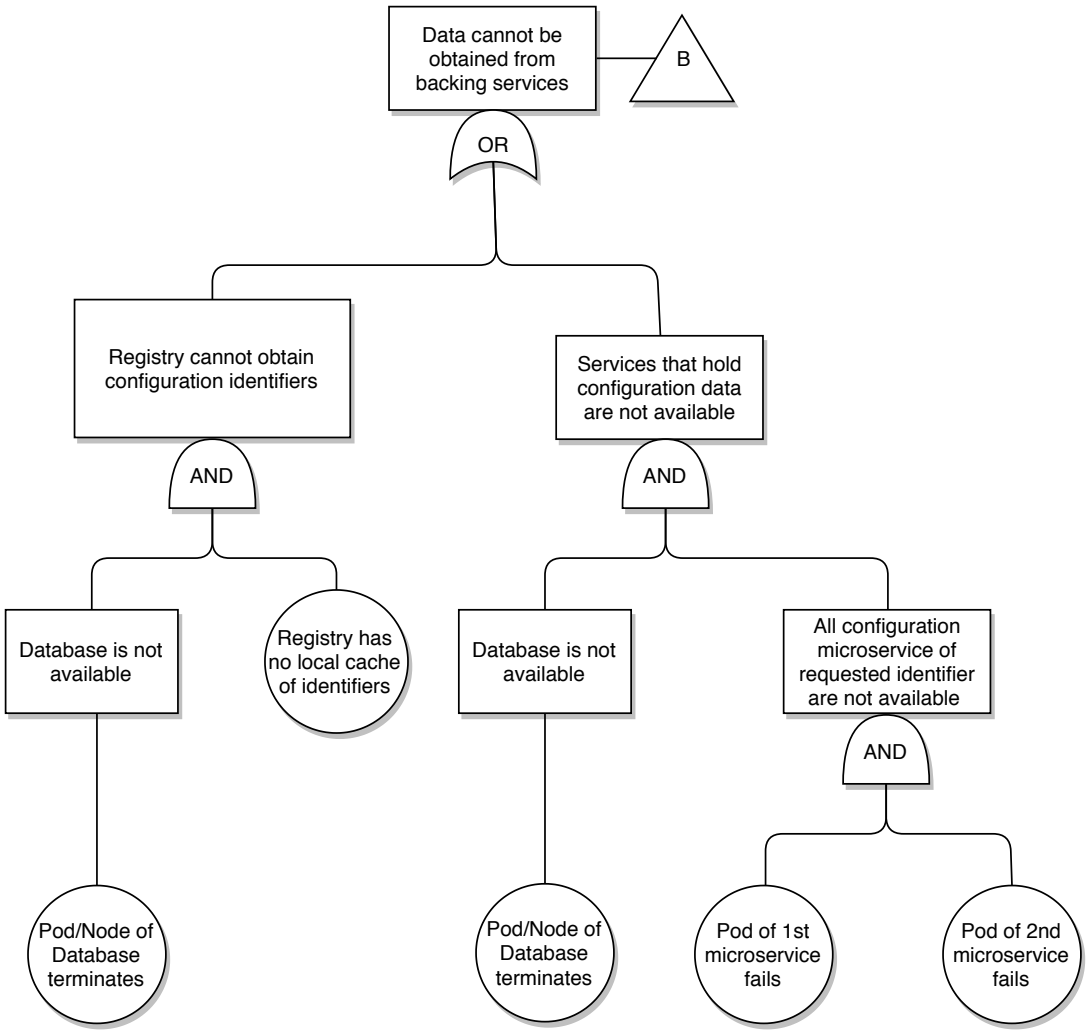

Figure 6.3.: Right sub tree of Figure 6.1

6.2.4. Analysis of Subtree B: Data cannot be obtained from backing services

Figure 6.3 illustrates the right sub tree of the right child node of Figure 6.1 which is passed as an input to B. The structure of Figure 6.3 shows a higher complexity than Figure 6.2, based on the number and interconnection of its events. The top-level hazard has two child nodes connected by an OR gate. The left child node describes an event where the Registry is not able to retrieve the identifiers of the requested configuration data. This occurs only if both of its child events occur at the same time, since they are connected by an AND gate. The right child is a basic event, that describes that the Registry does not provide any locally stored data that can be accessed. Consider, that this inevitably leads to the Registry querying the database for the necessary identifiers.

Label Description	Label
Pod/Node of Ingress terminates	A
Pod/Node of Kub. Master terminates	B
Pod/Node of user service terminates	C
Pod/Node of Registry terminates	D
Pod/Node of Database terminates	E
Registry has no local cache of identifiers	F
Pod/Node of 1st microservice terminates	G
Pod/Node of 2nd microservice terminates	H

Table 6.2.: Description of basic events

The left child event is an intermediate event, stating the unavailability of the database that occurs, if the corresponding pod or node that runs the database terminates.

Proceeding with the right child node of the top-level hazard, it is stated that responsible services that hold the requested configuration objects are unavailable, i.e., the database and the underlying configuration services that provide the configuration data. These two events are illustrated as intermediate events. The basic event that causes the unavailability of the database is equal to the basic event on the left side of the subtree. However in order for the configuration services to be unavailable, both instances of the same configuration service must become unavailable at the same time thus both basic events are connected by an AND gate. This is a result of the build in redudancy of the configuration service i.e., that always two instances of the same configuration service must be running.

6.2.5. Minimal Cut Sets

The minimal cut sets of Figure 6.1 are calculated by applying basic principles of boolean algebra, as explained in Section 2.2. Every FTA connected with multiple AND or OR gates can be expressed as a boolean expression by following the convention of boolean algebra. It has to be emphasized, that only basic events are considered. Intermediate events or other types of events will be ignored.

Label	CS 1	CS 2	CS 3	CS 4	CS 5	CS 6	CS 7
A	x						
B		x					
C			x				
D				x			
E					x	x	x
F					x		
G						x	
H							x

Table 6.3.: Mapping of basic events to their occurence in the minimal cut set

Table 6.3 shows a mapping of the basic events to their occurrence in the minimal cut set. A description of the labels is given in Table 6.2. Given the FTA of Figure 6.1 and the mapping shown in Table 6.3. The following boolean expression

$$
\begin{aligned}
FTA_{UserDoesNotReceiveAnyData} &= A + B + C + D + (E * F) + (E * G * H) \\
&= A + B + C + D + E * (F + G + H) \\
&= A + B + C + D + (E * F) + (E * G) + (E * H)
\end{aligned} \tag{6.1}
$$

references the given FTA. In conclusion the minimal cut set can be described as:

$$Minimal\ cut\ set\ m = \{\{A\}, \{B\}, \{C\}, \{D\}, \{E, F\}, \{E, G\}, \{E, H\}\} \tag{6.2}$$

6.2.6. Summary of Findings

The FTAs, referenced in Appendix B.2, have been constructed by systematic deduction of the top-level hazard. In order to evaluate the trees with respect to their minimal cut set, the basic events were assigned labels by mapping the events to an identifier such as in Table 6.3. The mapping allows to describe each FTA as a boolean expression based on the logical connectors AND and OR. Note that the mapping is not restricted to use the same labels as in Table 6.3.

Equation (6.2) shows the minimal cut set of a fault tree with respect to a top-level hazard that references an administrator use case. Every set in the equation references

a basic event in the FTA that is connected by a logical gate. Furthermore, it shows that each subset of m is a potential failure that upon occurence results in the top-level hazard to occur. Additionally, this strengthens the conclusion of the existence of multiple single points of failures. In general, it is reasonable to imply that a system's vulnerability increases, if a minimal cutset m contains more single entity subsets. A single entity subset is a subset that contains only one basic event, such as the subset $\{A\}$ in Equation (6.2). In contrast, it can be argued that if a minimal cutset contains more subsets with multiple basic events, such as $\{E, F\}$, the system's vulnerability decreases since all basic events of this subset are connected by an AND gate. Though, these observations are arguable considering that no probabilities of such events exist. For example, assuming every single entity subset of the previously described minimal cutset has a probability of 0. Then, the cutset is not considered to be fragile, as the probability that these single points of failure will get into a failure state, is equal to 0.

Considering the results of Section 6.1 it becomes evident that multiple failure modes, result in the top-level hazard described in Figure 6.1, where the user does not receive configuration identifiers. Therefore, the fault tree discussed in this section, is representative for a large range of failure modes. Despite changing the failure modes, the fault trees of the resulting events do not necessarily change, since how the events interact with each other remains the same.

With respect to user-focused events, there does not exist a wide variety of distinct events that would be meaningful to analyze other than the ones discussed in this section. This is different to the administrator-focused events. In fact, the FMEA revealed three fault trees with respect to the administrator. Interestingly based on the way how the components of the system communicate as shown in Figure 5.5 and Figure 5.6, both fault trees are equal in their structure and semantic. However, the minimal cutset of the user top-level hazard provides a lower vulnerability than the minimal cutset of all admin cutsets. The minimal cutset of the administrator as shown in Equation (6.3) contains merely four single entity subsets resulting in four single points of failure.

$$FTA_{AdministratorCannotCreateData} = \{A, B, C, D\} \tag{6.3}$$

The findings of this analysis were reviewed by the developers once with respect to a common understanding of how the system is expected to work. Suspicious is the behavior of the managed services provided by Kubernetes. As already discussed in the previous section and outlined by the interviewees it is not clear how the system responds upon any alterations done with respect to the provided services. The fault trees are one possibility to extract chaos experiments with a small blast radius. For example, a

possible experiment might state a hypothesis about the state of the application after one of the two microservices in Section 6.2.4 shuts down. Here again it must be considered that the blast radius might be larger than expected due to the unknown behavior of the Kubernetes services.

6.3. Computational Hazard Analysis and Operability

6.3.1. Summary of CHAZOP Findings of Appendix B.3 and Procedure

CHAZOP is a common technique in chemical industry in order to analyze deviations in chemical fluid flows and system design [FH94]. A CHAZOP study in software engineering aims to analyze interconnections between components and services, according to a combination of guidewords and attributes. In this particular case, the analyzed interconnections reference the sequence diagrams in Chapter 5 and comprise of components of the prototype system. It has to be considered, that the documentation style and semantics of guidewords of a CHAZOP study may deviate from other CHAZOP studies depending on the studied system and the researcher's intention.

However, the interconnections of this CHAZOP study focused on the data flow between two components or services. The data flow attribute has been combined with different guidewords, defined in Table 6.4. Based on this combination of attribute and guidewords, the causes were qualitatively elaborated as in the previous approaches. As CHAZOP is also a qualitatively group activity process, the results may be incomplete due to the lack of thorough reviews. Moreover, the documented result left out additional information, such as prevention techniques that explain how different causes and consequences could be avoided. As it is not the intention of this work to solve individual weaknesses of the investigated system, but rather surfacing them, the prevention techniques have been omitted. A lot of the identified consequences are similar to the system effects of the FMEA. Furthermore, investigating the findings of the CHAZOP, it becomes obvious that most findings concern about the abortion or delay of requests. As the system reflects a business critical system that aims to create revenue off of users, response times of requests become extremely important.

The criticality of the causes based on time outs and delays has yet to be determined. The interviews and architecture description did not provide any information about possible time out or abortion rulings inherent in the system. Therefore, it is almost impossible to derive any consequences that may result from a delayed or aborted request. The CHAZOP study of every interconnection implies, that delays and abortion of requests should be further investigated, e.g., executing chaos engineering experiments that inject

service call delays into the system. It has also been revealed, that the Kubernetes Master is a majorly important component that must be alive in order for the entire system to work. Though, further analysis of the master component is difficult yet impossible in terms of applying further analysis approaches such as a FTA or FMEA. This is based on the lack of information about how the Kubernetes Master reacts to certain events, e.g., shutting down due to a failure. Applying chaos engineering experiments on the Kubernetes Master must be done carefully due to this lack of information. Discussions with the lead architect of the system, in context of the CHAZOP findings, revealed that it is not clear what would happen if some of the findings of the CHAZOP study, would be applied in a chaos engineering experiment. This is different to other findings, e.g., from FMEA, where some expectation about possible failures exists. Though, as the Kubernetes Master is a component that actually manages the cluster, the results from applying chaos engineering experiments may result in a total outage of the cluster.

However, a major finding of the CHAZOP study is the necessity of applying chaos engineering experiments that concern about request abortions and request delays, as these have not been sufficiently explored by the developers of the system. Incidents that result from these causes are, in their severity, equivalent to a total outage of the system, as they lead to higher response times which may lead to a loss in customer revenue.

6.3.2. Definition of Guidewords and Documentation Style

In this case study, the results of the CHAZOP study are documented with a table similar to Table 6.1. The columns describe (1) the number of the investigated scenario, (2) the interconnection and flow, (3) the attribute respective to the interconnection, (4) a guideword, (5) possible causes of the guideword and (6) consequences and implications. The guidewords used in this CHAZOP study are based on the guidewords described in Section 2.2. Table 6.4 shows the semantics of the guidewords that have been used throughout this CHAZOP study.

6.3.3. Analysis of the Interconnection: Ingress to Registry

Table 6.5 shows a sample from the CHAZOP study results conducted on the prototype system. The interconnections are selected with respect to the interconnections illustrated in the sequences diagram Figure 5.5. In total, the CHAZOP study covers all communication paths illustrated in the sequence diagrams of Section 5.4.2. The attribute *data flow* describes that data is transmitted between the participating components. In this case the type of data is not specified however there may exist CHAZOP studies that document

Guideword	Semantic interpretation
no	No data is sent, received or exchanged as expected
more	More data is sent, received or exchanged as expected
less	Less data is sent, received or exchanged as expected
reverse	The communication, sender and receiver is reversed
early	Data is sent or received earlier than expected
late	Data is sent at a later point in time
before	Data is sent or received before a necessary event occurs
after	Data is sent or received after a necessary event occurs
part of	Data is sent or received that is part of a different interaction
other than	Data sent or received is different than the expected data
as well as	Two conflicting events happen at the same time during one interaction of two components

Table 6.4.: Guidewords related to CHAZOP study

the transmitted data type, e.g., XML or JSON. The semantics of the guidewords have been explained in Table 6.4.

The arrows next to the names of the participating components describe the direction of the data flow. For example, the arrow in the second column of row one is directed to the right, indicating that the data is sent from the Ingress to the Registry component. In contrast an arrow that is directed to the left indicates that the data flows from the Registry to the Ingress. The causes in the fifth column are the root cause for the selected guideword. For example, in case of the guideword *none*, the root cause for no transmitted data may result from an abortion of a sent request. The consequences in the last column are the result of a combination of attributes and guidewords.

The sample shows two participating components of Table 6.5 namely the Ingress and Registry component. The attribute indicates the investigated property of the interconnection combined with the guideword *none*. So far, the first row describes a scenario where there is no data sent or received between the two components Ingress and Registry. This may happen due to a request being aborted by the system. The abortion of the request leads to several consequences. Summarizing the consequences of the first row leads to the assumption that the combination of the attribute and guideword with respect to the interconnection of the Ingress and Registry results in either long response times or the user receiving an error by the system.

The subsequent rows of Table 6.5 emphazise the necessity to analyse one attribute with respect to multiple guidewords in order to get a better coverage of potential design

No.	Interconnection and Flow	Attribute	Guideword	Causes	Consequences
1	Ingress-Registry (→)	data flow	none	request was aborted or timed out	• Registry does not receive requests • Registry is going to be overloaded • Ingress may start a retry • Ingress does not receive any data • User waits long for a response
2	Ingress-Registry (→)	data flow	other than	Data content of response from Registry is removed	• Ingress receives an empty response • User does not receive any data
3	Ingress-Registry (→)	data flow	other than	Data content of response from Registry is changed to random list of configuration objects	• User receives data that he has not asked for • User experience is reduced • User has to request data again
4	Ingress-Registry (→)	data flow	other than	HTTP method is not permitted, header information is wrong or there is no controller present for the given URI	• Registry cannot handle request • Registry returns an error • User receives no data
5	Ingress-Registry (→)	data flow	less than	Remove identifiers in case of retrieving single objects by their respective identifier	• Registry returns an error since required identifiers are missing • User receives an error • User receives no data
6	Ingress-Registry (→)	data flow	more than	Body of request contains multiple identifiers instead of a single identifier	• Registry does not provide a controller to handle multiple identifiers in a request • Registry returns an error • User receives an error
7	Ingress-Registry (→)	data flow	late	request is delayed or request is aborted and client starts a retry	• Increased response time • User may abort the process

Table 6.5.: Result of a CHAZOP study of the interconnection between the Ingress and Registry

deviations. Additionally, the sample illustrates that some interconnections do not require the application of all guidewords. This occurs by analyzing interconnections where the application of specific guidewords do not make sense.

Inspecting the rows 2 to 4 the guideword *other than* states that the transmitted data to be different than expected. For example in row number two the root cause is stated to occur from the removal of the data content of a request forwarded by the Ingress. In this case the Registry expects a request that comprises either a list of parameters identifying a subset of configuration data or an empty list of parameters. For example the list of parameters may be changed so that the response will contain no data. This will result in the user not receiving any data as the scenario before.

Row 5 defines a partial loss of data indicated by the guideword *less than*. For example in case of a request where a single object will be retrieved by its identifier, the identifier passed as an argument will be removed. This may lead to the Registry returning an error to the user. The sixth rox describing the guideword *more than*, is intended to be the opposite *less than*. Given the previous request where an object will be retrieved by its identifier. Instead of passing a single identifier to the request, a list of identifiers is passed as arguments. Since the Registry may not provide a controller for such a request, the Registry will eventually return an error to the user.

The last row describes a common scenario with respect to the guideword *late*. Consider a request being aborted out due to some latency issue. Let's assume that the Ingress will start a retry in case of abortion. The response times will eventually increase due to the latency issue. An increase in response times will therefore lead to a decrease in user experience and may eventually result in the user aborting the process.

6.3.4. Findings of CHAZOP study

The CHAZOP study has been conducted on each interaction considered in the sequence diagrams of Section 5.4.2, as these are considered to be the interconnections that contribute to the system's use cases. The results were discussed with the same people who have been interviewed in order to gather information about the studied system. In general it can be said that interconnections between Ingress, Registry, and the database are considered to be the most critical. This becomes obvious by investigating the results provided in Appendix B. Considering the consequences of more common cases such as the loss and delay of information exchange, indicated by the guidewords *none* and *late*, the interconnections between these components are extremely fragile. However, it has to be mentioned that during discussion of the CHAZOP results, the developers are not aware nor certain about the consequences of the investigated interconnections with respect to the applied guidewords. For example, it is unclear what exactly will happen if a request, sent to the Registry and issued by the Ingress, is aborted. These edge cases that have emerged during the CHAZOP study, may be potential candidates to be evaluated during a chaos experiment but are also difficult to handle. This becomes more evident considering that almost each edge case leads to a system failure which is considered to be unsuitable for evaluating with a chaos experiment. Due to time constraints, the results of the CHAZOP study have only been reviewed by the developers once which leads to the assumption that the results of the study may be incomplete. Therefore, a thorough review of the CHAZOP results is considered to be meaningful in order to increase the quality of the study results.

As illustrated in Chapter 5, the architecture of the system comprises several managed services provided by the platform Kubernetes which have not been manually imple-

mented. However the services provided by Kubernetes were still considered during the CHAZOP study as shown in Appendix B since they are critical pieces of the system's architecture. Though it is yet unclear how the managed services such as the Ingress, will behave during a chaos experiment with respect to the findings of the CHAZOP study. Hence chaos experiments which involve these interconnections should be run in a non-harmful environment unlike production. Additionally it is advised to further elaborate the study results in order to build trust in the findings and enable a better formulation of experiments.

6.4. Summary of Hazard Analysis

The findings of the hazard analysis revealed vulnerabilities of the system which might have been discovered during a chaos experiment if the analysis would not have been executed before. Conspicuously is that the developers are not fully aware of most findings revealed by the FMEA and CHAZOP study as seen in Table 6.1 and Table 6.5. This is especially true for the components and interactions that include managed services such as the Ingress and Registry. It is true that those services are to some degree configurable and manageable by the developers unlike the Kubernetes master.

However, edge cases yield high potential to be incredibly difficult to evaluate during a chaos experiment. This is based on the unforeseeable behavior of the Kubernetes platform. The developers agree on the fact that commonly these services are not forced to terminate and they agree further on that the system is very likely to go down entirely if one of these components shuts down independent of the root cause.

The findings further showed that the system contains multiple single points of failure which almost include all managed services except the database and the configuration microservices. Consider the minimal cutsets in Equation (6.2) and Equation (6.3) of the corresponding user-focused and administrator-focused fault trees. From a chaos engineering perspective, the strong interconnection of the managed services and the rest of the system makes increases the complexity of creating chaos experiments with a small blast radius. Independent of the stage where chaos experiments are executed, executing an experiment that almost certainly crashes the system yields little to no experimental value.

Arguably, missing reviews on the findings and longer investigation cycles might have led to missing out findings that could have provided a higher value for creating chaos experiments. Currently, the most interesting experiments include edge cases where the developers are completely unaware of the consequences though it is difficult to minimize the blast radius. Nevertheless, the blast radius can be reduced by focusing on a single

component. It must be further evaluated if the blast radius is small enough with respect to an experiment that includes a design deviation revealed by the CHAZOP study. At first, it is considered to be more reliable to focus on a single managed service, e.g., Registry, instead of a communication between two managed services, e.g., as shown in Table 6.5.

In the end, there will be four experiments extracted from the findings. The experiments were selected by prioritizing a small blast radius followed by an estimated impact on the system as illustrated with the severity levels of Table 6.1. The first two experiments will be extracted from the minimal cutsets shown in Equation (6.2), i.e., the subsets $\{E, G\}$ and $\{E, H\}$ may be altered in order to illustrate an increasing blast radius. For example, by starting with focusing on the basic event $\{H\}$, i.e., terminate a pod of a configuration microservice, in the first experiment and then gradually increasing the blast radius by targeting two basic events $\{E, G\}$, i.e., terminate the pod of the database and the pod of a configuration microservice. The third experiment will increase the blast radius further by adding additional configuration data objects in the database. The experiment configuration will be equal to the second experiment but more configuration microservices will be available. The last experiment will cover the usage of the Chaos Monkey for Spring Boot [Wil18]. In this experiment, the chaos monkey will be configured to inject latency into two service calls. This experiment references to multiple findings from the CHAZOP study.

Chapter 7

Experiment Environment and Tool Configuration

The general process of conducting research in computer science and other research domains is commonly understood to include some kind of measuring, interpreting and reporting the results where reproducibility is a key concern of the scientific method [PVB+19]. Based on the findings of Chapter 3 it may become obvious that currently no real standard exists in the field of chaos engineering for reporting chaos experiments. However missing standards of metric selection, measurement, and monitoring as well as heterogeneity among chaos engineering tools increases the complexity of reporting reproducible chaos engineering experiments. In order to be reproduceable it is considered that a chaos experiment report should at least include a description of the investigated system, if possible provide the system as an executable, a description of the steady-state and how it was measured, a set of hypothesises related to the steady-state and the declared chaos engineering experiments with observed measurements. In addition to the chaos experiment related information, a report should also include any information about the environment where the steady-state was measured and where the experiments have been executed.

This chapter is structured as follows. Section 7.1 describes the cloud environment, i.e., the experiment environment. Section 7.2 explains the deployment of the prototype system. Section 7.3 gives a description of the JMeter configuration. Due to customer anonymization it is not possible to provide the investigated system in the final report.

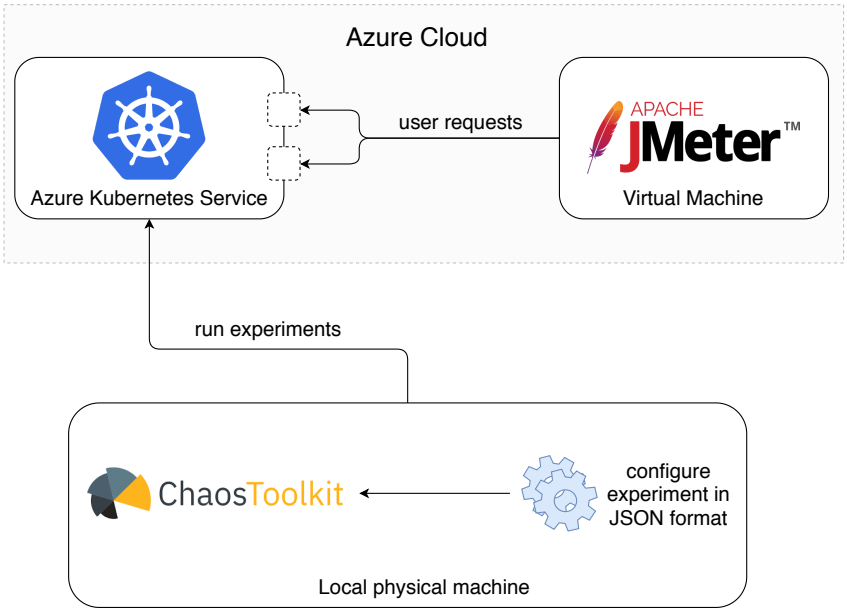

Figure 7.1.: Experiment environment

7.1. Cloud Environment

The deployment platform Microsoft Azure [Mica] is going to be used as a cloud provider. This work will not cover a detailed explanation of Microsoft Azure nor any evaluation of different cloud providers. Also since the prototype system is deployed to a Kubernetes cluster, explanations with regard to any Kubernetes related configurations that need to be performed, e.g., kubectl, will be omitted as well. Therefore, the platform and corresponding configurations can be stated as a given in the scope of this study.

Figure 7.1 illustrates the environment and interaction of the deployed applications. As shown two applications are currently running in the Azure Cloud. The left application is a managed service by Azure, i.e., Azure Kubernetes Service (AKS) [Micb]. AKS provides a Kubernetes cluster with the possibility to deploy containerized applications. The Kubernetes API is equal to any locally deployed cluster except that AKS is able to provision more than one cluster node and takes care of several infrastructure-related concerns that are hidden from the user. The AKS runs on version 1.12.8 and is hosted in the region *North Europe*. Furthermore a horizontal pod autoscaler is enabled with a target value of 80%, i.e., the autoscaler object will calculate a quotient from the required resources from the target value with respect to the current metrics and scale the replicasets or pods respectively.

The right side of Figure 7.1 shows a Virtual Machine (VM) that runs the load testing tool Apache JMeter. The VM is deployed to the region *North Europe* and runs Ubuntu

Listing 7.1 Provisioning deplyoments for AKS cluster

```
kubectl apply -f registry.yaml
kubectl apply -f admin.yaml
kubectl apply -f config-database.yaml
kubectl apply -f ingress.yaml
```

as its Operating System (OS) with a memory size of eight Gigabyte. It is intented to deploy both applications to the same region in order to minimize the relative distance between both applications considering reduction of latency. Apache JMeter runs on version 5.1.1. which requires at least Java version 8 or higher whereas the VM runs Java version 11.0.3.

The properties of the application running in AKS will be changed during JMeter's test period by running Chaostoolkit from a distinct physical machine. The physical machine runs on MacOS as its host OS. Chaostoolkit runs on its newest version 1.2.0 with the Kubernetes driver installed and enabled. As the Chaostoolkit Command Line Interface (CLI) requires a working Python installation [Chab] the physical machine running Chaostoolkit has Python 3.7.3. installed. The chaos engineering experiments will be declared as JSON files though it is possible to declare experiments in YAML as well. JSON is selected out of personal preference where the file type has no impact on the quality of experiments. The structure of the experiments will follow the explanations in Section 2.4.5.

7.2. Azure Kubernetes Service – Deployment

In the following, it is assumed that kubectl is connected to the remote cluster on Azure. Furthermore, the following deployments and service names have been altered in order to ensure costumer privacy. As the prototype system is given in this work a set of Kubernetes deployment files is already in place.

In order to deploy the prototype to Azure the AKS will receive a set of deployments which will be passed to the cluster by using the Kubernetes API. Respectively Listing 7.1 shows a series of commands that send the appropriate deployments to the AKS cluster. After receiving the deployments, AKS will immediately begin to spin up containers and expose services if declared. After all pods are running the application is ready to receive requests.

In the current deployment, the cluster runs one pod for each dedicated service. It is expected that the application dynamically creates configuration microservices if there exists no running container with a configuration microservice for a dedicated

configuration object in the database. For example, if the database holds 100 different configuration objects, upon initial start Kubernetes will create 100 different configuration microservices for each individual configuration object that is stored in the database. In addition, the stakeholders insisted that for each configuration microservice a minimum number of two pods should exist for redudancy purposes which is also specified in Chapter 5.

7.3. Simulating and Monitoring Environmental Conditions

As discussed in Chapter 5 the prototype system has never been exposed to real user behavior, e.g., in a production environment. Thus, in order to run meaningful chaos experiments it is necessary to simulate the expected user behavior. The findings of Chapter 5 revealed that the current legacy application is said to experience approximately 250 requests per second. As no other data or metric exists, this workload will serve as a baseline which the prototype will be exposed to over a certain period of time. As stated in Chapter 5, two types of users will use the system, i.e., business users and administrators. Since the frequency of administrator accesses is stated to be around every two to three months, the administrator use case will not be considered negligible. Due to time constraints of this thesis the JMeter test will run for 25 minutes in order to allow the system to normalize and align to the work load. Properties and elements of a JMeter test plan have been explained in Chapter 2.

In the following, a parameterized JMeter configuration is given. In order to create a parameterized JMeter test, JMeter's build-in $property$ function will be used. The $property$ function allows to inject the values in CLI mode before the test executes by using the $-J$ flag appended to the name of the declared parameter passed into the function. For example, given a property function and a variable name identifying the number of threads, the value can then be passed in CLI mode by declaring $-JNUMBER_OF_USERS = 10$. If a test is not parameterized any future change will require to start JMeter in its Graphical User Interface (GUI) mode. Using the $property$ function allows to inject the required values in CLI mode and thus eliminates the necessity to start JMeter in GUI mode.

The test plan will be the root of the test. Figure 7.2 shows the test plan tree with one child element, i.e., the user thread group. The test plan is provided with a precise name with respect to its purpose in this work, i.e., *CHAOS_ENGINEERING_LOADTEST*. The thread group references business users that use the business logic of the application, i.e., retrieving configuration identifiers from the database in order to retrieve configuration objects. In order to behave like real users the thread group is further configured by thread properties, scheduler configuration, HTTP samplers and throughput timers.

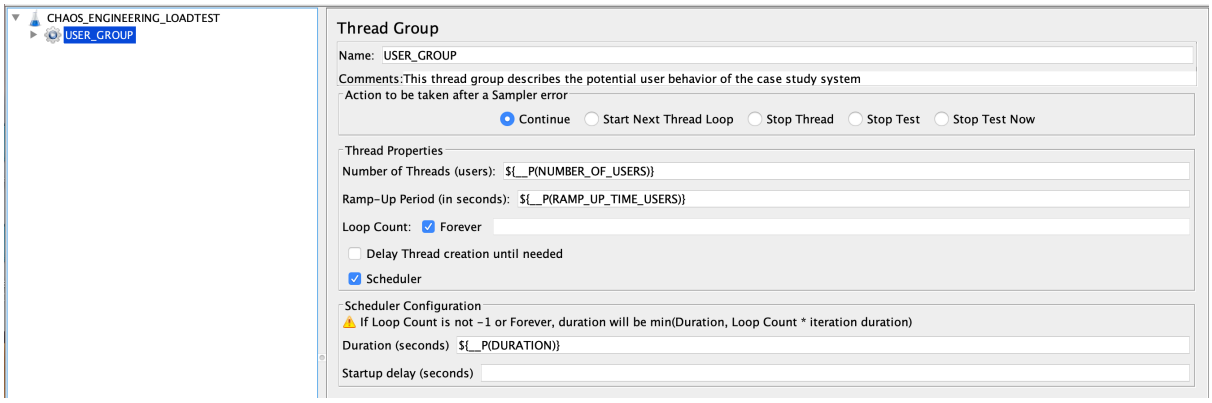

Figure 7.2.: JMeter test plan

The thread properties and other values relevant to the test plan can be injected in CLI mode by declaring a $property$ function inside the corresponding property fields. JMeter allows two declaration types for using $property$ functions which can be used interchangeably. In this case, the first declaration type is going to be used throughout the test plan to keep it consistent.

1. $\${__P(NUMBER_OF_USERS)}$
2. $\${__property(NUMBER_OF_USERS)}$

Next, an HTTP sampler will be added to both thread groups. Figure 7.3 shows the configuration for the HTTP sampler of the user thread group. The HTTP requests will receive a path to the requested service and the data body as a parameter in CLI mode. The HTTP methods will be declared in GUI mode. In addition to a basic HTTP sampler it is possible to enable an HTTP Request Default Header. The default header allows to declare properties that are true for every HTTP request in the selected thread group. Hence, within an HTTP Request Default Header a base Unified Resource Locator (URL) can be defined that will be used by every HTTP request sent in the corresponding thread group. For example if the base URL is www.simple-store.de/ then any subsequent HTTP request within the same thread group will use www.simple-store.de/ as their base URL. Note that if an HTTP Request Default Header with a base URL is declared than subsequent HTTP Request samplers will not need to declare a server name or IP, though it is possible to provide additional paths to resources.

Finally, a Constant Throughput Timer is added to the user thread group. As stated in Section 2.4.4 the Constant Throughput Timer is responsible for maintaining a constant throughput. For example if it is required that a thread group generates 50 requests per second, then the target throughput would be calculated as $RequiredThroughput * 60 = 50 * 60 = 300$. In this case, in order to maintain a throughput of 250 requests per second, the throughput timer is provided with a value of $250 * 60 = 15000$.

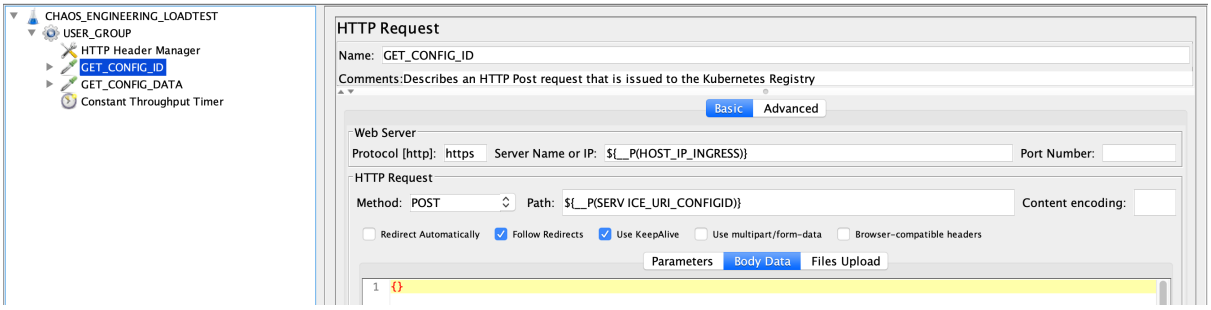

Figure 7.3.: JMeter user thread group HTTP request configuration

No.	Property name	Value
1	$\{__P(NUMBER_OF_USERS)\}$	300
2	$\{__P(RAMP_UP_TIME_USERS)\}$	120
3	$\{__P(THROUGHPUT_USERS)\}$	15000
4	$\{__P(CONTENT_TYPE)\}$	Content-type:
5	$\{__P(CONTENT_VALUE)\}$	application/json
6	$\{__P(HOST_IP_INGRESS)\}$	public IP of AKS
7	$\{__P(SERVICE_URI_CONFIGID)\}$	/registry/ID
8	$\{__P(SERVICE_URI_CONFIGDATA)\}$	/entity/getData/ID
9	$\{__P(DURATION)\}$	1500

Table 7.1.: User thread group related property functions

Table 7.1 shows all declared $property$ functions with respect to their corresponding properties that are used within the test plan. The first row of Table 7.1 shows the declared *property* function that defines the number of threads the user thread group. Row two declares the ramp up time. Row three specifies the throughput for the user thread group. Row four and five declare the *property* that specifies the content-type and content-value of each HTTP request, e.g., Content-type: application/json. $\{HOST_IP_INGRESS\}$ represents the public IP of the Kubernetes cluster, i.e., the ingress controller that routes external requests to internal services of the AKS. The rows seven and eight represent the URIs of the corresponding services that receive the user requests. The last row represents the duration of the entire JMeter test.

Chapter 8

Chaos Experiments Design, Execution, and Analysis

Chaos engineering experiments require thoughtful and careful planning with respect to selecting appropriate targets and minimizing the blast radius as stated in Section 2.6. The targets have already been selected in Chapter 6 by identifying potential risks in the discussed prototype's architecture given in Chapter 5. Chapter 7 provided a detailed description of the experiment environment. This chapter will follow to complete the experiment report by providing a description of the steady-state and a detailed explanation of the experiments that have been derived from the findings of Chapter 6. Additionally the experiments will be executed and the findings will be discussed.

This chapter is structured as follows. Section 8.1 describes the steady-state and how it was measured. Section 8.2 describes the experiment layout used throughout this chapter. Section 8.3 contains a set of hypothesis in addition to corresponding experiments. Section 8.4 discusses the findings of the experiments. Section 8.5 elaborates possible threats to the validity of this work.

8.1. Steady State Description

Since the prototype system does not provide the necessary business logic to establish any business metrics the response times measured by JMeter were used as a technical metric to reflect the health of the system.

For the purposes of this work, JMeter was configured to store response times, timestamp and label of each HTTP request executed during each run, i.e., GET_CONFIG_ID and GET_CONFIG_DATA as seen in Figure 7.3. The steady-state ideally reflects the

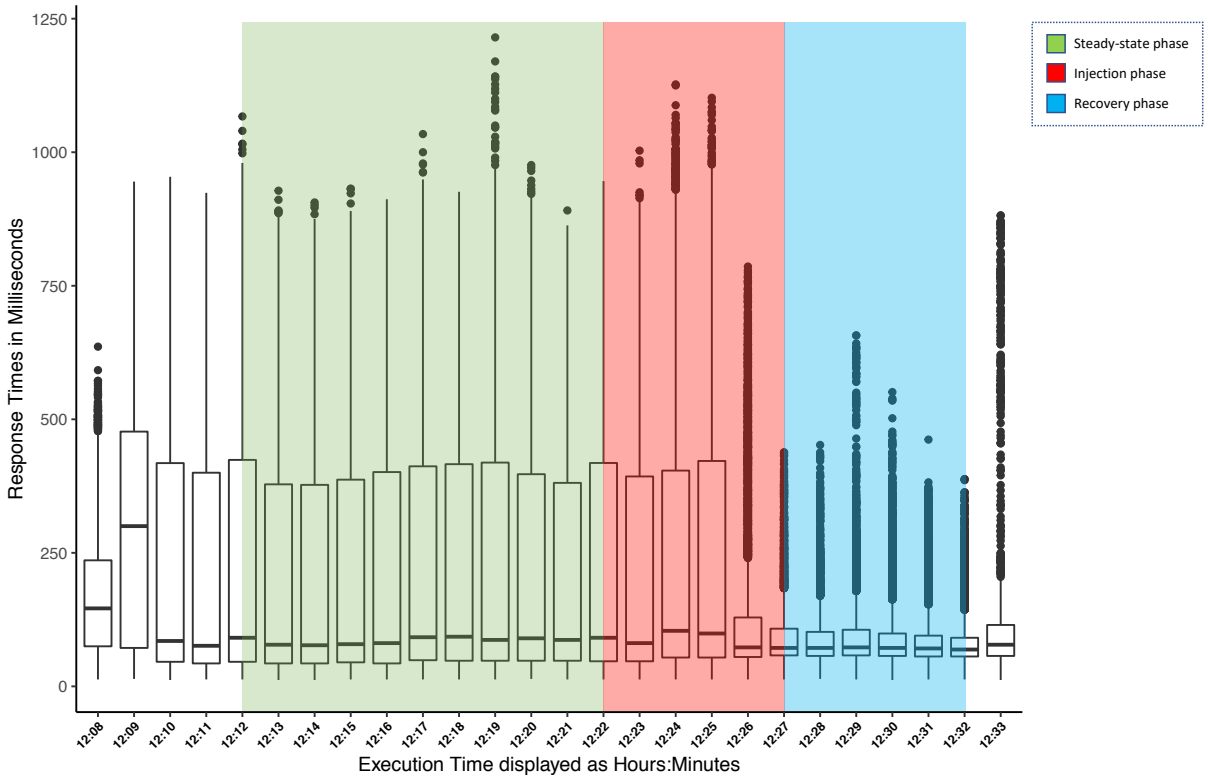

Figure 8.1.: Response times for warmup, steady-state, injection and recovery phase with respect to the GET_CONFIG_ID operation

health of the system by the response times evening out at a certain level at a given point in time, i.e., the system will eventually experience static workload. In order to know when the system will be in a steady-state the duration of a JMeter test run has been varied with the values 300 seconds, 600 seconds, 900 seconds, 1200 seconds and 1500 seconds. The response times were observed to even out after four to five minutes with a given ramp-up time of two minutes. In order to gather more observations, the final duration of the JMeter test was set to 1500 seconds, i.e., 25 minutes. With respect to the ramp-up time, the first two to four minutes were not considered for describing the steady-state. Also because JMeter will gradually decrease the throughput at the end of the test, the last minute was not considered during the chaos experiments.

Figure 8.1 shows a boxplot diagram of the first experiment run. The y-axis shows the measured response times in milliseconds. The x-axis shows the local wallclock time of execution displayed as $Hours : Minutes$. The diagram is divided into four different phases highlighted in white, green, red, and blue from left to right to illustrate the distributions that will be observed and compared. The green area highlights the time interval where the system is believed to be in a steady-state. The red area shows the time interval where the chaos experiment will be started. Usually a chaos experiment

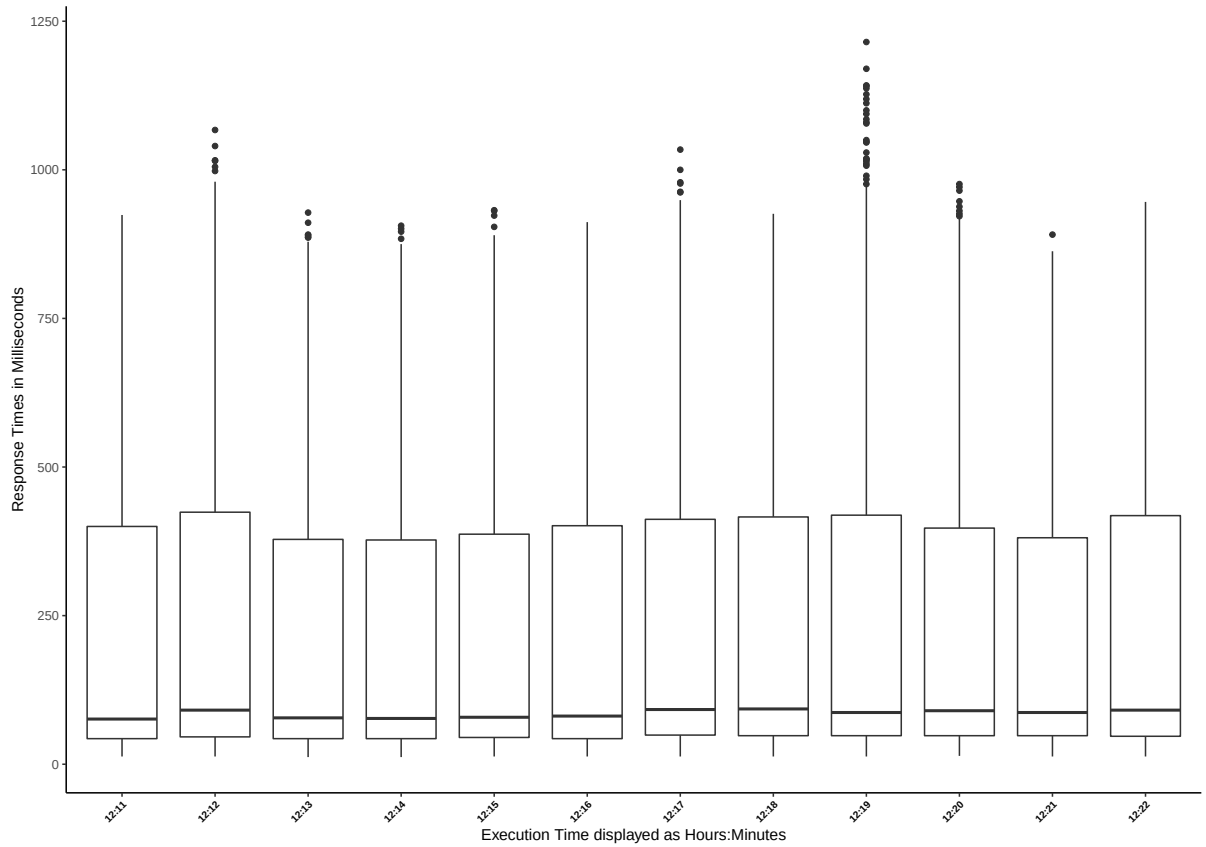

Figure 8.2.: Steady-state phase of the first experiment

executed in Chaostoolkit takes eight to 20 seconds to execute. Therefore, in order to observe possible changes in the system, a time interval of five to six minutes after the experiment has been started will be observed. This is called the injection phase. The blue area illustrates the phase where the system is expected to recover from the injection. A more precise illustration of the steady-state phase is given in Figure 8.2 is shown in relation to Figure 8.1.

After the experiments have been executed the distributions highlighted in the coloured areas will be compared by applying a statistical non-parametric test. Since JMeter runs for 1500 seconds the chaos experiment will be started after 900 seconds. This allows the system to experience its steady-state before the chaos experiment is started.

Target Service	
Experiment Type	
Hypothesis	
Blast Radius	
Status Before	
Status After	
Start Time	
End Time	
Findings	

Table 8.1.: Chaos experiment template

8.2. Experiment Description

Describing chaos experiments properly is essential in order to provide a detailed documentation and allow a thorough analysis of the findings. Table 8.1 shows a table that is used as a layout throughout this work for describing chaos engineering experiments. Note that this is not a fixed layout. Chaos engineering experiments can be described and documented in any way the researchers and engineers consider good and precise enough. The layout focuses to capture the most important properties about a chaos engineering experiment.

- Target Service – Which service is going to be attacked/observed?
- Experiment Type – What type of chaos/failure is going to be injected into the target?
- Hypothesis – Hypothesis about the steady-state that is assumed to hold
- Blast Radius – Potential system-wide affected services of the experiment
- Status Before – State of the system before the experiment is executed
- Status After – State of the system after the experiment has been executed
- Start Time – Start time of the experiment
- End Time – End time of the experiment
- Findings – Potential weaknesses that may have been uncovered

At the beginning, the *Target Service* or services are identified. In this case, this has been done in Chapter 6. Next, the *Experiment Type* is declared. The experiment type denotes the type of failure that is going to be injected into the target services. Then, the *Hypothesis* is stated in relation to the system's steady-state. The *Blast Radius* reflects potential system-wide effects the experiment will have and should therefore be considered carefully in order to avoid unexpected cascading failures. The *Status Before* illustrates a concrete state of the system before the experiment is run. The status is directly related to the steady-state and can be stated as an expected HTTP status code or expected response time interval. Lastly the *Start Time* of the experiment should be documented as precise as possible to allow a higher quality of analysis.

The fields *Status After*, *End Time*, and *Findings* are documented after the experiment has been executed. The first experiment will show the JSON structure of the experiment in Chaostoolkit. However further experiments will not show the detailed JSON structure. The experiments will be provided as supplementary material.

8.3. Hypothesis Backlog and Experiment Execution

In the following, the chaos experiments identified in Chapter 6 will be documented according to Table 8.1 and executed in the environment discussed in Section 7.1. After the experiments have been executed, the results are tested by applying a non-parametric test. It has been decided to use the Kolmogorov-Smirnov test. The decision has been made by following Field's et al. decision matrix [FH02, p. 274,278]. The decision matrix leads to the conclusion that the Wilcoxon test should be used. However, since the Wilxocon test does not compare the shape of the distributions, the Kolmogorov-Smirnov test is used as a suitable alternative as suggested by Field et al. The Kolmogorov-Smirnov test receives two distributions as input and calculates two statistics, i.e., a D- and a $p-$value. The $D-$value lies between 0 and 1 indicating the difference in the distributions shapes. If the $D-$value is closer to 0, the difference in the shapes is considered to be not statistically significant, i.e., the shapes of the distributions is considered to be equal. However, if the $D-$value is closer to 1, the difference in the shapes is considered to be statistically significant. In this case, a higher difference in the distributions shapes indicates a reasonable impact on the system's state by executing a chaos engineering experiment.

The diagrams and calculations have been performed by using the R programming language [R F]. The corresponding scripts will be provided as supplementary material. The data was displayed as boxplots. Boxplots show a summary of five statistics that can directly be read from the diagramm. The whiskers at both ends are commonly set to a length of 1.5 times or 3 times the interquartile length. Though the whiskers end

Target Service	Configuration Microservice with ID = 1
Experiment Type	Terminate Kubernetes pod
Hypothesis	The application is able to respond providing the requested configuration data but the response times will increase
Blast Radius	Configuration Microservice with ID = 1
Status Before	List_of_ID = [ID = 1]
Status After	List_of_ID = [ID = 1]
Start Time	2019-08-04 12:24:16
End Time	2019-08-04 12:24:29
Findings	Increase in response times but the requested data was received as expected

Table 8.2.: Experiment details of experiment no. 1

at the last value that still lies within this range, which results in different lengths of the whiskers. If no value lies outside of the interquartile length, the whiskers' length depends on the minimum and maximum value of the data. Furthermore, a boxplots shows the 25% quantile, the median and the 75% quantile. The median is shown by the horizontal line inside the boxes. The 25% quantile is the bottom line of each box whereas the the 75% quantile is the top line of each box. The box itself illustrates the interquartile range whereas the black dots are considered to be outliers.

The Kolmogorov-Smirnov test will receive the steady-state distribution and the injection phase distribution at first. This comparison may show a deviation in the steady-state after a chaos experiment has been started. Secondly, the test will receive the steady-state distribution and the recovery phase distribution as input. Thus, revealing if the system was able to recover from the chaos experiment. In regards of the hypothesis tests, a significance level $\alpha = 0.05$ is assumed. The null-hypothesis is the negation of the steady-state hypothesis stated in the corresponding chaos experiment description. If the calculated $p - value$ is lower than the significance level α, the null-hypothesis will be rejected in favor of the steady-state hypothesis.

Listing 8.1 Steady-state-hypothesis declaration according to Table 8.2

```
   {
       "steady-state-hypothesis": {
           "title": "The services are ready, healthy and at least the registry responds",
           "probes": [
           {
               "name": "application-responds",
               "type": "probe",
               "tolerance": {
                   "type": "jsonpath",
                   "path": "$.dataPools.*[?(@.dataVersion)].dataVersion",
                   "expect": [1],
                   "target": "body"
               },
               "provider": {
                   "type": "http",
                   "url": "https://104.46.14.61/vccrest/registry/getDataPools",
                   "method": "POST",
                   "verify_tls": false,
                   "headers": {
                       "Content-Type": "application/json"
                   },
                   "arguments": {
                       "body": {}
                   }
               }
           },
           {
               "name": "configuration-service-responds",
               "type": "probe",
               "tolerance": {
                   "type": "jsonpath",
                   "path": "$.configurationContext.dataPool.dataVersion",
                   "expect": [1],
                   "target": "body"
               },
               "provider": {
                   "type": "http",
                   "url": "https://104.46.14.61/vccrest/entity/getConfigurationContext/1",
                   "method": "POST",
                   "verify_tls": false,
                   "headers": {
                       "Content-Type": "application/json"
                   },
                   "arguments": {
                       "body": {}
                   }
               }
           }
           ]
       }
},
```

Listing 8.2 Method and rollback declaration according to Table 8.2

```
    "method": [{
            "type": "action",
            "name": "terminate-vcc-service-n-pod",
            "provider": {
                "type": "python",
                "module": "chaosk8s.pod.actions",
                "func": "terminate_pods",
                "arguments": {
                    "label_selector": "app=vcc-service-1",
                    "ns": "default",
                    "mode": "fixed",
                    "qty": 1
                }
            },
            "pauses": {
                "after": 10
            }
        },
    {
        "ref": "application-responds"
    }
    ],
    "rollbacks": []
}
```

8.3.1. Experiment No. 1: Terminating one configuration microservice by terminating its Kubernetes pod

Design

This experiment has been extracted from Figure 6.2. The Kubernetes pod that hosts a configuration microservice is going to be terminated by using Chaostoolkit's Kubernetes library. The pod is identified by its label which can be retrieved by using the Kubernetes API, i.e., $kubectl\ describe\ pod\ service - 1$. The experiment information can be seen in Table 8.2.

To reduce the blast radius of experiments it has been decided to not create more than one configuration microservice of a corresponding identifier if only one type of configuration microservice is targeted, e.g., as in Table 8.2. This ensures that the blast radius will not unintentionally be bigger than expected since it is not possible to affect multiple configuration microservices. However, it is still possible that other services might be affected. In subsequent experiments the blast radius will slowly be increased by adding additional configuration microservices which is done manually by creating additional configuration objects in the database through the administrator API. The configuration objects are created by sending an HTTP request to the administrator service that implements a controller for randomly generating configuration data.

The JSON declaration of Table 8.2 is illustrated in Listing 8.1 and Listing 8.2. Listing 8.1 shows the steady-state hypothesis. The hypothesis is stated to be true if the registry service returns a status code 200 followed by a JSON containing a list of identifiers. Since only one configuration object is persisted in the database, it is expected that the list will only contain one identifier.

Listing 8.2 shows the residual experiment containing the method and rollback declaration. The method block contains the Kubernetes library command to look for a single or multiple pods with a specific label, i.e., declared by the $label_selector$ attribute. Since each configuration microservice with respect to an identifier is deployed twice, two pods will be found with the same label. The $mode$ and $quantity$ attributes ensure that only one pod is selected. Lastly, there will be no rollbacks be declared since the state is expected to be restored by Kubernetes automatically. Note that if no rollbacks are declared the attribute can be removed completely since it is stated as being optional.

Results

After the experiment has been executed, the boxplot diagrams Figure 8.3 and Figure 8.4 were created based on the data collected by JMeter. Figure 8.3 shows the response times of retrieving the list of configuration identifiers from the registry whereas Figure 8.4 shows the response times of retrieving the configuration data objects from the configuration microservices. The corresponding operations are labeled as GET_CONFIG_ID and GET_CONFIG_DATA in the JMeter configuration seen in Figure 7.2.

Visual analysis of the diagrams shows, that at the time the experiment has been executed, i.e., 15 minutes after the JMeter has been started, an increase of response times occurs. The increase is more noticeable in Figure 8.4 than in Figure 8.3. A more precise analysis is given by applying the Kolmogorov-Smirnov test.

The test results for Figure 8.3 are shown in Listing 8.3, i.e., the GET_CONFIG_ID operation. However the test results for Figure 8.4 are shown in Listing 8.4, i.e., the GET_CONFIG_DATA operation. Furthermore, Table 8.3 and Table 8.4 show a summary of standard statistics of the corresponding operations with respect to the observed phases. The summary comprises of the mean, median, standard deviation, 25% quantile, 75% quantile, and 95% quantile. The values of the standard statistics are also reflected in the boxplots.

Figure 8.5 and Figure 8.6 show the respective Empirical Cumulative Distribution Function (ECDF) of both operations. The difference of the range of response times of both operations is evident. However, it is shown that approximately 80% of the response times of the first operation, i.e., GET_CONFIG_ID, are higher than 400 milliseconds. On the other side, between 80% and 100% of the response times of the second operation,

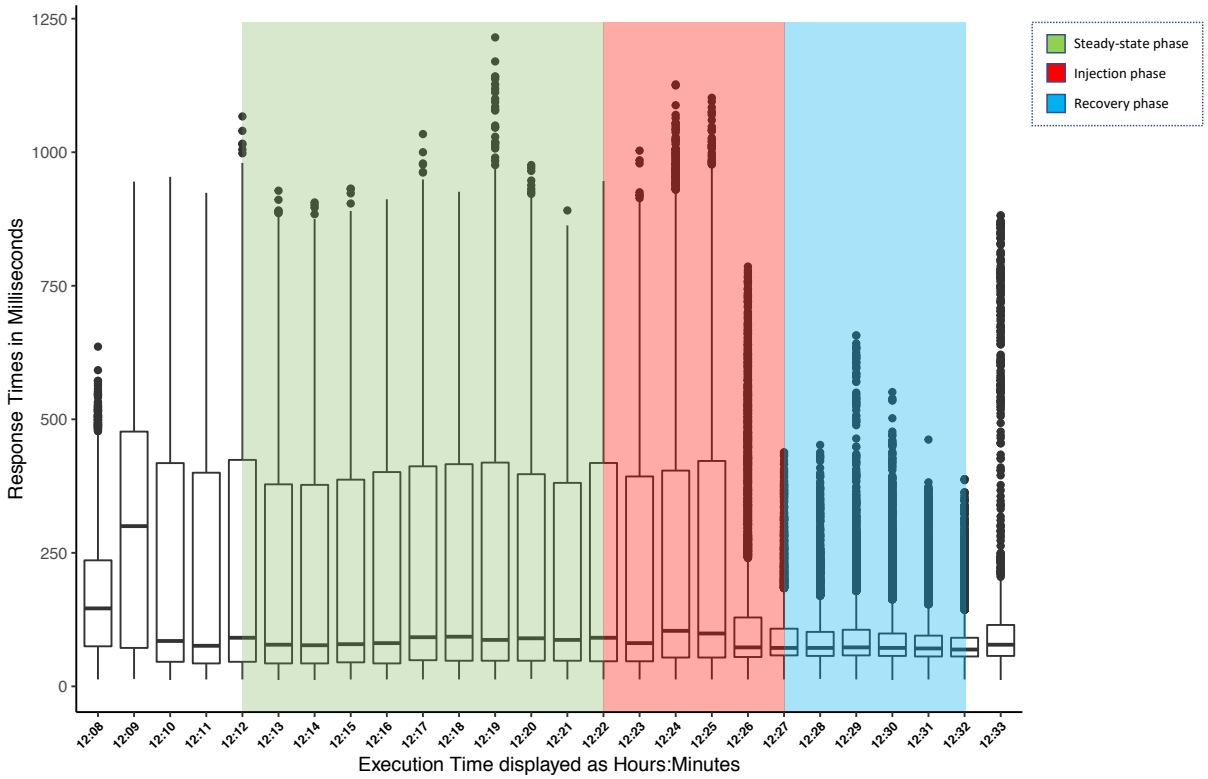

Figure 8.3.: Response times of GET_CONFIG_ID operation after first experiment

Steady-State Phase					
Mean	Median	Standard Deviation	25% Quantile	75% Quantile	95% Quantile
221.6	85.0	228.6701	46.0	401.0	686
Injection Phase					
Mean	Median	Standard Deviation	25% Quantile	75% Quantile	95% Quantile
164.3	77.0	186.4005	55.0	174.0	588
Recovery Phase					
Mean	Median	Standard Deviation	25% Quantile	75% Quantile	95% Quantile
87.76	71.00	56.5072	57.00	97.00	189

Table 8.3.: Standard statistics with respect to the observed phases of GET_CONFIG_ID operation, illustrated in Figure 8.3

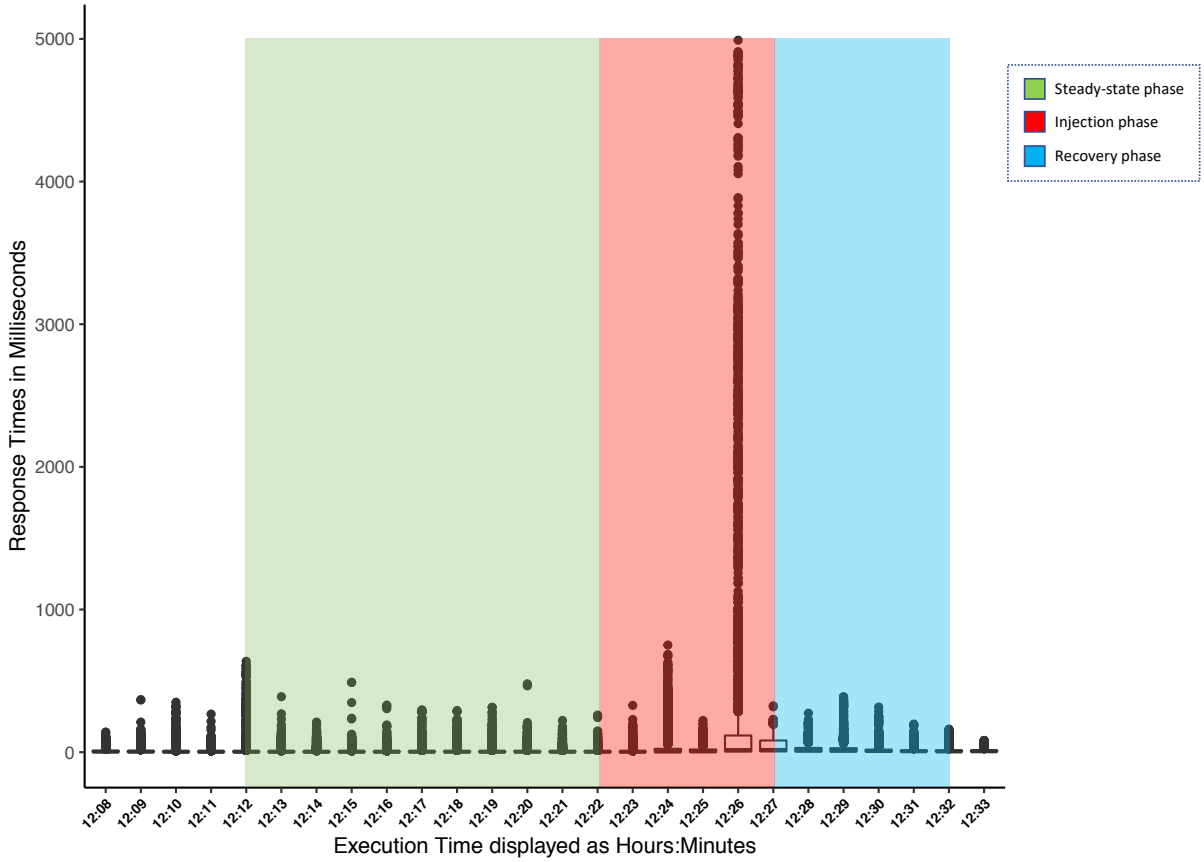

Figure 8.4.: Response times of GET_CONFIG_DATA operation after first experiment

Steady-State Phase					
Mean	Median	Standard Deviation	25% Quantile	75% Quantile	95% Quantile
7.375	3.000	21.93762	2.000	6.000	25
Injection Phase					
Mean	Median	Standard Deviation	25% Quantile	75% Quantile	95% Quantile
64.97	8.00	306.8682	3.00	27.00	146
Recovery Phase					
Mean	Median	Standard Deviation	25% Quantile	75% Quantile	95% Quantile
17.31	8.00	28.73427	4.00	15.00	90

Table 8.4.: Standard statistics with respect to the observed phases of GET_CONFIG_DATA operation, illustrated in Figure 8.4

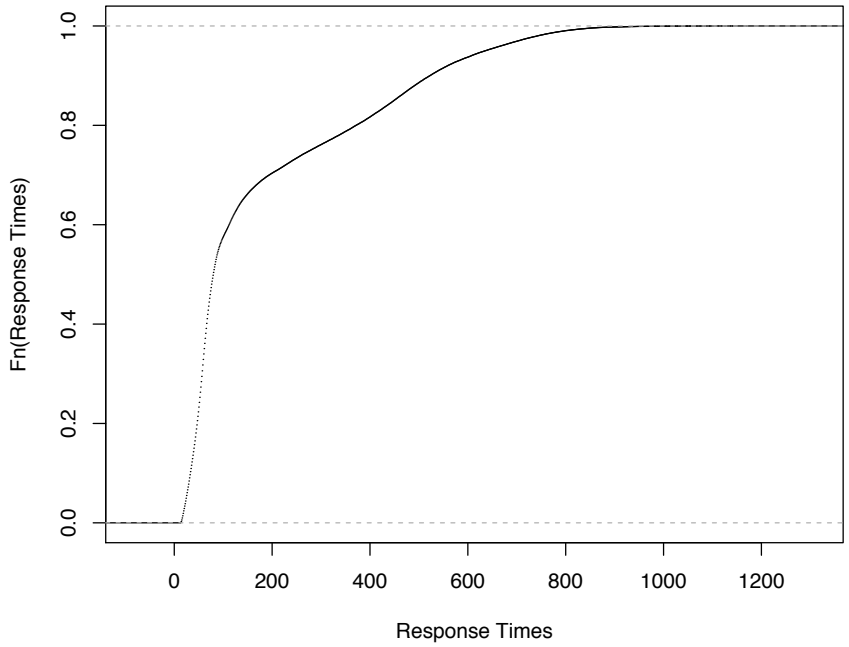

Figure 8.5.: Empirical Cumulative Distribution Function (ECDF) of GET_CONFIG_ID operation after experiment no. 1

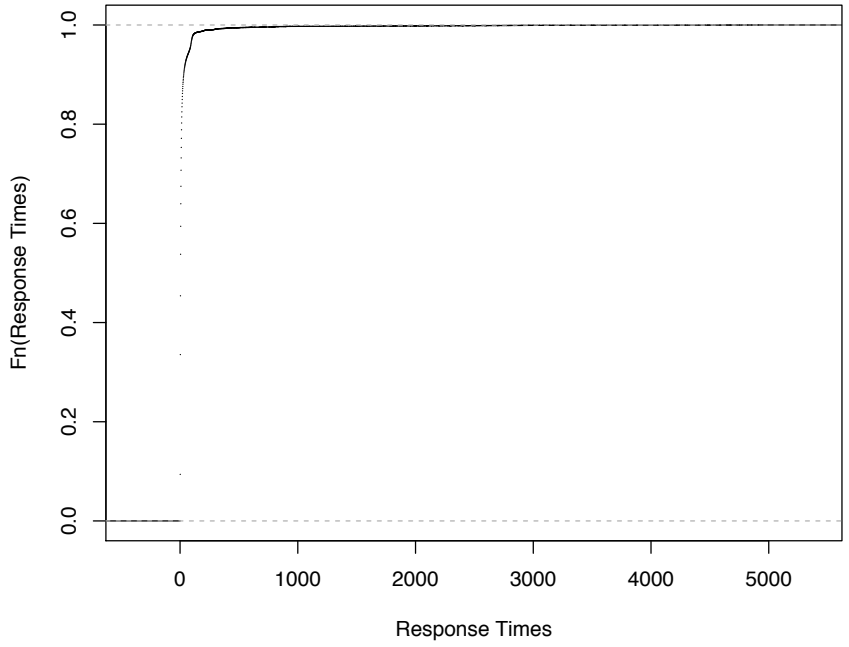

Figure 8.6.: Empirical Cumulative Distribution Function (ECDF) of GET_CONFIG_DATA operation after experiment no. 1

Listing 8.3 Kolmogorov-Smirnov test results related experiment no. 1 with respect to steady-state and injection phase of Figure 8.3

```
  Two-sample Kolmogorov-Smirnov test

data: steady_state_distribution_get_id_experiment_one and
    injection_distribution_get_id_experiment_one
D = 0.15062, p-value < 2.2e-16
alternative hypothesis: two-sided
```

Listing 8.4 Kolmogorov-Smirnov test results related experiment no. 1 with respect to steady-state and injection phase of Figure 8.4

```
      Two-sample Kolmogorov-Smirnov test

data: steady_state_distribution_get_data_experiment_one and
    injection_distribution_get_data_experiment_one
D = 0.33061, p-value < 2.2e-16
alternative hypothesis: two-sided
```

i.e., GET_CONFIG_DATA are higher than 1000 milliseconds. A better comparison is shown by applying the Kolmogorov-Smirnov test.

The results of Listing 8.3 show a calculated D-value of 0.15062 and a p-value of $2.2 * e^{-16}$ for the distributions of Figure 8.3, i.e., GET_CONFIG_ID operation. The calculated D-value of 0.15062 indicates that the shapes of the steady-state and injection phase distributions are insignificant since the D-value is close to 0. The D-value of the GET_CONFIG_ID operation, shown in Listing 8.4, is almost three times higher with a value of 0.33061 in comparison to the GET_CONFIG_DATA operation. Though the D-value is three times higher it is still closer to 0 than to 1, thus, implying that the shapes of the distributions are not significantly different.

The p-values of both operations are lower than the significance level of $\alpha = 0.05$. Thus, implying that the null-hypothesis should be rejected in favor of the alternative-hypothesis stated in the chaos experiment description of Table 8.2. This further indicates, that the steady-state-hypothesis of the first experiment is met but with an insignificant increase in response times as indicated by the D-values.

The comparison of the recovery distributions of both operations, shown in Listing 8.5 and Listing 8.6, indicate that the shapes are slightly different based on higher D-values. The D-value corresponding to the GET_CONFIG_ID operation is calculated to be 0.33924, which is slightly lower than the D-value of the GET_CONFIG_DATA operation which is 0.40061. Considering possible deviations in response times because of latency, the system can be assumed to have returned into a steady-state after injection. In the case

Listing 8.5 Kolmogorov-Smirnov test results related experiment no. 1 with respect to recovery phase of Figure 8.3

```
      Two-sample Kolmogorov-Smirnov test

data: steady_state_distribution_get_id_experiment_one and
    recovery_distribution_get_id_experiment_one
D = 0.33924, p-value < 2.2e-16
alternative hypothesis: two-sided
```

Listing 8.6 Kolmogorov-Smirnov test results related experiment no. 1 with respect to recovery phase of Figure 8.4

```
      Two-sample Kolmogorov-Smirnov test

data: steady_state_distribution_get_data_experiment_one and
    recovery_distribution_get_data_experiment_one
D = 0.40061, p-value < 2.2e-16
alternative hypothesis: two-sided
```

of the recovery phases, the p-values are not discussed since the steady-state-hypothesis only concerns about the comparison between the steady-state and injection phase.

8.3.2. Experiment No. 2: Terminating two configuration microservices with the same identifier by terminating their Kubernetes pods

Design

This experiment has been extracted from Figure 6.2. In the previous experiment only one pod was terminated. This time the blast radius is increased by targeting both pods of the same configuration microservice. The steady-state-hypothesis is stated as in the first experiment. Though, it is expected that the response times increase significantly more than in the first experiment since no configuration microservice will be available. Hence, before the experiment is executed the response of the registry comprises a list of a single identifier. After the experiment will be started, it is expected that the data will not be received, followed by an increase in response times. The experiment information can be seen in Table 8.5.

Target Service	Two Configuration Microservices with ID = 1
Experiment Type	Terminate Kubernetes pods
Hypothesis	The application should be able to respond with the requested data and the response times should increase significantly because of missing services
Blast Radius	Two Configuration Microservices with ID = 1
Status Before	List_of_ID = [ID = 1]
Status After	List_of_DATA = [] and List_of_ID = [ID = 1]
Start Time	2019-08-04 12:54:33
End Time	2019-08-04 12:54:39
Findings	The experiment failed because the steady-state hypothesis has not been met after injection. The response times increased significantly.

Table 8.5.: Experiment details of experiment no. 2

Results

The measured response times from experiment no. 2 can be seen in Figure 8.7 and Figure 8.8, again referencing two operations, i.e., GET_CONFIG_ID and GET_CONFIG_DATA. Note that the y-axis has been limited to allow a better visual analysis. The limitation of the y-axis results in missing outliers that are not critical to the analysis.

Inspecting both diagrams shows that there is a significant increase in response times after the experiment has been started. Since both configuration microservices are not available but users keep requesting the services, the response times increase to a large extent. However, it is also shown that after a given amount of time the response times decrease and the system tries to recover from the injection by slowly getting back into a steady-state. Furthermore, it is noticeable that the medians in the recovery phase of both diagrams are slightly shifted compared to the medians of the steady-state phase.

Observing the steady-state distribution and the injection phase distribution with the Kolmogorov-Smirnov test of the GET_CONFIG_ID operation, provides the results in 8.7. The p-value is equal to Listing 8.3 whereas the D-values is higher with a value of 0.38137. The difference in shapes is significantly higher than in the previous test

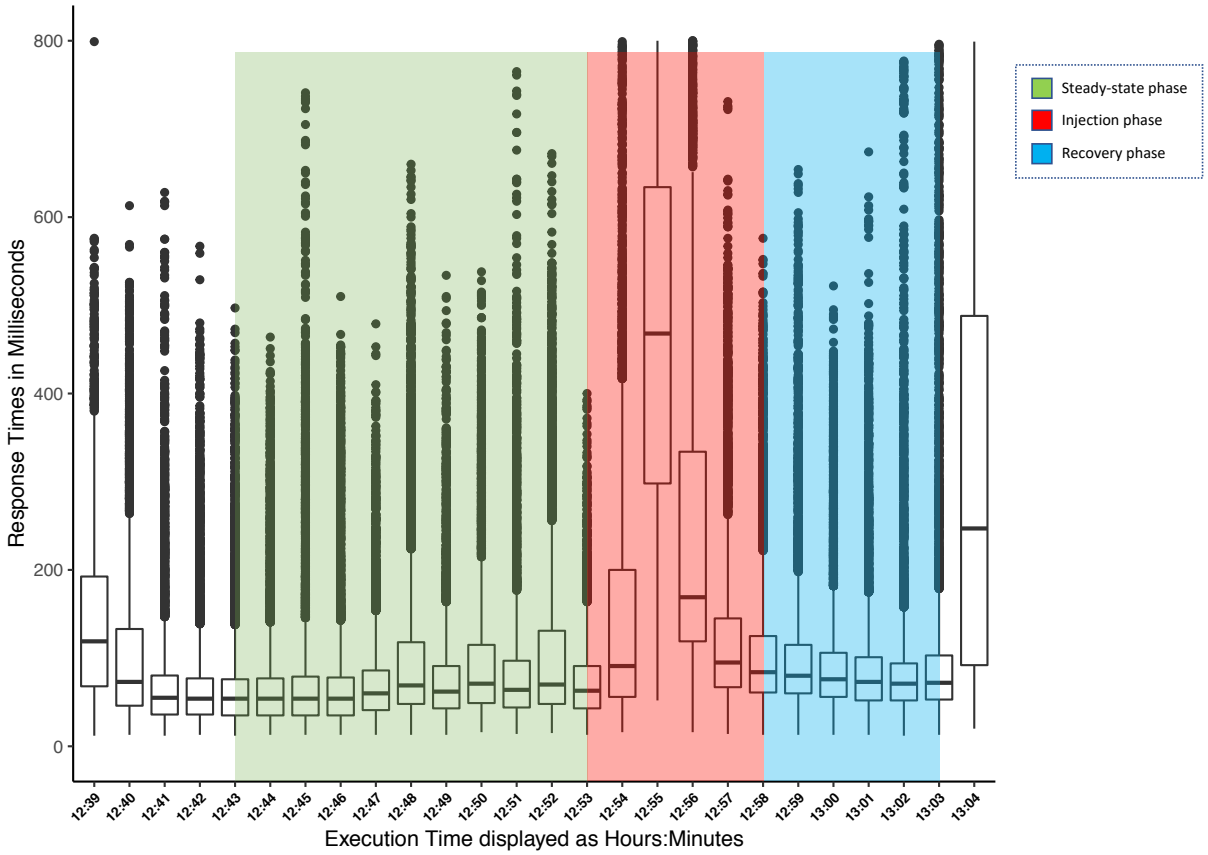

Figure 8.7.: Response times of of GET_CONFIG_ID after second experiment

Steady-State Phase					
Mean	Median	Standard Deviation	25% Quantile	75% Quantile	95% Quantile
221.6	85.0	228.6701	46.0	401.0	686
Injection Phase					
Mean	Median	Standard Deviation	25% Quantile	75% Quantile	95% Quantile
164.3	77.0	186.4005	55.0	174.0	588
Recovery Phase					
Mean	Median	Standard Deviation	25% Quantile	75% Quantile	95% Quantile
87.76	71.00	56.5072	57.00	97.00	189

Table 8.6.: Standard statistics with respect to the observed phases of GET_CONFIG_ID operation, illustrated in Figure 8.7

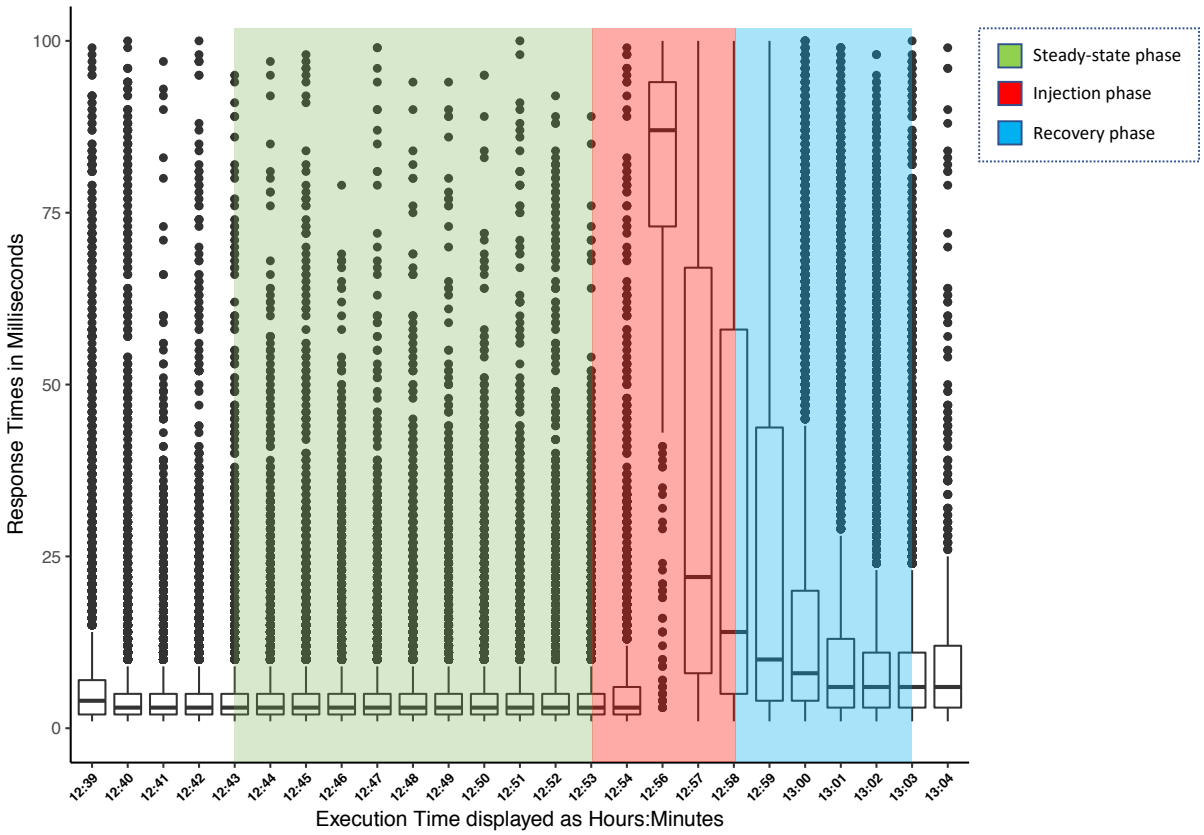

Figure 8.8.: Response times of of GET_CONFIG_DATA after second experiment

Steady-State Phase					
Mean	Median	Standard Deviation	25% Quantile	75% Quantile	95% Quantile
5.047	3.000	8.731445	2.000	5.000	15
Injection Phase					
Mean	Median	Standard Deviation	25% Quantile	75% Quantile	95% Quantile
165.6	16.0	735.6111	5.0	74.0	585
Recovery Phase					
Mean	Median	Standard Deviation	25% Quantile	75% Quantile	95% Quantile
14.7	6.0	24.03997	3.0	13.0	66

Table 8.7.: Standard statistics with respect to the observed phases of GET_CONFIG_DATA operation, illustrated in Figure 8.8

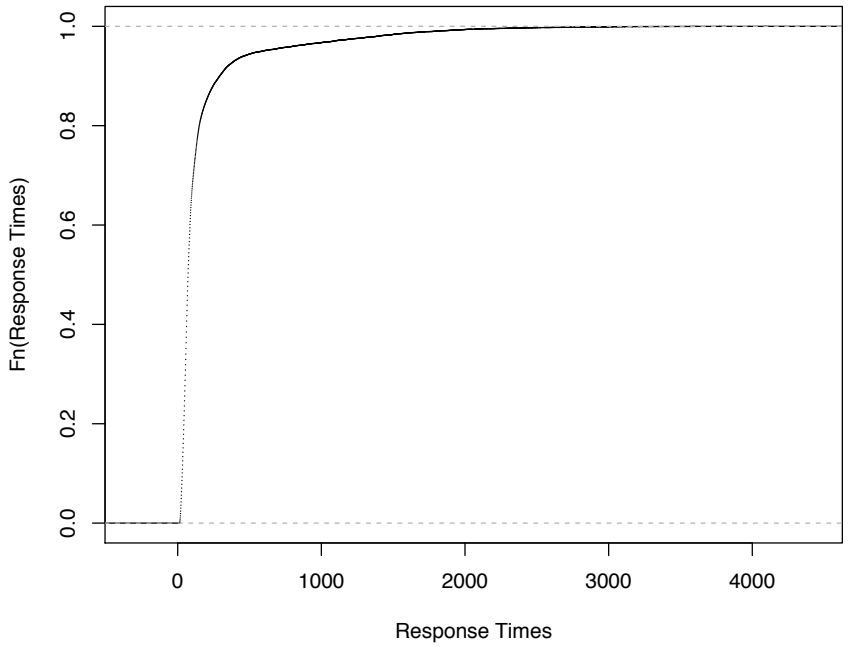

Figure 8.9.: Empirical Cumulative Distribution Function (ECDF) of GET_CONFIG_ID operation after experiment no. 2

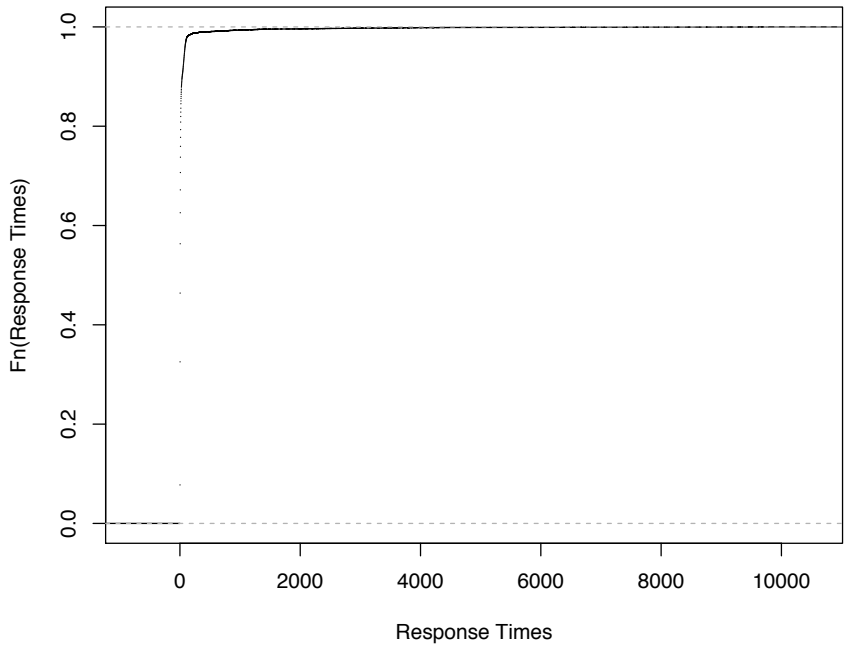

Figure 8.10.: Empirical Cumulative Distribution Function (ECDF) of GET_CONFIG_DATA operation after experiment no. 2

Listing 8.7 Kolmogorov-Smirnov test results related experiment no. 2 with respect to steady-state and injection phase of Figure 8.7

```
      Two-sample Kolmogorov-Smirnov test

data: steady_state_distribution_get_id_experiment_two and
    injection_distribution_get_id_experiment_two
D = 0.38137, p-value < 2.2e-16
alternative hypothesis: two-sided
```

Listing 8.8 Kolmogorov-Smirnov test results related experiment no. 2 with respect to steady-state and injection phase of Figure 8.8

```
      Two-sample Kolmogorov-Smirnov test

data: steady_state_distribution_get_data_experiment_two and
    injection_distribution_get_data_experiment_two
D = 0.50391, p-value < 2.2e-16
alternative hypothesis: two-sided
```

which can also be verified by inspecting the corresponding intervals in both diagrams Figure 8.3 and Figure 8.7. However, despite the higher D-value, since it is closer to 0 it is still considered not significant. Though, it is obvious that the second experiment must have had a larger impact on the system than the first experiment.

Again, Figure 8.9 and Figure 8.10 show the respective ECDFs of both operations. As given in the boxplot diagrams, the response times are fairly higher than in the first experiment, which can also be observed in the ECDF plots. However, both plots show that approximately 95% to 100% of the response times of the response times are lower than 100 milliseconds. Compared to the ECDF of the first experiment with respect to the GET_CONFIG_ID operation, a slight increase in response times is noticeable. In contrast to the second operation, i.e., GET_CONFIG_DATA the increase in response times can vaguely be observed in the ECDF plots, but it becomes more evident in the statistical test results.

The D-value of the GET_CONFIG_DATA operation, as seen in Listing 8.8, is much higher than the D-value of the first experiment. With a value of 0.50391 the D-value gets closer to 1, reflecting the expecting increase in response times due to the lack of configuration microservices. The p-values imply to reject the null hypothesis in favor of the alternative hypothesis since the $p - values$ are all calculated to be $2.2 * e^{-16}$ and thus lower than the significance level α.

Inspecting the recovery phase of the system shows that the medians of Figure 8.8 are again slightly shifted. However the medians of Figure 8.7 are not as far shifted as in the

Listing 8.9 Kolmogorov-Smirnov test results related experiment no. 2 with respect to the recovery phase of Figure 8.7

```
      Two-sample Kolmogorov-Smirnov test

data: steady_state_distribution_get_id_experiment_two and
    recovery_distribution_get_id_experiment_two
D = 0.16619, p-value < 2.2e-16
alternative hypothesis: two-sided
```

Listing 8.10 Kolmogorov-Smirnov test results related experiment no. 2 with respect to the recovery phase of Figure 8.8

```
      Two-sample Kolmogorov-Smirnov test

data: steady_state_distribution_get_data_experiment_two and
    recovery_distribution_get_data_experiment_two
D = 0.33378, p-value < 2.2e-16
alternative hypothesis: two-sided
```

first experiment. Comparing the distributions with the Kolmogorov-Smirnov test shows the results in Listing 8.9 and Listing 8.10.

As expected, the $D - value$ of Figure 8.7 with 0.16619 is double as low as the D-value of Figure 8.8 with a value of 0.33378, indicating that the system was not able to maintain a steady-state with respect to the GET_CONFIG_DATA operation. This may result from the time Kubernetes needs to create new pods with the corresponding microservices. Longer measurement could have shown, that the response times in Figure 8.8 will decrease further without a shift in the median.

8.3.3. Experiment No. 3: Terminating one configuration microservice with ID=1 while requesting configuration microservice with ID=2

Design

This experiment is done to illustrate a further increase in the blast radius by adding an additional configuration object to the database. Thus, there will be two distinct configuration microservices with two distinct identifiers initialized. This results in four pods being started by Kubernetes. Table 8.8 shows the information of this experiment. The new configuration object is created as the first one by using the administrator API.

Target Service	One Configuration Microservices with ID = 1
Experiment Type	Terminate Kubernetes pods
Hypothesis	The application should be able to respond all identifiers of all microservices and the response times should increase because of missing services
Blast Radius	One Configuration Microservice with ID = 1
Status Before	List_of_ID = [ID = 1, ID = 2]
Status After	List_of_DATA = [ID = 1 ID = 2] and List_of_ID = [ID = 1, ID = 2]
Start Time	2019-08-04 15:13:40
End Time	2019-08-04 15:13:46
Findings	The experiment succeeded. However the response times of the registry increase and did not stabilize. Additionally the database pod went into an evicted mode which will inevitably lead to the inability to create, delete or request data from the database.

Table 8.8.: Experiment details of experiment no. 3

The identifier is set to two whereas the content is created randomly. After the pods are running and ready the experiment is ready to be performed.

Before the experiment is executed, the system correctly response with a list of identifiers containing two distinct identifiers, i.e., 1 and 2. In addition, the configuration microservices to the corresponding identifiers, return the appropriate configuration data bodies.

8.3.4. Results

The response times of the JMeter run with respect to the third experiment are illustrated in Figure 8.11 and Figure 8.12. Inspecting the response times of both operations shows, that the response times of the registry spiked significantly after the experiment has been started whereas the response times of the configuration microservices did increase slightly but not to a significant extend. This is an unexpected behavior since it was expected, that the response times of the configuration microservices increase to a larger extend. However, the diagrams do not reflect the behavior of the database service.

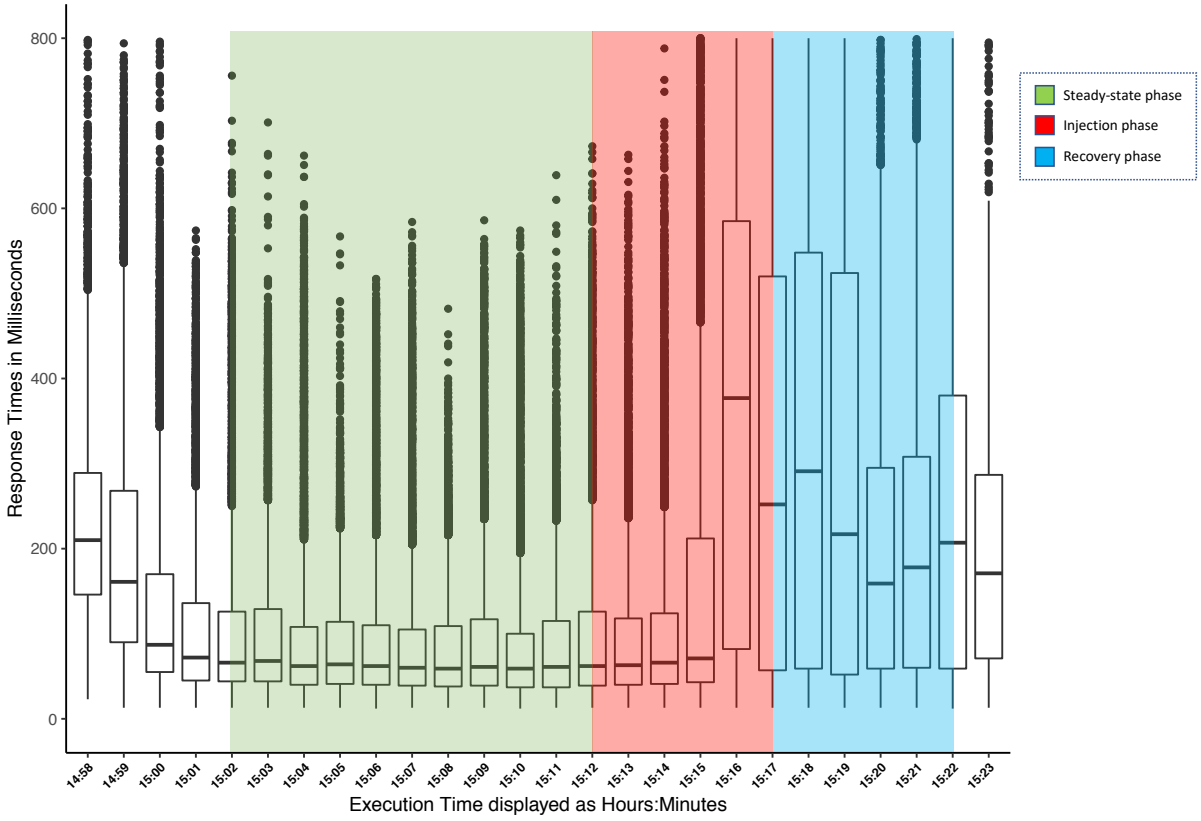

Figure 8.11.: Response times of GET_CONFIG_ID after third experiment

Steady-State Phase					
Mean	Median	Standard Deviation	25% Quantile	75% Quantile	95% Quantile
96.13	62.00	91.26923	40.00	113.00	288
Injection Phase					
Mean	Median	Standard Deviation	25% Quantile	75% Quantile	95% Quantile
287.1	106.0	344.678	49.0	452.0	974
Recovery Phase					
Mean	Median	Standard Deviation	25% Quantile	75% Quantile	95% Quantile
289.2	207.0	283.2689	61.0	422.0	899

Table 8.9.: Standard statistics with respect to the observed phases of GET_CONFIG_ID operation, illustrated in Figure 8.11

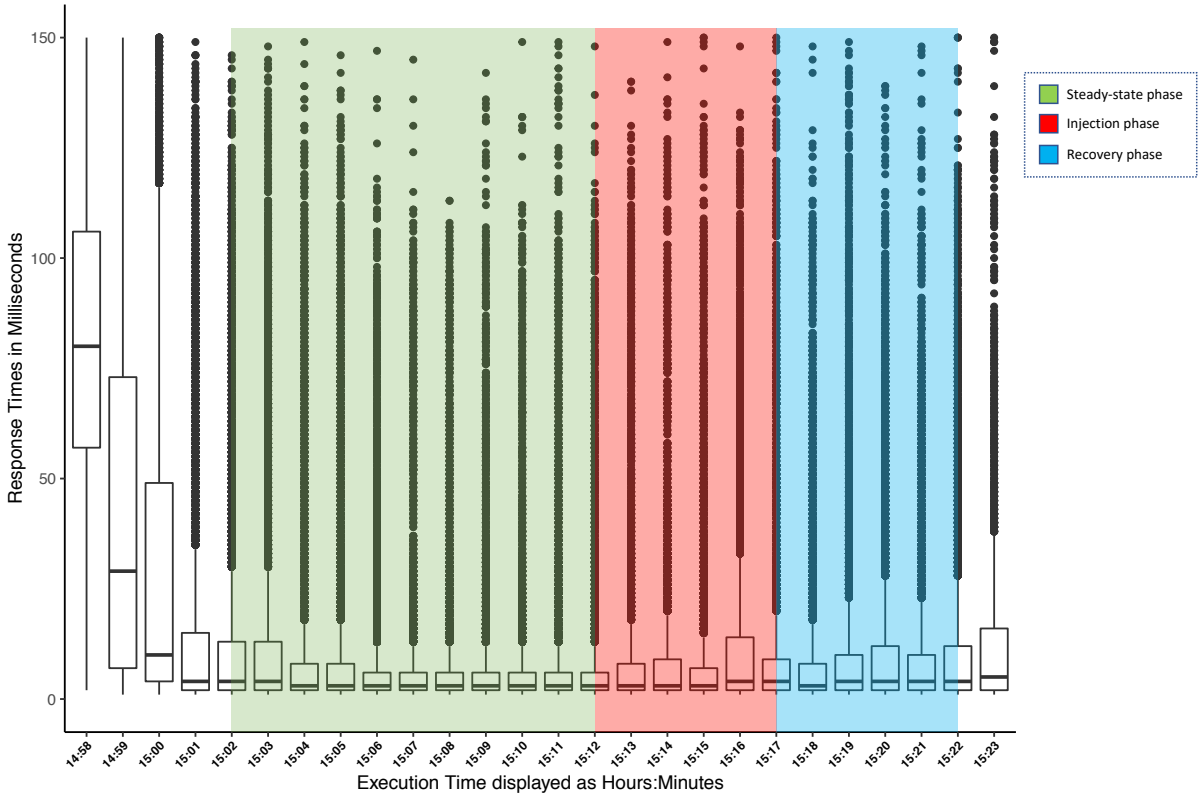

Figure 8.12.: Response times of GET_CONFIG_DATA after third experiment

Steady-State Phase					
Mean	Median	Standard Deviation	25% Quantile	75% Quantile	95% Quantile
9.85	3.00	19.28709	2.00	7.00	52
Injection Phase					
Mean	Median	Standard Deviation	25% Quantile	75% Quantile	95% Quantile
14.96	3.00	81.43952	2.00	8.00	52
Recovery Phase					
Mean	Median	Standard Deviation	25% Quantile	75% Quantile	95% Quantile
12.97	4.00	24.77287	2.00	11.00	61

Table 8.10.: Standard statistics with respect to the observed phases of GET_CONFIG_DATA operation, illustrated in Figure 8.12

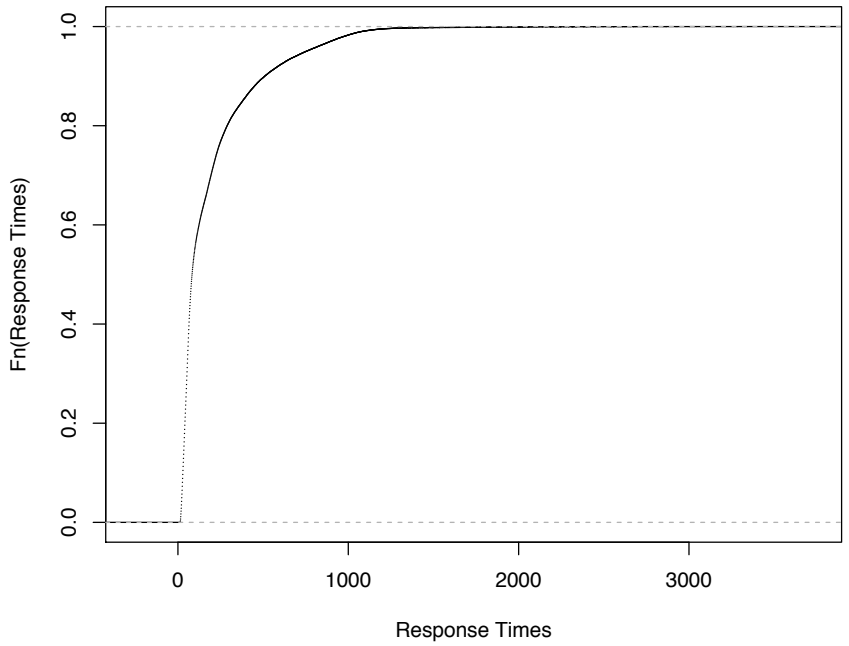

Figure 8.13.: Empirical Cumulative Distribution Function (ECDF) of GET_CONFIG_ID operation after experiment no. 3

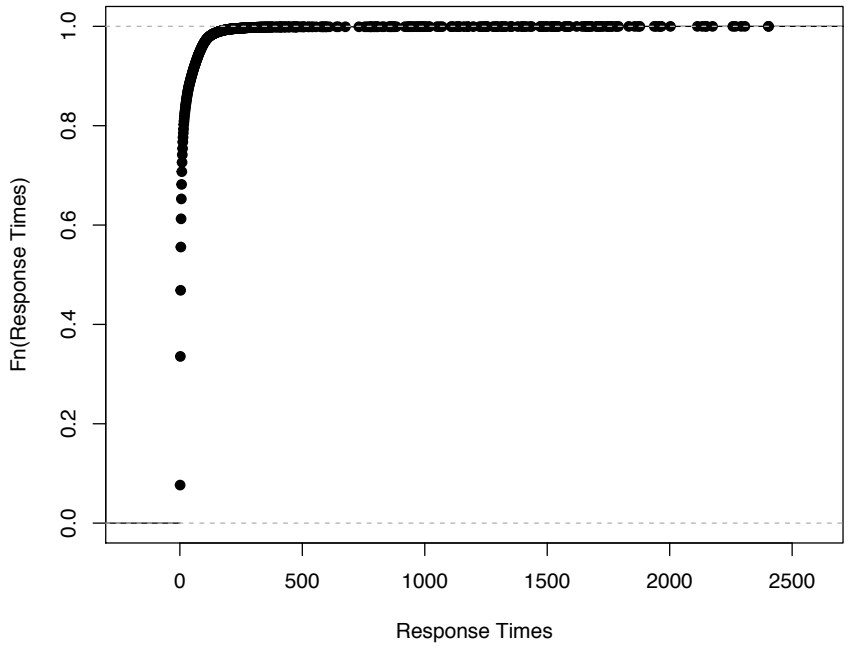

Figure 8.14.: Empirical Cumulative Distribution Function (ECDF) of GET_CONFIG_DATA operation after experiment no. 3

Before the experiment has been executed, the Kubernetes pod that hosts the database service went into an evicted phase which states that the Kubernetes node run out of memory thus terminating the pod that exceeds its memory threshold. The effects are at first not visible to the user and might not become visible. Nonetheless, if an administrator would have tried to create, delete, or request any data from the database, the request would have returned an error since the database is not available anymore. Furthermore, evicted pods are not removed from the cluster automatically. Evicted pods have to be removed manually by a person using the Kubernetes API in addition, to manually creating a new pod hosting the database.

The ECDFs of the third experiment are quite similar to the ECDFs of the second experiment. Therefore, a further analysis does not provide much information. Though, as it is observable in the boxplots, the ECDF of the first operation, i.e., Figure 8.13, shows a slightly shifted curve after the 1000 millisecond mark on the x-axis indicating the loss of the database. In contrast, the ECDF in Figure 8.14 does not show any interesting changes according to the loss of the database.

Despite the unexpected behavior of the database, Listing 8.11 shows the calculations of the Kolmogorov-Smirnov test with respect to the GET_CONFIG_ID operation. The D-value is calculated to be 0.28815 which is close to 0, thus, as in previous tests not considered to be significant. On the other side, the D-value of the corresponding GET_CONFIG_DATA operation, seen in Figure 8.12, is calculated to be 0.041004 which can be considered to be 0. Hence, not being significant.

Moreover it is evident that the system is not able to recover properly as seen in Figure 8.11 during the last minutes. Comparing the steady-state phase with the recovery phase makes the difference in the distributions more evident as seen in Listing 8.13. The D-value of 0.094853 shows, that the shape of the distributions of the GET_CONFIG_DATA operation with respect to the steady-state and recovery phase are almost equal. In contrast to the D-value of 0.39223, the shape of the distributions of the GET_CONFIG_ID operation are not equal as seen in the diagram. This means, that the chaos experiment had almost no effect on the configuration microservices but rather on the registry service which led to the system not being able to recover to its steady-state.

The p-values have not changed compared to previous experiments. Thus, still being lower than the significance level α, the null-hypothesis is again rejected in favor of the alternative-hypothesis. However, the experiment revealed an additional weakness, where the registry was somehow affected by the experiment despite being not targeted directly. Additionally, it was revealed that the database pod was terminated based on a too high memory usage.

Listing 8.11 Kolmogorov-Smirnov test results related experiment no. 3 with respect to the steady-state and injection phase of Figure 8.11

```
      Two-sample Kolmogorov-Smirnov test

data: steady_state_distribution_get_id_experiment_three and
    injection_distribution_get_id_experiment_three
D = 0.28815, p-value < 2.2e-16
alternative hypothesis: two-sided
```

Listing 8.12 Kolmogorov-Smirnov test results related experiment no. 3 with respect to the steady-state and injection phase of Figure 8.12

```
      Two-sample Kolmogorov-Smirnov test

data: steady_state_distribution_get_data_experiment_three and
    injection_distribution_get_data_experiment_three
D = 0.041004, p-value < 2.2e-16
alternative hypothesis: two-sided
```

Listing 8.13 Kolmogorov-Smirnov test results related experiment no. 3 with respect to the recovery phase of Figure 8.11

```
      Two-sample Kolmogorov-Smirnov test

data: steady_state_distribution_get_id_experiment_three and
    recovery_distribution_get_id_experiment_three
D = 0.39223, p-value < 2.2e-16
alternative hypothesis: two-sided
```

Listing 8.14 Kolmogorov-Smirnov test results related experiment no. 3 with respect to the recovery phase of Figure 8.12

```
      Two-sample Kolmogorov-Smirnov test

data: steady_state_distribution_get_data_experiment_three and
    recovery_distribution_get_data_experiment_three
D = 0.094853, p-value < 2.2e-16
alternative hypothesis: two-sided
```

Target Service	Registry and Ingress
Experiment Type	Inject Latency
Hypothesis	The application should be able to respond with the requested data but not being in the expected response time interval
Blast Radius	One Configuration Microservice with ID = 1
Status Before	N/A
Status After	N/A
Start Time	N/A
End Time	N/A
Findings	N/A

Table 8.11.: Experiment details of experiment no. 4

8.3.5. Experiment No. 4: Inject Latency into the Registry Component

Design

This experiment has been derived from Table 6.5, row seven. Due to lack of technical implementation, this experiment has not been executed but still be designed. The idea was to use Chaostoolkit's Chaos Monkey for Spring Boot integration [Wil18] to inject latency between two service calls. Table 8.11 shows the details of this experiment. Since the experiment has not been executed there is missing information. Ideally, *Status Before* contains a response time interval that is still valid after the experiment has been executed. Based on the previous experiment the *Status Before* could be set to $[0, 200]$ for retrieving configuration identifiers and $[0, 100]$ for retrieving the configuration data. The intervals are then also implemented in the steady-state-hypothesis of the Chaostoolkit experiment.

However, Chaos Monkey for Spring Boot requires that Spring's Actuator endpoint is enabled as it is the case in the prototype system. Unfortunately the Chaostoolkit's integration of the Chaos Monkey for Spring Boot is not able to handle communication by HTTPS which prevents an actual execution in this work. In order for the Chaos Monkey to work it would have been required to significantly extend the Chaostoolkit API which is not possible in the scope of this work. Listing 8.15 shows the method declaration of the experiment implemented in Chaostoolkit. As seen in the method declaration the actuator endpoint is used to enable the Chaos Monkey. Furthermore the assault's configurations are passed as arguments where it can be specified what type of assault is going to be needed and how it is configured.

Listing 8.15 Potential method declaration of experiment no. 4

```
"method": [
          {
              "type": "action",
              "name": "enable_chaosmonkey",
              "provider": {
                  "verify_tls": false,
                  "arguments": {
                      "base_url": "https://52.169.78.204/actuator"
                  },
                  "func": "enable_chaosmonkey",
                  "module": "chaosspring.actions",
                  "type": "python"
              }
          },
              {
                  "name": "configure_assaults",
                  "provider": {
                      "arguments": {
                          "base_url": "https://52.169.78.204/vccrest/registry/getDataPools",
                          "assaults_configuration": {
                              "level": 5,
                              "latencyRangeStart": 2000,
                              "latencyRangeEnd": 5000,
                              "latencyActive": true,
                              "exceptionsActive": true,
                              "killApplicationActive": false
                          }
                      },
                      "verify_tls": false,
                      "func": "change_assaults_configuration",
                      "module": "chaosspring.actions",
                      "type": "python"
                  },
                  "type": "action"
              }
   ]
```

Nonetheless, assuming the experiment would have been executed on the prototype system the further procedure would have been the same. The response times would have been analyzed with respect to the different phases, i.e., the steady-state, injection phase and recovery phase. The distributions may then be further analyzed by the Kolmogorov-Smirnov test to determine if the experiment would have had a significant impact on the steady-state of the system.

8.4. Discussion

In total four experiments were designed where only three could be executed based on technical feasibility. The experiments were derived from the results of Chapter 6. To verify a potential impact on the steady-state or the system's ability to recover the results of the experiments were analyzed by the Kolmogorov-Smirnov test. The hypothesis tests

produced a $D-$ and a $p-value$ as output. The $D-values$ describe the difference in the shape between two distributions that have been compared, i.e., the steady-state distribution and the injection phase distribution as well as the steady-state distribution and the recovery phase distribution. The distributions that have been compared are highlighted in Figure 8.1. For the Kolmogorov-Smirnov test a significance level of $\alpha = 0.05$ was assumed.

8.4.1. Summary of Experiment No. 1

The first experiment was successfully executed. The null-hypothesis stated that the response times will not increase despite the termination of a Kubernetes pod. However the diagrams showed a slight increase after the pod has been terminated. Furthermore the system was able to quickly recover from the pod termination. The $p-$ and $D-values$ calculated by the Kolmogorov-Smirnov test allowed a more precise analysis of the data. The first comparison of distributions, i.e., the steady-state and the injection phase, led to a $p-value$ of $2.2 * e^{-16}$ and a $D-value$ of 0.15062 with respect to the registry service. Moreover the comparison of the steady-state and injection phase distributions with respect to configuration microservices led to a $p-value$ of $2.2 * e^{-16}$ and a $D-value$ of 0.33061. The $D-values$ showed that the first experiment did not lead to a significant change in the shape of distributions concerning the registry component. However the distribution resulting from any request towards the configuration microservices shown in Figure 8.4 changed significantly with a $D-value$ double as high as the $D-value$ concerning the registry's distribution. This has been expected since one microservice was unavailable for a specific period of time due to the chaos experiment injection. Considering the recovery phase of the system with a $D-value$ of 0.40061 it is evident that Kubernetes is not able to completely recover from the injection within the last minutes of the experiment. The $p-values$ were lower than the defined significance level α which indicates that the null-hypothesis should be rejected in favor of the alternative-hypothesis. In scope of the chaos experiment the alternative-hypothesis, i.e., the steady-state-hypothesis, is met whereas the $p-value$ strengthen the confidence in the decision to reject the null-hypothesis.

8.4.2. Summary of Experiment No. 2

In the second experiment the blast radius of the experiment has been slightly increased by aiming to terminate all available configuration microservices that are currently deployed as pods. The diagrams Figure 8.7 and Figure 8.4 showed a much higher steady-state deviation as in the first experiment. This is an expected outcome of the experiment

since all configuration microservices are unavailable until Kubernetes creates new pods. As observed Kubernetes approximately needs two to three minutes until the pods are created and running. This effect can be observed in the diagrams in the last four to five minutes where the system tries to get back into a steady-state. The steady-state deviations concerning the registry and the configuration microservices become more evident analysing the $D - values$. The $D - value$ concerning the registry has a value of 0.38137 indicating a higher impact on the steady-state than the previous experiment since the shapes of distributions have a higher difference. The $D - value$ concerning the configuration microservices is 0.50391 which indicates a significant impact on the steady-state and is so far the highest calculated difference in the distribution's shape. The $D - values$ with respect to the recovery of the system show that Kubernetes was not able to fully recover the steady-state. However it is evident that the $D - value$ with respect to the recovery phase is almost equal to zero which indicates a small difference in the shape of the distributions. The $D - value$ with respect to the configuration microservices is still high considering the $D - value$ during the experiment injection. The $p - values$ of the second experiment are equal to the $p - values$ of the first experiment leading to the decision that the null-hypothesis should be rejected in favour of the alternative-hypothesis. Therefore this experiment also resulted in the steady-state-hypothesis being met.

8.4.3. Summary of Experiment No. 3

The third experiment increased the blast radius significantly more than the previous experiments. An additional random configuration object was created by requesting the administrator API. Thus, two additional pods containing one configuration microservices each were created by Kubernetes. In total four Kubernetes pods with respect to configuration microservices were running inside the Kubernetes cluster. The experiment was expected to terminate two configuration microservices with the same label but still return at least the configuration data of the additional configuration object despite an increase in response times. The distributions concerning the registry showed an interesting deviation in the steady-state as well as in the recovery phase as seen in Figure 8.11. However, interestingly the distributions concerning the microservices as seen in Figure 8.12 were not as significant as the deviations in Figure 8.11. The findings are suspicious based on the expectation that the response times concerning the configuration microservices should have increased to a higher extend that the response times concerning the registry. Inspecting the statistics calculatued by the Kolmogorov-Smirnov test emphasize the findings visible in the diagrams. The $D - value$ with respect to the registry has been calculated to be 0.28815 which seems a little bit too small considering the large shift of the medians. The $D - value$ with 0.041004 concerning the configuration microservices reflects the observations made by inspecting Figure 8.12.

Furthermore, the system has not been able to get back into a steady-state considering the $D - value$ of 0.39223 concerning the registry. Additionally, it has been observed that the cluster node terminated the pod running the database service. During the experiment the database pod transitioned into an *evicted* state which states that the pod exceeded its assigned memory threshold. This might at first be not visible to the user but becomes crucial to operating the system from an administrator point of view. Since the database is unavailable, configuration objects cannot be created, deleted or updated. Furthermore Kubernetes does not remove evicted pods automatically leading to the necessity of manual operation of the Kubernetes cluster state. The $p - values$ of this experiment were equal to previous experiments. Hence the null-hypothesis is rejected in favor of the alternative-hypothesis.

8.4.4. Summary of Experiment No. 4

The fourth experiment could not have been executed based on lacking technical feasibility. However, the Chaos Monkey for Spring Boot could have been used to a large extent by implementing experiments based on the results of the CHAZOP study in Section 6.3. It has been shown, that the experiment was designed equal to the previous experiments with respect to the description layout and possible measurements and hypothesis tests. Listing 8.15 showed a possible implementation in Chaostoolkit where the Chaos Monkey is going to be enabled and configured. Though it has to be considered that the Chaos Monkey for Spring Boot is restricted to Spring Boot based application. Furthermore, it requires Spring's Actuator endpoint to be enabled in the application.

8.4.5. Summary of Findings

To summarize, the experiments were executed as designed in this chapter. Furthermore, the results have been investigated visually by inspecting the resulting boxplot diagrams as well as statistically by applying the Kolmogorov-Smirnov test based on different distributions highlighted in Figure 8.1. The results were as expected with the exception of the third experiment revealing major weaknesses of the system. The impact this weakness would have had on the fourth experiment is unknown due to missing information. However, it is expected that the revealed weakness would have had a serious impact on the fourth and possible future experiments. Based on the principles of chaos engineering it is recommended to fix any revealed weakness before further experiments are executed. This mitigates the possibility that potential weakness that occur during further experiments might result from previous detected weaknesses instead of resulting from the experiments themselves. Since it is not the objective of this work to perform

any maintenance on the system the execution of further experiments would have been prevented by the revealed weakness of the third experiment. Nonetheless, as stated the next step in the chaos engineering cycle would require the engineers to fix the issue and run the third experiment again in order to verify that the issue has indeed been fixed. This finally leads to the necessity of automating chaos experiments which allows continuous verification of eliminated weaknesses.

8.5. Threats to Validity

In the following, the threats to validity are discussed with respect to the conducted case study.

8.5.1. Threats to Interviews

There are several threats to the interview process. All information about the prototype system have been collected from interviews which have been conducted at the beginning of this case study. A potential threat is the selection of a wrong interview type. This has been addressed by thorough analysis of different types of interviews. A major concern of interviewing participants is the group-think effect which often occurs during focus groups or group interviews. This issue has been dissolved by conducting one-on-one interviews. Another threat is stating interview questions not precise enough to such an extend that the participants might understand the question different to the researcher's intend. This might result in wrong interpretations of the participant's responses. To mitigate this issue, the responses of the participants were repeated and summarized by the interviewer in order to give the participants the possibility to adjust misunderstandings. Further, the participants were provided a transcript of the corresponding interviews which have been reviewed and sent back to the interviewer. Though, there is still a small risk that the participants have missed out to adjust false interpretations done by the interviewer.

Furthermore, as the diagrams and other drawings created from the interviews were further edited, it is possible that the artifacts were altered to a state were the information delivered by the artifacts is not representing the intended information. This issue has been solved by discussion the edited material with the interviewees to verify that the intended information has been preserved.

Another major threat to interviews is lacking questions to gather all necessary information with respect to the research questions. For example, the set of questions that has been stated during the interviews might not have covered all research questions that were intended to be answered by the interview sessions. In order to mitigate this

threat the interviewer provided a list of questions with a mapping to the corresponding research questions. This allowed a precise overview of the questions that have been asked in order to reduce the chance of leaving a research question unanswered.

Interviewing developers about the investigated case increases the chance of leaving out information that is crucial to the system's properties. This threat has been tried to be mitigated by giving the participants the opportunity to add comments after every question topic. However there is still a small risk that participants may have left out crucial information unintentionally. A better approach for future case studies would be to discuss the artifacts resulting from interviews with the participants, e.g., architecture descriptions, behavior and structural diagrams.

Despite the interview question related threats there is the risk of the existence of other resources to elicit the required information. This issue has been completely solved by asking each interview participant if other resources exist and are open for evaluation. As every participant stated there were no other resources available that could have yield any important information.

8.5.2. Threats to Hazard Analysis

As most hazard analysis approaches are conducted qualitatively there are several threats to the results of the hazard analysis. The FMEA and CHAZOP approaches are group activities as stated in Section 2.2.1 and Section 2.2.3. The scope of this work and time constraints of the developers, did not allow a thorough group review of the results. This increases the chance of working with incomplete or inadequate data. However, this issue has been mitigated by reviewing most of the results of the FMEA, CHAZOP and FTA with two developers once during one-hour session. Though there is still a high risk of results being incomplete incomplete or inadequate based on too less conducted reviews. Future approaches to these techniques should be done in strong collaboration with the developers including multiple review sessions to increase the quality of the findings.

Moreover, there are threats to a specific aspect of the FMEA as well. The severity levels of the FMEA were qualitatively estimated. Thus, there is a high chance in the severity levels being not accurate to the real severity of the events. This issue has been tried to be mitigated, by providing a mapping of severity levels to the impact an event would have had on the users of the system. However, the mapping of severity levels and the estimations of individual events is based on the researchers perception. Though the researchers perception might be different to the developers perception. Thus, leading to individual events having a severe outcome from an operators or developers perspective. Future migitation strategies may include thorough reviews with the developers and stakeholders of the system as it has been suggested.

8.5.3. Threats to the Experiment Environment

Threats to the experiment environment comprise threats to the generated workload and the platform choice. Generating large workloads with JMeter, e.g., large number of concurrent users, might lead to an instability of the JMeter instance. Thus leading to significant performance issues. As JMeter gathers a lot of information that require a lot of memory space, allocating too less memory to the JMeter instance might also lead to significant performance losses. In order to avoid such performance losses JMeter was configured to store only the most necessary information. This also means that any kind of listener was removed from the test plan in order to avoid additional usage of memory space.

Another threat to the generated workload might is the lack of adequate metrics with respect to the expected workload and performance. The interviewees provided rough estimates of the expected workload which have been used as a threshold for this case studies' chaos experiments. However, since the metric was a rough estimate there is still a chance of being inaccurate with respect to the real production-environment. Thus, leading to non-representative response times. This threat could not have been mitigated in scope of this work since it would have required more information from the stakeholders who were not accessible during this work.

The observations measured by JMeter allow to assume that the duration of the JMeter runs were not long enough to capture the steady-state of the application. This has been mitigated by performing multiple JMeter runs with different duration times without performing any failure injection. Inspecting the results of these runs increased the confidence in the provided time intervals were the steady-state is expected to occur.

Additionally, there exist further threats to the platform choice. As each platform behaves differently the measured response times in context of the Azure Cloud might not be significant if the investigated system will later be deployed on a different platform. However it was not part of this work to evaluate different platform providers therefore this issue has been neglected. Though it has to be considered that it is possible that on a different platform, e.g., Amazon Web Services (AWS), Kubernetes would have been able to recover faster from any chaos experiment than on AKS. Therefore, a variance on measured response times is possible.

8.5.4. Threats to Chaos Engineering Experiments

Several threats to chaos engineering experiments have to be considered. Lacking further metrics also constitute a threat to the drawn conclusions of the chaos experiments. Missing proper resilience and performance metrics prevents to accurate conclusions

from the results being in a given tolerance. As no tolerances have been provided, it is not possible to state if any deviation from the steady-state is considered to be alright or not. This issue could have only been reduced by providing well-defined performance and resilience metrics by the stakeholders. Therefore, the results of the hypothesis tests and experiments could merely be considered to have an impact on the system or not. Any further assessment of the results is not possible. As the scope of this work and the inaccessibility of the stakeholders prevented a detailed requirements elicitation, this issue has to be left as it is.

With respect to the chaos experiment execution, the results are threatened by inaccurate monitoring. A major concern in conducting chaos engineering experiments, is to provide accurate monitoring in order to extrapolate from the injected failure to the experiment findings. Inadequate monitoring might prevent precise conclusions or even lead to wrong conclusions based on the measured effects. This issue was addressed, by trying to run every experiment at the exact same time. Observing the measurements of different experiments at the same time interval, suggest that the experiments hold true to the expected outcome. However, for future approaches to chaos-experiments it is suggested to use precise monitoring tools in combination with performance and resilience metrics to allow a better analysis and interpretation of the results.

Moreover, there are threats not directly linked to the chaos engineering experiments but to the investigated system. As the chaos experiments revealed, the Kubernetes cluster has never been tested under the conditions given in the experiments, i.e., experiencing the workload generated by JMeter. The eviction of the database pod occurred after creating more than one configuration data object in the database. Hence, the state of the system prevented to create a representative environment with more than two configuration data objects. Unfortunately, it was not possible to solve the issue in the scope of this work nor has it been considered part of this work to solve the findings from chaos experiments. However, this issue should be addressed immediately before any future chaos experiments are executed. If this issue is not solved first, it is possible that future experiments will be influenced negatively to an extend where it is not possible to draw conclusions from the experiment.

8.5.5. Threats to Hypothesis Tests

Despite the threats to the chaos engineering experiments, there exist threats to the hypothesis tests as well. A major threat is the risk of applying the wrong hypothesis test. This issue has been mitigated by following the decision matrix provided by Field et al. [FT00, p. 274,275]. The decision matrix resulted in applying the Wilcoxon test. However since the shape of the distributions were considered as well it has been decided

to use the Kolmogorov-Smirnov test which has been stated to be a reasonable alternative to the Wilcoxon test.

Chapter 9

Conclusion

In the following, we will summarize the content and findings of this work, in addition, the findings will be discussed. At the end, an outlook is given with respect to extending this work.

9.1. Summary

In this thesis we provided means to conduct reasonable chaos engineering experiments in context of an industrial case study. The investigated system has been stated to be a prototype and shall substitute a legacy system that is insufficient in meeting crucial performance and resilience requirements. The information about the investigated system have been gathered by conducting semi-structured interviews with the developers of the system. This information was used to provide a detailed description of the structure and behavior of the system. Existing risks and potential hazards present in the prototype system have been identified by applying hazard analysis techniques, in particular Failure Modes and Effects Analysis, Fault Tree Analysis and Computational Hazard and Operations Study. To create an environment similar to production, the prototype system was deployed onto the Azure Cloud by using a managed Azure Kubernetes Service. Environmental conditions were simulated by generating specific workloads with Apache Jmeter. The results of the hazard analysis were used as a basis to implement chaos engineering experiments in Chaostoolkit. Lastly, the chaos engineering experiments were designed and executed according to the hazard analysis findings. Furthermore, the results of the chaos engineering experiments were analyzed by applying the Kolmogorov-Smirnov hypothesis test in order to observe a potential impact on the system's steady-state.

In addition to the overall approach on conducting chaos engineering, major contributions were made by introducing hazard analysis techniques, known from other software engineering domains, to chaos engineerig as well as a hypothesis-testing based evaluation of chaos engineering experiment findings.

Thorough research (Chapter 3) has shown, that current chaos engineering approaches require high sensitivity towards the occurrence of failures in a production environment. Hence, forcing a change in the mindset of engineers of how systems should be built. Fault Tolerance and Fault Avoidance techniques were elaborated to lack the capacity of facing changes in a running system by requiring previous knowledge of the type of failures that may occur. Chaos engineering suggests to shift the perspective of engineers from avoiding failures to actively confront them. However considering that fault injection techniques have already been existing long enough to be well-established a major issue in applying chaos engineering lies in identifying potential chaos engineering experiments.

9.2. Discussion

The current state of the art suggests to randomly inject failures into a system that runs in a production environment. Though recent developments in the field of chaos engineering have shown a shift from applying chaos engineering in production environment to pre-production environments such as at design-level and testing environments. Moreover the randomization aspect is decreasing and substituted by a more systematic approach targeting individual services and components such as provided by the Gremlin platform. Applying chaos engineering in any pre-production environment requires means to identify potential failures that can be injected during chaos engineering experiments.

The proposed hazard analysis techniques are applicable on the architecture level by investigating individual components and their connections. Nonetheless the proposed means to identify potential hazards require time and effort in order to establish reasonable results. FMEA and CHAZOPs require continuous group reviews as well as a sensitivity for potential risks within the application's architecture. Further analysis with a FTA or other top-down approaches allows to identify root causes that were not considered during the design of the system. In this work such root causes included the ceasing of managed Kubernetes objects such as the ingress or registry service. As such services are not operated by engineers but rather by Kubernetes itself these components and services are often neglected during analysis. Considering managed services for chaos engineering experiments opens up a large range of potential chaos engineering experiments that can further increase the resilience of a system. Furthermore it is possible to specifically assess Kubernetes' self-healing mechanism by targeting managed Kubernetes services.

Thus operators and engineers are able to predict to which extend Kubernetes is able to maintain its state and at which point in time a manual operation is necessary.

In scope of this thesis, the proposed hazard analysis techniques with respect to FMEA and CHAZOP yielded great results though being not as time-consuming as with larger systems. In context of a microservice architecture the number of microservices decides on the effort required to conduct the proposed means to identify risks and hazard. Within a monolithic application the number of services may not be sufficient to the effort but the number of interconnections increases the complexity of the CHAZOP and FTA significantly. Other means such as Event Tree Analysis and Failure Modes, Criticality and Effects Analysis require probabilistic quantification of events. Therefore these techniques were not feasible to be applied in this work. However, these techniques may become feasible if chaos engineering is taken to the production level where more information can be gathered such as probabilistic estimations of potential failures. Thoug, establishing such estimations requires proper monitoring and documentation over a long period of time.

The FTA conducted in Chapter 6 had acceptable complexity with respect to the number of microservices and potential events. A FTA applied on larger distributed systems may become infeasible due to growing complexity of component interactions and tree size. Restructuring trees is a mean to reduce the complexity of a fault tree but always comes with a trade-off between complexity and readability. This leads to the question if a FTA at such a scale is still feasible or not. Growing complexity and time effort has to be considered for FMEA and CHAZOP approaches as well. An increasing number of microservices leads to significant increases in interconnections that have to be considered during a CHAZOP study. Furthermore the FMEA will grow in size as well since more components have to be investigated.

The results of the hazard analysis were used to implement chaos engineering experiments in Chaostoolkit. As research has shown there exists a large variety of distinct chaos engineering tools. However current approaches have not revealed a state of the art with respect to a dedicated tool. Existing tools mainly differ in their capability of injecting chaos into specific application-levels such as platform-, application-, infrastructure- or network-level. Therefore decisions about which tool to take are primarily driven by the experiment design. In this work Chaostoolkit has been selected based on its capability to include multiple drivers for different platforms thus increasing the number of potential experiments that can be executed with a single tool. Despite providing an open API the Chaostoolkit's capability of injecting chaos on the Kubernetes platform-level are restricted to a few being useful. Though it has to be considered that the possibilities of Chaostoolkit in the scope of this thesis are restricted by the system environment. For example, Chaostoolkit provides a mechanism to drain a Kubernetes node with respect to

its memory. Within a one node cluster, as given in this work, draining a node may lead to significant damages in the state of the system if no proper fallbacks are declared.

Nonetheless, Chaostoolkit provides a rich set of injections that are easy to integrate and easy to use. The focus of the executed experiments in Chapter 8 were narrowed down to terminate specific Kubernetes pods. Hence the experiments reflected root causes identified by FMEA and FTA. The declaration in JSON or YAML provides transparency and well-structured chaos engineering experiments. Other chaos engineering practitioners develop their own tools that are specifically targeted at their infrastructure, e.g., Netflix's Simian Army. However most of these tools are publicly available but lack simplicity and transparency to become valuable at a beginner's level.

As far as research goes means to determine if a chaos engineering experiment had significant impact on the system's steady-state are not made transparent. It is known that Netflix uses a well-defined business metric, Stream Starts per Second, to identify impacts on the steady-state. Other metrics used by other chaos engineering practitioners are yet unknown. In this work I have applied basic hypothesis testing as it is known from research activities to statistically evaluate the results from chaos engineering experiments. Despite the difficulty in evaluating the results without well-defined resilience metrics the selection of the proper hypothesis test is crucial to establish meaningful results. The choice of a proper hypothesis test is entirely dependent on the data set that is to be analyzed. Therefore the test used in this work might not be applicable to other data sets.

Another difficulty is the lack of proper monitoring. Monitoring is a key aspect of being successful with chaos engineering. Lacking proper monitoring may influence the experiment results to a large extend. In the scope of this thesis it was not possible to setup a proper monitoring environment. Although proper monitoring should already be in place from the beginning not only from a chaos engineering perspective but also from a performance monitoring perspective.

In total, the goal of this case study has been achieved by successfully applying means to identify potential risks and hazards on the architectural level. The hazard analysis results imply that the prototype system is considered to be rather fragile as proven by revealing a multitude of single points of failures. Further a dedicated number of risks were implemented as chaos experiments and later been injected on the platform-level to observe deviations in the system's steady-state. The observations were analyzed statistically by applying a non-parametric two-sided Kolmogorov-Smirnov test. The chaos engineering experiments revealed major weaknesses where most of them have been expected to occur.

9.3. Future Work

In the following I will outline possible extensions to this work as well as potential research areas that require more work.

9.3.1. Visualization

Visualization is tightly coupled to the monitoring aspect within chaos engineering experiments. Currently each chaos engineering experiment is designed as a black box. Steady-state deviations can be statistically analyzed but it is yet difficult to determine what happens within a system after an experiment has been started. This makes analysis of chaos engineering experiments incredibly difficult. Visualizing the interconnections within an architecture could disentangle the complexity of chaos engineering experiments by precisely visualizing the experiments. This can be extended on to the platform-level, e.g., leveraging the Kubernetes API to continuously observe the state of the cluster. As of now there are no related works done in this field.

9.3.2. Evaluation of different Chaos Engineering Tools

As of now, there exists a large variety of potential chaos engineering tools. A brief summary of tools has been provided in Section 2.7. However, most of these tools are dedicated to inject chaos into one specific level, e.g., Pumba is specifically designed to target Docker containers. As no case studies or other evaluations exist it is not known if other tools have equal potential to Chaostoolkit as being valuable both on the beginner's as well as the professional level. A thorough evaluation of different tools may serve as an extension to this work by conducting an industrial case study with the proposed approach but with a distinct chaos engineering tool.

9.3.3. Exploration of Resilience Pattern Configurations

The prototype system investigated in this work did not include any resilience patterns such as Circuit Breaker or Bulkhead. Though applying resilience patterns to increase the resilience of a software application is a common procedure. Nonetheless, deciding which pattern to use is mainly driven by the problem statement that is expected to be solved by the pattern. If a pattern has been chosen it has to be decided how the parameters of the resilience patterns are configured. Even for a single pattern this may result in a time-consuming task. Now consider the presence of multiple patterns within a single

application where each pattern has a set of parameters that can take on several complex configurations. The number of potential configurations for a set of resilience patterns quickly increases into millions of possible configurations. Automatic exploration of such resilience pattern configurations is a yet unsolved problem. Chaos engineering provides the possibility to automatically explore such configurations on a large scale. Related works have already been proposed in different research areas such as in the context of architecture optimization. As chaos engineering is yet evolving there exists no approach leveraging the proposed approaches used in this work to solve this problem.

Appendix A

Interview Resources

In the following the supplementary material used in the interviews is provided. This chapter is structured as follows.

Section A.1 – Question Catalogue: - Interview Question Catalogue.

Section A.2 – Consent Form: - Interview Consent Form (unsigned)

A.1. Question Catalogue

Interview Schedule Researcher: Dominik Kesim 04/09/2019

Interview Schedule

I) Opening

A) (Establish Rapport) [Welcome] Welcome, and thanks that you accepted to take part in this interview. My name is _____ and I am currently writing my bachelor's thesis under the supervision of Matthias Häussler.

B) (Purpose) I would like to ask some questions about the kbv project. Matthias has told me that you are currently involved in the project. My questions will cover different topics about the application such as its architecture, behavior, and your role within the project. This is especially helpful for my thesis, in order to get a better understanding of the application.

C) (Motivation) As I said, I hope to use this information to get a better understanding of the application's behavior and architecture and in order to establish a better experiment design for the chaos engineering experiment.

D) (Time Line) This interview should take approximately 1 hour and 30 minutes.

(Transition: Let us start by asking some questions about yourself)

II) Body

A) (Topic) Demo graphical data

i) What's your name and how old are you?
ii) What is your job title?
iii) How many years have you been working for this company?
iv) What kind of projects have you been working on so far (in general) ?
v) Are you currently working on other projects than kbv?

(Transition: At this point I would like to transition to some project related questions)

B) (Topic) Project related questions

i) How long have you been involved in the project?
ii) What is your role within the project?
iii) What is the purpose/motivation for this project?
iv) What is the current state of the project?
v) How is the project developed (scrum (agile), waterfall …)?
vi) What have you planned as your next steps?
vii) Retrospectively, what would you do differently?

Figure A.1.: Question Catalogue page 1/3

Interview Schedule Researcher: Dominik Kesim 04/09/2019

(Transition: Thanks a lot for those insight! Now, I would like to talk a little bit more about the requirements of the application.)

C) (Topic) Requirements
   i) Could you please, briefly summarize what functional and qualitative requirements the “kbv” application has to meet?
   ii) How did you elicit these requirements?
   iii) Have there been any constraints, either given by the customer or which you concluded by yourself?
   iv) Up to now, have you been able to implement/meet all requirements?
   v) Is there any documented version of those requirements that is open for analysis?

(Transition: Thanks! After we have talked about the requirements you have to implement, let’s talk about the technologies that allows you to implement them.)

D) (Topic) Technology stack

   i) What technology stack do you use?
   ii) Have there been other potential technologies you considered and did not choose?
   iii) How did you decide on which particular technology to use?
   iv) Retrospectively, would you have done something differently? (e. g. use a different container virtualization or deployment platform)

(Transition: Let’s get to a more specific topic, namely the architecture of the application. I have separated the topic into two sub topics which consist of the structural and behavioral view on the application’s architecture. We will start with the structural view.)

E) (Topic) Structural View

   i) What architectural style did you use?
   ii) Why did you choose this particular style?
   iii) What kind of data do you work with in your application?
   iv) What does the data look like (type, structure, format)?
   v) What are the main functional elements of your architecture?
   vi) Can you please briefly summarize, what each component’s responsibility is?
   vii) How did you identify your microservices?
   viii) Is there any relationship between components such as a generalization or specialization?
   ix) Is any component dependent on other or external software components?
   x) How are the components or more precisely the application itself deployed?
   xi) Have you integrated any architectural patterns? If yes, which ones?
   xii) How did you decide on which patterns to use?
   xiii) What would you consider to be the benefits and drawbacks of your current architecture?
   xiv) Retrospectively, would you do anything differently?

F) (Topic) Behavioral View

Figure A.2.: Question Catalogue page 2/3

i) Can you give me an explanation of what you expect, how the application should work? If possible, can you draw a use case diagram or sequence diagram?
ii) How do the components communicate/interact with each other?
iii) How do the components communicate with the outside world (e. g. external systems)?
iv) What type of communication do you have within the application (e. g. asynchronous)?
v) Do you have any metrics regards the performance of your current implementation?
vi) Do you have any metrics regards the load profile of your application?
vii) Is there any archival data open for analysis (requirements specification, architecture diagrams …)?

(Transition: Thanks a lot for your time, I appreciate it! I have gathered a lot of important information.)

III) Closing

A) (Feedback) Thanks for your time. Is there anything else you would like to leave as a comment or feedback?

B) (Next steps) I will use the recording and make a transcription of this interview. May I send you the transcribed interview so that you can provide corrections? Lastly, may I contact you if I have further questions?

Figure A.3.: Question Catalogue page 3/3

A.2. Consent Form

Consent for participation in research interview

Assessing Resilience of Software Systems by Application of Chaos Engineering – A Case Study

Bachelor thesis title: Assessing Resilience of Software Systems by Application of Chaos Engineering

Name of researcher: Dominik Kesim
Name of participants:

The purpose of this interview is to collect important information about the investigated software application, referred to as a caching service for car configuration data, which is part of my bachelor thesis.

The interview will take approximately 120 minutes. The questions are written and asked in German, hence, answers to these questions should be stated in German as well. If you feel uncomfortable or do not agree with the terms below, you have the right to withdraw or drop out of the interview session at any time. If you have any questions about the interview or need more information about the process, you have the possibility to ask questions before the interview begins.

Thank you for agreeing to being interviewed as part of the above bachelor thesis. Ethical procedures in academic research require that participants explicitly agree on being interviewed and how the information contained in their interview will be used. This consent form is necessary, to ensure that you fully understand the purpose of your involvement and that you agree to the conditions of your participation. I therefore ask you to read the information below and sign this form to confirm that you agree to the following:

- I …… voluntarily agree to participate in this research study.
- I understand that even if I agree to participate now, I can withdraw at any time or refuse to answer any question without any consequences of any kind.
- I understand that I will not benefit directly from participating in this research.
- I understand that all information I provide for this study will be treated confidentially.
- I understand that in any report on the results of this research my identity and the identity of any project-related members or customers, will remain anonymous. This will be done by changing the names and disguising any details of my interview which may reveal my identity or the identity of people I speak about.
- I understand that disguised extracts from my interview may be quoted in the bachelor thesis of Dominik Kesim, with the title “Assesing Resilience of Software Systems by Application of Chaos Engineering”. The thesis is conducted in collaboration with Novatec Consulting GmbH and the Institute for Reliable Software Systems of the University of Stuttgart. Submission date is the 1st September 2019.
- I understand that signed consent forms will be retained as physical data, where only the researcher and the responsible supervisors have access to, until the bachelor thesis is evaluated. The original audio recordings will be retained as digital data, where only the researcher and the responsible supervisors have access to, until the bachelor thesis is evaluated.
- I understand that under freedom of information legislation I am entitled to access the information I have provided at any time while it is in storage as specified above.
- I understand that I am free to contact any of the people involved in the research to seek further clarification and information.

Figure A.4.: Consent Form page 1/3

- I agree to my interview being audio-recorded.
- I agree that a transcript of the interview will be produced and analysed by the researcher Dominik Kesim.
- I understand that I have the opportunity to receive the transcript, recordings or any other Interview related documents in order to review the content and correct any factual errors.
- Access to the interview transcript and recordings is limited to Dominik Kesim, Dr. André van Hoorn (academic supervisor), Matthias Häussler (Novatec supervisor) and Christian Schwörer (Novatec supervisor).

Quotation Agreement

I understand that my words may be quoted directly. With regards to being quoted, please initial next to any of the statements you agree with.

	I wish to review the notes, transcripts, or other data collected during the research pertaining to my participation
	I agree to be quoted directly.
	I agree to be quoted directly, if my name is not published and a made-up name (pseudonym) is used.
	I agree that the researcher may publish documents that contain quotations by me.

Contact Information of Researcher:

Researcher: Dominik Kesim
Course of Study: Software Engineering
Affiliation: Thesis student at Novatec Consulting GmbH
Bachelor student at University of Stuttgart
Work e-mail: Dominik.Kesim@novatec-gmbh.de,
Alt. e-mail: Dominik.Kesim@gmail.com

Contact Information of Supervisors:

Academic Supervisor: Dr.-Ing. André van Hoorn
E-Mail: van.hoorn@informatik.uni-stuttgart.de
Novatec Supervisors: Matthias Häussler (mhs@novatec-gmbh.de),
Christian Schwörer (csh@novatec-gmbh.de)

Figure A.5.: Consent Form page 2/3

Signature of research participant

With your signature you confirm to have read the consent form and approve with the information provided above.

---------- ----------
Signature of participant Date

Signature of researcher

I believe the participant is giving informed consent to participate in this study

---------- ----------
Signature of researcher Date

Figure A.6.: Consent Form page 3/3

Appendix B

Incident Analysis Results

This chapter contains the results of the Hazard Analysis conducted in Chapter 6. This chapter is structured as follows.

Section B.1 – Failure Modes and Effects Analysis: - Results of Failure Modes and Effects Analysis.

Section B.3 – Computational Hazard and Operations Study: - Results of Computational Hazards and Operations Study

Section B.2 – Fault Tree Analysis: - Results of Fault Tree Analysis

B.1. Failure Modes and Effects Analysis

No.	Components	Failure mode	Sub system effects	System effects	Degree of severity
1	Ingress	Pod that hosts ingress is terminated	Registry does not receive any requests	User cannot request the registry to handle his/her requests	very high
2	Ingress	Replicaset that hosts the ingress is terminated	Registry does not receive any requests	User cannot request the registry to handle his/her requests	very high
3	Ingress	Service that exposes the ingress is removed	Ingress is still accessible inside the cluster but not from the outside	Users cannot access the system since the ingress is not exposed.	very high
4	Ingress	Deployment that hosts the ingress is deleted	Kubernetes Master cannot check cluster state for the ingress component	Kubernetes Master cannot heal the cluster state for the ingress component if deviations of cluster state occur	high
5	Ingress	Clusternode that hosts the ingress is terminated	Registry does not receive any requests	User cannot request the registry to handle his/her requests	very high
6	Ingress	Ingress receives not allowed http method (PUT, POST, DELETE) on user requests	Ingress returns an error message	User receives an error message in addition to missing data	very high
7	Ingress	http method of request is changed to POST, PUT or DELETE method	Registry returns an error because of not specified http method	User receives an error in addition to missing data	very high
8	Ingress	URI is changed from „.../datapool/all“ to „.../datapool/{id}“ but an identifier is not provided	Registry receives request for a single config data object but without an identifier	Registry returns an error to ingress because of missing identifier field	very high
9	Ingress	URI is changed from „.../datapool/all“ to „.../datapool/{id}“ but a random identifier is generated	Registry receives request for a random single config data object	User receives a random config data object	middle
10	Ingress	URI is changed from „.../datapool/{id}“ to „.../datapool/all”	Registry receives request for all config data objects	User receives a list of data instead of a single config object	middle
11	Ingress	URI is changed from „.../datapool/all“ to „.../datapool/create“	Registry does not provide a controller for given URI	User receives an error message	very high
12	Ingress	URI is changed from „.../datapool/all“ to „.../datapool/delete/{id}“ but an identifier is not provided	Registry does not provide a controller for given URI	User receives an error message	very high

Figure B.1.: Failure Modes and Effects Analysis of the Ingress component 1/3

No.	Components	Failure mode	Sub system effects	System effects	Degree of severity
13	Ingress	URI is changed from „.../datapool/all“ to „.../datapool/delete/{id}“ but a random identifier is not provided	Registry does not provide a controller for given URI	User receives an error message	very high
14	Ingress	URI is changed from „.../datapool/create“ to „.../datapool/delete/{id}“ but an identifier is not provided	Data management receives a request for deleting an object without an identifier	Data management returns an error because of a missing identifier	high
15	Ingress	URI is changed from „.../datapool/delete/{id}”to „…/datapool/create”	Data management receives request without data content to create	Data management returns an error because of missing data content	high
16	Ingress	Requests to registry are delayed by x ms/s with timeout	Requests to ingress are aborted after x ms/s	User receives an error message that request took too long	very high
17	Ingress	Requests to registry are delayed by x ms/s without timeout	Calling threads are blocked by ingress due to synchrony and blocks further incoming requests	Multiple users are not able to perform requests since ingress blocks incoming requests	very high
18	Ingress	Requests to registry are delayed	Response from registry is delayed	User receives a late response from ingress	middle to high
19	Ingress	Requests to registry are aborted without enforcing a retry	Registry does not receive a request	User receives an error message that request took too long	very high
20	Ingress	Requests to registry are aborted with enforcing a retry	Registry receives multiple requests of same kind and may shut down due to overload	Registry is not able to respond	very high
21	Ingress	Data content of response by registry is deleted	Ingress receives empty data content	User receives no data	very high
22	Ingress	Requests to registry are delayed by x ms/s with timeout	Requests to ingress are aborted after x ms/s	User receives an error message that request took too long	very high

Figure B.2.: Failure Modes and Effects Analysis Results of the Ingress component 2/3

No.	Components	Failure mode	Sub system effects	System effects	Degree of severity
23	Ingress	Requests to registry are delayed by x ms/s without timeout	Calling threads are blocked by ingress due to synchrony and blocks further incoming requests	Multiple users are not able to perform requests since ingress blocks incoming requests	very high
24	Ingress	Requests to registry are delayed	Response from registry is delayed	User receives a late response from ingress	high
25	Ingress	Requests to registry are aborted without enforcing a retry	Registry does not receive a request	User receives an error message that request took too long	very high
26	Ingress	Requests to registry are aborted with enforcing a retry	Registry receives multiple requests of same kind and may shut down due to overload	Registry is not able to respond	very high
27	Ingress	Kubernetes object that hosts ingress terminates after admin request is handled by database	Admin request has been correctly handled by the system	Ingress is not able to receive responses. Admin will not receive a response and thus does not know if request has been handled	middle
28	Ingress	Kubernetes object that hosts ingress terminates after user request is handled by registry	User request has been correctly handled by the system	Ingress is not able to receive responses. User will not receive any data	middle

Figure B.3.: Failure Modes and Effects Analysis Results of the Ingress component 3/3

No.	Components	Failure mode	Sub system effects	System effects	Degree of severity
1	Registry	Clusternode that hosts registry is terminated	Registry is no longer available for requests by Ingress and communication to Kubernetes Master	User may request the service but may not receive a response	very high
2	Registry	Deployment of registry is deleted	Kubernetes Master is not able to “heal” if cluster changes occur (e.g. pod termination)	Kubernetes Master is not able to “heal” if cluster changes occur (e.g. pod termination)	high
3	Registry	Pod that hosts registry is terminated	Registry is no longer available for requests by Ingress and communication to Kubernetes Master	User may request the service but may not receive a response	very high
4	Registry	Replicaset that hosts registry is terminated	Registry is no longer available for requests by Ingress and communication to Kubernetes Master	User may request the service but may not receive a response	very high
5	Registry	Service of registry is deleted	Registry is only available from inside the cluster	Registry is no longer available from outside the cluster	low
6	Registry	Cluster IP of registry is deleted	Registry is no longer available from inside the cluster	Ingress and other components may not be able to communicate with registry	very high
7	Registry	Registry receives requests with unknown URI (i.e. “datapool/create“)	Registry has no controller to serve URI. Registry returns an error message	User receives an error message and no data	very high
8	Registry	Registry receives requests with unknown URI (i.e. “datapool/delete/{id}”)	Registry has no controller to serve URI. Registry returns an error message	User receives an error message and no data	very high
9	Registry	Registry receives requests with unknown URI (i.e. “datapool/delete/{id”}	Registry has no controller to serve URI. Registry returns an error message	User receives an error message and no data	very high

Figure B.4.: Failure Modes and Effects Analysis Results of the Registry component 1/2

No.	Components	Failure mode	Sub system effects	System effects	Degree of severity
10	Registry	Registry receives requests with not authorized http method (i.e. “datapool/all” with http POST)	Registry returns an error message with status code 405 (method not allowed)	User receives an error message and no data	very high
11	Registry	Registry receives requests with not authorized http method (i.e. “datapool/all” with http PUT)	Registry returns an error message with status code 405 (method not allowed)	User receives an error message and no data	very high
12	Registry	Registry receives requests with not authorized http method (i.e. “datapool/all” with http DELETE)	Registry returns an error message with status code 405 (method not allowed)	User receives an error message and no data	very high
13	Registry	Response by registry to ingress is aborted	-	User does not receive any data	very high
14	Registry	Request to Kubernetes Master is aborted	Registry starts a retry.	Kubernetes Master receives delayed request. User may wait a long time for his request to be handled.	high
15	Registry	Initial request is aborted and subsequent retries to Kubernetes Master are delayed	Registry starts several retries with delay	Kubernetes Master receives several requests of same kind. Kubernetes Master may shut down due to overload	very high
16	Registry	Response from registry to ingress is delayed	Calling thread at ingress is blocked for a longer time.	Ingress may abort the request due to a timeout.	very high

Figure B.5.: Failure Modes and Effects Analysis Results of the Registry component 2/2

No.	Components	Failure mode	Sub system effects	System effects	Degree of severity
1	Data management	Clusternode that hosts data management is terminated	Communication to data management is no longer possible.	User may still receive data from cached configuration files, but data base confirmation is not possible	high
2	Data management	Deployment of data management is deleted	Kubernetes Master cannot re-create cluster state if changes in the cluster happen	Scaling becomes more difficult due to missing deployment. Terminations become more critical	high
3	Data management	Pod that hosts data management is terminated	Communication with data management is not possible.	User may still receive data from cached configuration files, but data base confirmation is not possible	high
4	Data management	Replicaset that hosts data management is terminated	Communication to data management is no longer possible.	User may still receive data from cached configuration files, but data base confirmation is not possible	high
5	Data management	Service that exposes data management is deleted	Data management is not accessible from outside the cluster	-	low
6	Data management	Cluster IP assigned to data management is removed	Data management is available but not accessible	User may still receive data from cached configuration files, but data base confirmation is not possible	high
7	Data management	Response from data management to Kubernetes Master is aborted	Data management does not know response is aborted	Kubernetes Master never receives a response. Kubernetes Master returns an error.	high
8	Data management	Response from data management to Kubernetes Master is aborted	Data management does know response is aborted and sends another response	-	high

Figure B.6.: Failure Modes and Effects Analysis Results of the data management component

No.	Components	Failure mode	Sub system effects	System effects	Degree of severity
1	Admin service	Pod that hosts admin service is terminated	Admin service is not able to receive requests	Admin cannot issue requests	very high
2	Admin service	Replicaset that hosts the admin service is terminated	Admin service is not able to receive requests	Admin cannot issue requests	very high
3	Admin service	Service that exposes the admin service is removed	Admin service is still accessible inside the cluster but not from the outside	Admin cannot access the system and therefore cannot issue any requests	very high
4	Admin service	Deployment that hosts the admin service is deleted	Kubernetes Master cannot check cluster state for the admin service component	Kubernetes Master cannot heal the cluster state for the admin service component if deviations of cluster state occur	high
5	Admin service	Clusternode that hosts the admin service is terminated	Admin service is not able to receive requests	Admin cannot issue requests	very high
6	Admin service	Admin service receives not allowed http method (GET) on user requests	Admin service returns an error message to ingress	Admin receives an error message by ingress and hence, request isn't handled by the system	very high
7	Admin service	http method of request is changed to GET method	Admin service does not provide a controller for GET requests	Admin service returns error to ingress which is received by the Admin	very high

Figure B.7.: Failure Modes and Effects Analysis Results of the Admin service 1/2

No.	Components	Failure mode	Sub system effects	System effects	Degree of severity
8	Admin service	URI is changed from „.../datapool/create“ to „.../datapool/{id}“ but an identifier is not provided	Admin service does not provide a controller for the given URL	Admin service returns error to ingress which is received by the Admin	very high
		URI is changed from „.../datapool/create“ to „.../datapool/{id}“ but a random identifier is generated			
		URI is changed from „.../datapool/delete/{id}“ to „.../datapool/all”, an identifier is provided			
9	Admin service	URI is changed from „.../datapool/delete/{id}“ to „.../datapool/create”, an identifier is provided, random data content is generated	Admin service receives correct request for creating an object	Admin receives response that object already exists in database and therefore cannot be created	low

Figure B.8.: Failure Modes and Effects Analysis Results of the Admin service 2/2

No.	Components	Failure mode	Sub system effects	System effects	Degree of severity
1	User service	Pod that hosts user service is terminated	User service is not able to receive requests	User cannot issue requests	very high
2	User service	Replicaset that hosts the user service is terminated	User service is not able to receive requests	User cannot issue requests	very high
3	User service	Service that exposes the user service is removed	User service is still accessible inside the cluster but not from the outside	User cannot access the system and therefore cannot issue any requests	very high
4	User service	Deployment that hosts the user service is deleted	Kubernetes Master cannot check cluster state for the user service component	Kubernetes Master cannot heal the cluster state for the user service component if deviations of cluster state occur	high
5	User service	Clusternode that hosts the user service is terminated	User service is not able to receive requests	User cannot issue requests	very high
6	User service	User service receives not allowed http method (GET, DELETE) on user requests	User service returns an error message to ingress	User receives an error message by ingress and hence, request isn’t handled by the system	very high
7	User service	http method of request is changed to GET, DELETE method	User service does not provide a controller for GET requests	Admin service returns error to ingress which is received by the Admin	very high

Figure B.9.: Failure Modes and Effects Analysis Results of the User service

B.2. Fault Tree Analysis

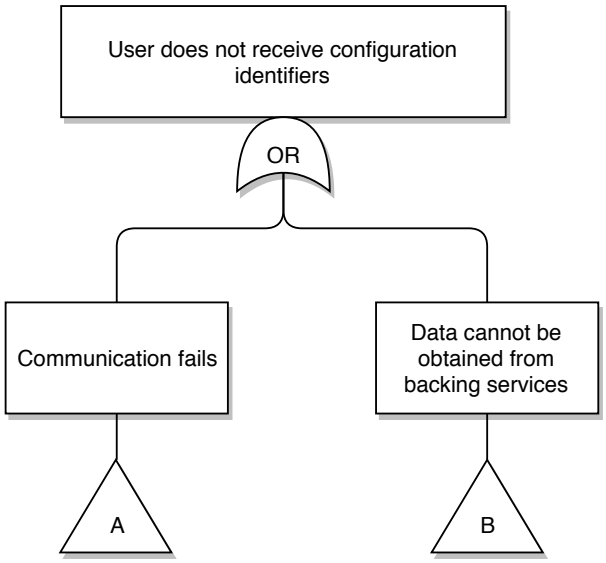

Figure B.10.: Fault Tree Analysis Results: User does not receive configuration identifiers 1/3

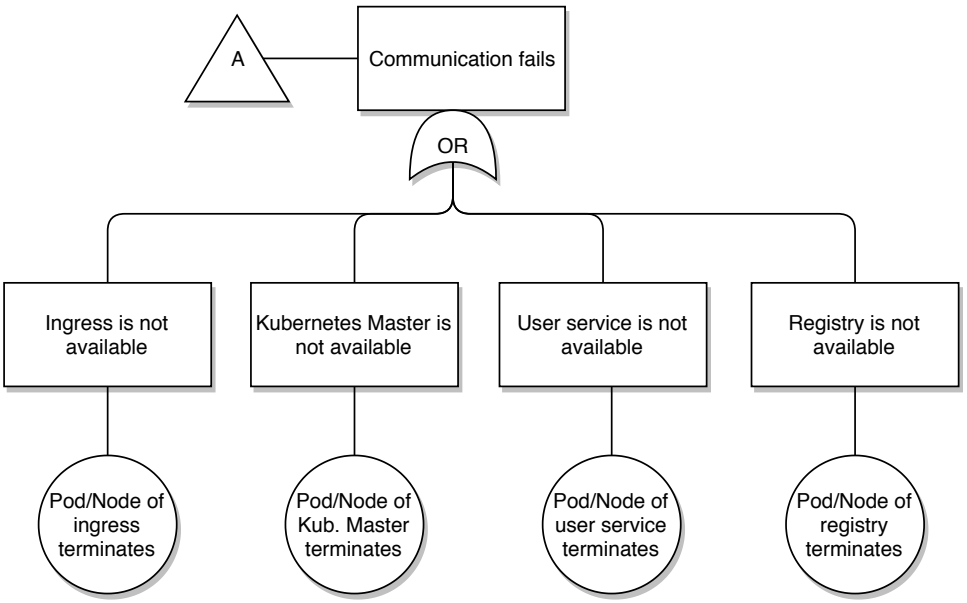

Figure B.11.: Fault Tree Analysis Results: User does not receive configuration identifiers 2/3

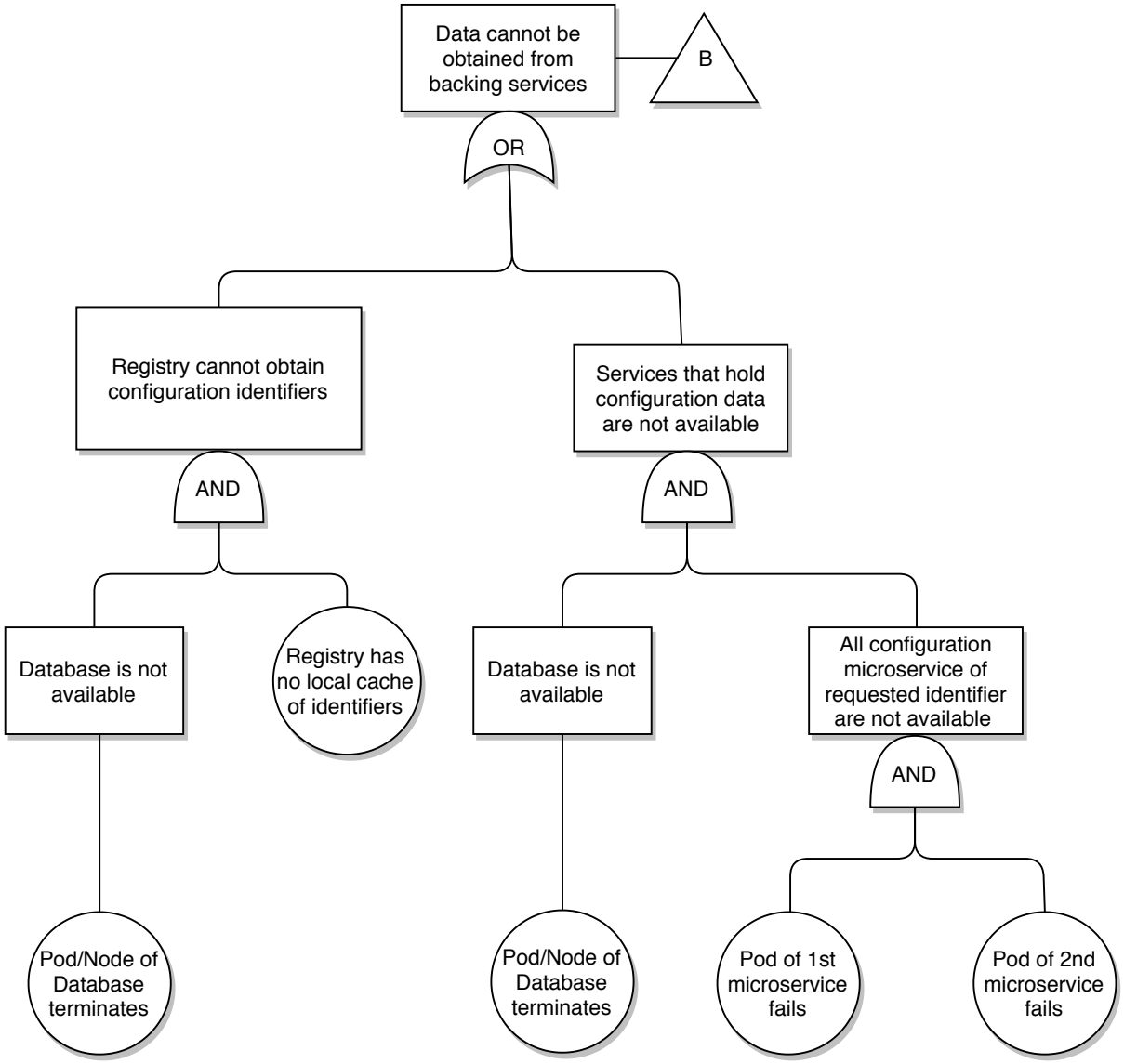

Figure B.12.: Fault Tree Analysis Results: User does not receive configuration identifiers 3/3

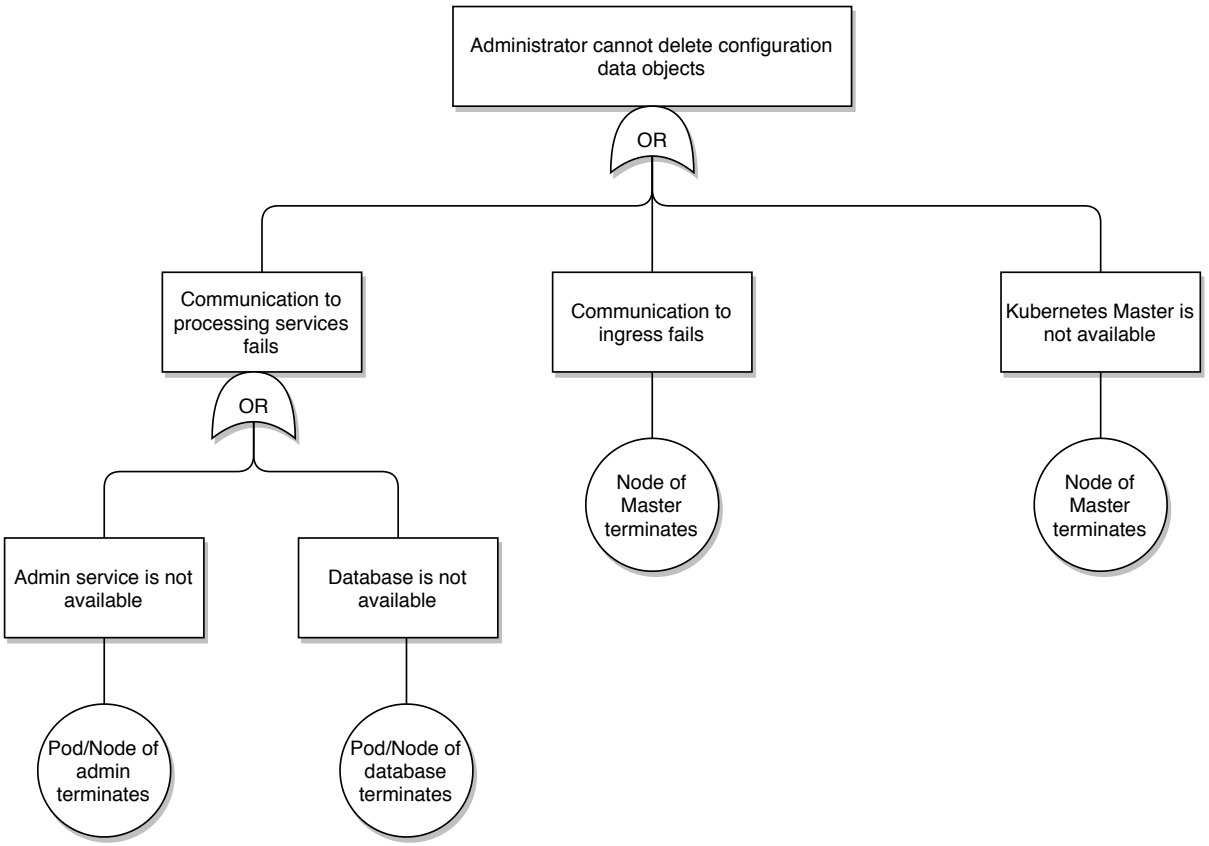

Figure B.13.: Fault Tree Analysis Results: Administrator cannot delete configuration data objects

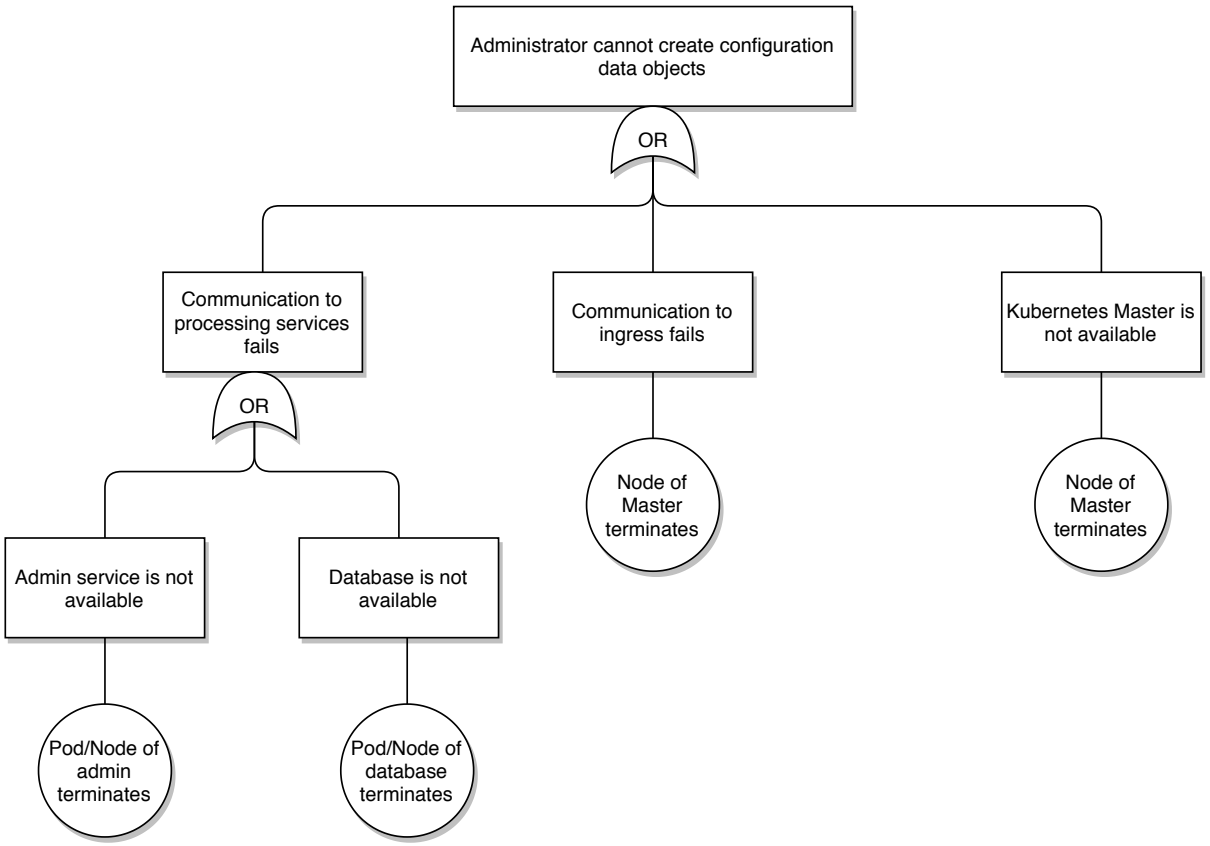

Figure B.14.: Fault Tree Analysis Results: Administrator cannot create configuration data objects

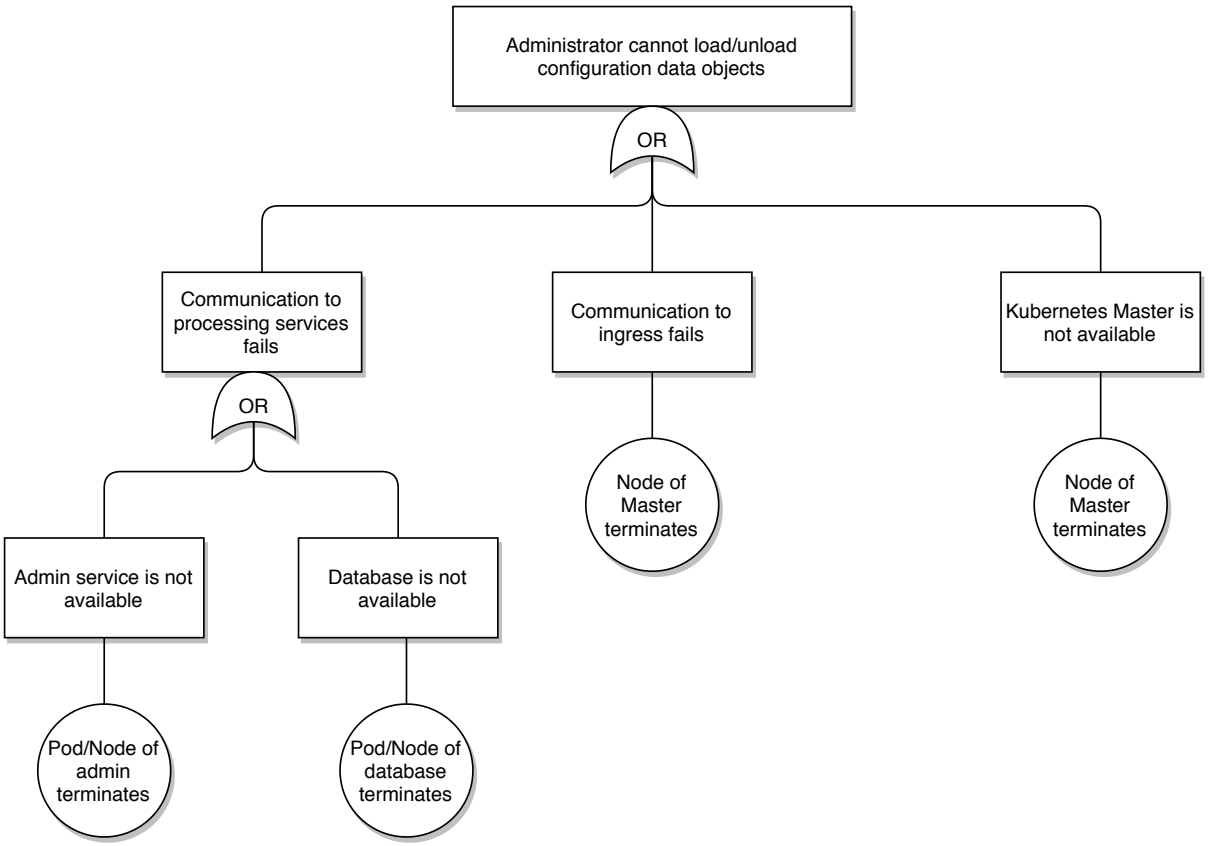

Figure B.15.: Fault Tree Analysis Results: Administrator cannot load/unload configuration data objects

B.3. Computational Hazard and Operations Study

No.	Interconnection and Flow	Attribute	Guideword	Cause	Consequences/Implication
1	Ingress-Registry (→)	data flow	none	request was aborted, request timed out	**Consequences:** • Registry does not receive a request • Ingress does not receive any data • Ingress may start a retry • User waits for response • Registry is overloaded with retries
2	Ingress-Registry (→)	data flow	other than	Data content of response from registry is removed	**Consequences:** • Ingress receives empty response • User does not receive any data
3	Ingress-Registry (→)	data flow	other than	Data content of response from registry is changed to random config objects	**Consequences:** • User receives data that he has not asked for • UX is reduced • User has to request data again
4	Ingress-Registry (→)	data flow	other than	http method is not permitted, header information is wrong, no controller for given URI	**Consequences:** • Registry cannot handle request • Registry returns an error • User receives no data
5	Ingress-Registry (→)	data flow	less than	In case of GET requests that ask for single config objects a config object identifier is left out/deleted	**Consequences:** • Registry returns an error since object identifier is missing • User receives no data
6	Ingress-Registry (→)	data flow	more than	In addition to a single identifier an array of identifiers is passed to a GET request	**Consequences:** • Registry does not provide a controller to handle multiple identifier requests • Registry returns an error • User receives no data
7	Ingress-Registry (→)	data flow	late	request is delayed, request is aborted and a retry is started	**Consequences:** • Increased response time • User may abort the process

Figure B.16.: Computational Hazard and Operations Results: Ingress-Registry

No.	Interconnection and Flow	Attribute	Guideword	Cause	Consequences/Implication
1	Registry-Master (→)	data flow	none	request was aborted, request timed out	**Consequences:** • Kubernetes Master does not receive a request • Registry does not receive data from Kubernetes Master • User does not receive any data
2	Registry-Master (→)	data flow	part of	Pod/Clusternode of registry terminates	**Consequences:** • Kubernetes Master does not receive a request • Registry doesn't know about previous requests • User will not receive any response • Registry does not initiate a retry
3	Registry-Master (→)	data flow	as well as	request was aborted, request timed out and a retry is started	**Consequences:** • Kubernetes Master receives multiple requests of same session • Kubernetes Master may shut down due to overload • User does not receive any data • Kubernetes Master is no longer available
4	Registry-Master (→)	data flow	late	Request is delayed	**Consequences:** • User does not receive any data • Increased response time • User may abort the process
5	Master-Registry (←)	data flow	less than	Registry receives less objects that are loaded than objects that are actually loaded	**Consequences:** • User receives less data than expected UX is reduced
6	Master-Registry (←)	data flow	more than	Registry receives more objects that are loaded than objects that are actually loaded	**Consequences:** • Further requests of the user may contain id's that are not present • UX is reduced • User does not receive the data intended on further requests

Figure B.17.: Computational Hazard and Operations Results: Registry-Master 1/2

7	Master-Registry (←)	data flow	none	Response was aborted/deleted/got lost	**Consequences:** • Registry does not receive data from Kubernetes Master • User does not receive any data • Registry blocks calling thread until response is received
8	Master-Registry (←)	data flow	none	Pod/Clusternode of master terminates	**Consequences:** • Registry does not receive data from Kubernetes Master • User does not receive any data • Kubernetes master does not know about previous requests • Registry blocks calling thread until response is received

Figure B.18.: Computational Hazard and Operations Results: Registry-Master 2/2

No.	Interconnection and Flow	Attribute	Guideword	Cause	Consequences/Implication
1	Master-Data Management (→)	data flow	none	request was aborted, request timed out	**Consequences:** • Data management does not receive a request/query • Data is not queried from the database • User may not receive any data besides the cached files • Kubernetes Master may block calling thread until a response is received by Database
2	Master-Data Management (→)	data flow	less than	Config id is not provided with admin request to delete a config data object	**Consequences:** • Data management returns an error since no identifier is provided • Admin receives an error
3	Master-Data Management (→)	data flow	more than	An array of config identifiers is provided by the Admin that are to be deleted	**Consequences:** • Data management returns an error since no controller is available to handle multiple identifiers
4	Master-Data Management (→)	data flow	other than	The request is changed from a POST request to a DELETE request in terms of URI, http method and content. A random identifier is generated in terms of a DELETE request	**Consequences:** • A random config object is unintentionally deleted from the database • The admin does not know which config data is deleted • The data may not be available to the user anymore
5	Master-Data Management (→)	data flow	none	Pod/Clusternode of master terminates	**Consequences:** • Data management does not receive a request/query • Data is not queried from the database • User may not receive any data besides the cached files • Kubernetes Master does not initiate a retry • Kubernetes Master does not know anything about a request previous to crash

Figure B.19.: Computational Hazard and Operations Results: Master-Data management 1/2

6	Data Management-Master (←)	data flow	none	response was aborted, response timed out	**Consequences:** • Data management does not receive a request/query • Data is not queried from the database • User may not receive any data besides the cached files • Response times increases • UX is decreased and user may abort the process • Kubernetes Master may block calling thread until a response is received by Database
7	Data Management-Master (←)	data flow	none	Pod/Clusternode of data management terminates	**Consequences:** • User receives a late response • Response times are increased • UX decreases • Kubernetes Master may block calling thread until a response is received by Database
8	Data Management-Master (←)	data flow	late	request is delayed	**Consequences:** • User receives a late response • Response times are increased • UX decreases • Kubernetes Master may block calling thread until a response is received by Database

Figure B.20.: Computational Hazard and Operations Results: Master-Data management 2/2

No.	Interconnection and Flow	Attribute	Guideword	Cause	Consequences
1	Master-Config MS (→)	data flow	part of	One out of two redundant microservices is shut down	**Consequences:** • One microservice is still able to respond • Request may be redirected due to sudden shutdown of requested service
2	Master-Config MS (→)	data flow	more than	All redundant services are shut down	**Consequences:** • Pods are restarted • Request is delayed by restart and retry time • Master may request the generation of additional services and request the data from the data management
3	Master-Config MS (→)	data flow	other than	The service is not loaded	**Consequences:** • The requested id is not loaded • The user can't load the requested data • The user may receive an error

Figure B.21.: Computational Hazard and Operations Results: Master-Config MS

No.	Interconnection and Flow	Attribute	Guideword	Cause	Consequences
1	Ingress-Adminservice (→)	data flow	none	Pod/Clusternode that hosts Admin terminates	**Consequences:** • Admin is not able to perform requests • Configuration data cannot be created/deleted or loaded
2	Ingress-Adminservice (→)	data flow	other than	http request contains a http method that is not allowed on the admin API	**Consequences:** • Admin receives a http status code 405 which signifies that the http method is not allowed
3	Ingress-Adminservice (→)	data flow	more than	In case of delete requests, more than one identifier is provided	**Consequences:** • Admin service does not provide a controller for multiple identifiers • Admin will receive an error message
4	Ingress-Adminservice (→)	data flow	less than	In case of delete requests, an identifier is not provided	**Consequences:** • Admin service does not provide a controller for multiple identifiers • Admin will receive an error message
5	Adminservice-Ingress (←)	data flow	none	Pod/Clusternode that hosts Ingress terminates	**Consequences:** • Admin does not receive a response • Admin has to re-issue his request

Figure B.22.: Computational Hazard and Operations Results: Ingress-Adminservice

No.	Interconnection and Flow	Attribute	Guideword	Cause	Consequences
1	Ingress-Userservice (→)	data flow	none	Pod/Clusternode that hosts Userservice terminates	**Consequences:** • User is not able to perform requests • UX is decreased
2	Ingress-Userservice (→)	data flow	other than	http request contains a http method that is not allowed on the user API (get, delete)	**Consequences:** • User receives an http status code 405 which signifies that the http method is not allowed
3	Ingress-Userservice (→)	data flow	more than	http request provides more parameters than allowed	**Consequences:** • User receives an error that additional parameters are not allowed
4	Userservice-Ingress (←)	data flow	none	Pod/Clusternode that hosts Ingress terminates	**Consequences:** • User does not receive data • UX is decreased

Figure B.23.: Computational Hazard and Operations Results: Ingress-Userservice

Appendix B

Bibliography

[ALR+01] A. Avizienis, J.-C. Laprie, B. Randell, et al. *Fundamental concepts of dependability*. University of Newcastle upon Tyne, Computing Science, 2001 (cit. on p. 24).

[AOT10] H. Arabian-Hoseynabadi, H. Oraee, P. Tavner. "Failure modes and effects analysis (FMEA) for wind turbines." In: *International Journal of Electrical Power & Energy Systems* 32.7 (2010), pp. 817–824 (cit. on p. 8).

[ARH15] P. Alvaro, J. Rosen, J. M. Hellerstein. "Lineage-driven fault injection." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 331–346 (cit. on pp. 42, 45).

[BBR+16] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, C. Rosenthal. "Chaos Engineering." In: *IEEE Software* 33.3 (2016), pp. 35–41 (cit. on pp. 1, 2, 26–31, 39–41, 43).

[BCK03] L. Bass, P. Clements, R. Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003 (cit. on pp. 5, 6).

[Ben13] G. Bengel. *Verteilte Systeme: Client-Server-Computing für Studenten und Praktiker*. Viewg-Lehrbuch. Vieweg+Teubner Verlag, 2013. ISBN: 9783322969156. URL: https://books.google.de/books?id=xOYkBgAAQBAJ (cit. on p. 66).

[BGM87] I. Benbasat, D. K. Goldstein, M. Mead. "The case research strategy in studies of information systems." In: *MIS quarterly* (1987), pp. 369–386 (cit. on p. 12).

[Dat08] C. S. Date. "ReSIST: Resilience for Survivability in IST." In: *Interaction* (2008), p. 1 (cit. on p. 26).

[DFVA10] J. Dunjó, V. Fthenakis, J. A. Vılchez, J. Arnaldos. “Hazard and operability (HAZOP) analysis. A literature review.” In: *Journal of hazardous materials* 173.1-3 (2010), pp. 19–32 (cit. on p. 11).

[Eri99] C. A. Ericson. “Fault tree analysis.” In: *System Safety Conference, Orlando, Florida*. Vol. 1. 1999, pp. 1–9 (cit. on p. 7).

[FF98] D. Farace, J. Frantzen. *GL'97 Conference Proceedings: Third International Conference on Grey Literature: Perspectives on the design and transfer of scientific and technical information, Luxembourg DGXIII, November 13-14, 1997.-Grey Literature Network Service.-Amsterdam: GreyNet/TransAtlantic, March 1998. X, 294 p.; 30 cm.-(GL-conference series, Conference proceedings, ISSN 1386-2316; No. 3)*. Amsterdam: GreyNet/TransAtlantic, 1998 (cit. on p. 39).

[FH02] A. Field, G. Hole. *How to design and report experiments*. Sage, 2002 (cit. on pp. 3, 16, 99).

[FH94] P. Fenelon, B. Hebbron. “Applying HAZOP to software engineering models.” In: *Risk management and critical protective systems: proceedings of SARSS*. 1994, pp. 11–116 (cit. on pp. 10, 82).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer, 2014 (cit. on pp. 7, 65).

[FT00] R. T. Fielding, R. N. Taylor. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Doctoral dissertation, 2000 (cit. on pp. 6, 65, 66, 129).

[Gra93] J. Gray. *The Benchmark Handbook for Database and Transaction Systems*. 2nd ed. Morgan Kaufmann, 1993 (cit. on p. 27).

[HADP18] A. van Hoorn, A. Aleti, T. F. Düllmann, T. Pitakrat. “ORCAS: Efficient Resilience Benchmarking of Microservice Architectures.” In: *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE. 2018, pp. 146–147 (cit. on pp. 27, 42).

[HKD14] N. Hyett, A. Kenny, V. Dickson-Swift. “Methodology or method? A critical review of qualitative case study reports.” In: *International journal of qualitative studies on health and well-being* 9.1 (2014), p. 23606 (cit. on p. 47).

[HRJ+16] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, V. Sekar. “Gremlin: Systematic resilience testing of microservices.” In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2016, pp. 57–66 (cit. on p. 43).

[HTI97] M.-C. Hsueh, T. K. Tsai, R. K. Iyer. “Fault injection techniques and tools.” In: *Computer* 30.4 (1997), pp. 75–82 (cit. on p. 25).

[Jan08] I. L. Janis. “Groupthink.” In: *IEEE Engineering Management Review* 36.1 (2008), p. 36 (cit. on p. 14).

[KC14] R. A. Krueger, M. A. Casey. *Focus groups: A practical guide for applied research*. Sage publications, 2014 (cit. on p. 14).

[LGTL85] W.-S. Lee, D. L. Grosh, F. A. Tillman, C. H. Lie. “Fault Tree Analysis, Methods, and Applications A Review.” In: *IEEE transactions on reliability* 34.3 (1985), pp. 194–203 (cit. on pp. 9, 10).

[LH83] N. G. Leveson, P. R. Harvey. “Analyzing software safety.” In: *IEEE Transactions on Software Engineering* 5 (1983), pp. 569–579 (cit. on pp. 8, 9).

[Mar02] R. C. Martin. “The single responsibility principle.” In: *The principles, patterns, and practices of Agile Software Development* 149 (2002), p. 154 (cit. on p. 6).

[Mar17] R. C. Martin. *Clean architecture: a craftsman’s guide to software structure and design*. Prentice Hall Press, 2017 (cit. on p. 5).

[MEH01] M. W. Maier, D. Emery, R. Hilliard. “Software architecture: Introducing IEEE standard 1471.” In: *Computer* 4 (2001), pp. 107–109 (cit. on p. 6).

[New15] S. Newman. *Building Microservices*. O’Reilly Media Inc., 2015, pp. 1–2 (cit. on pp. 6, 66).

[ORe17] G. O’Regan. “Software Reliability and Dependability.” In: *Concise Guide to Software Engineering*. Springer, 2017, pp. 171–184 (cit. on pp. 1, 24).

[PG95] W. Pree, E. Gamma. *Design patterns for object-oriented software development*. Vol. 183. Addison-wesley Reading, MA, 1995 (cit. on p. 7).

[Poh10] K. Pohl. *Requirements engineering: fundamentals, principles, and techniques*. Springer Publishing Company, Incorporated, 2010 (cit. on pp. 13–15, 50).

[Pul01] L. L. Pullum. *Software fault tolerance techniques and implementation*. Artech House, 2001 (cit. on p. 25).

[PVB+19] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. von Kistowski, A. Ali-Eldin, C. L.Abad, J. N. Amaral, P. Tuma. *Methodological Principles for Reproducible Performance Evaluation in Cloud Computing*. Tech. rep. SPEC RG Cloud Working Group, 2019 (cit. on p. 89).

[PW92] D. E. Perry, A. L. Wolf. “Foundations for the study of software architecture.” In: *ACM SIGSOFT Software engineering notes* 17.4 (1992), pp. 40–52 (cit. on p. 5).

[Ran] Random House. *Random House Kernerman Webster's College Dictionary, © 2010 K Dictionaries Ltd. Copyright 2005, 1997, 1991 by Random House, Inc.* Random House Inc. (cit. on p. 23).

[RH09] P. Runeson, M. Höst. "Guidelines for conducting and reporting case study research in software engineering." In: *Empirical software engineering* 14.2 (2009), p. 131 (cit. on pp. 3, 11, 12, 15, 47, 50, 51, 54).

[RM16] C. Robson, K. McCartan. *Real world research*. John Wiley & Sons, 2016 (cit. on p. 11).

[Sah11] G. K. Saha. "Software Fault Avoidance." In: *IEEE Reliability Society Newsletter Vol. 57* (2011) (cit. on p. 25).

[SC98] A. Strauss, J. Corbin. *Basics of qualitative research techniques*. Sage publications Thousand Oaks, CA, 1998 (cit. on p. 13).

[SI03] J. R. Seung, K. Ishii. *Using cost based FMEA to enhance reliability and serviceability*. Elsevier B. V., 2003 (cit. on p. 25).

[Str12] L. Strigini. "Fault tolerance and resilience: meanings, measures and assessment." In: *Resilience assessment and evaluation of computing systems*. Springer, 2012, pp. 3–24 (cit. on p. 25).

[TC04] P. C. Teoh, K. Case. "Failure modes and effects analysis through knowledge modelling." In: *Journal of Materials Processing Technology* 153 (2004), pp. 253–260 (cit. on p. 7).

[VWAM12] M. Vieira, K. Wolter, A. Avritzer, A. van Moorsel. *Resilience Assessment and Evaluation of Computing Systems*. Springer Science Business Media, 2012, 2012 (cit. on pp. 1, 19, 23, 25, 27).

[Wig11] A. Wiggins. "The twelve-factor app." In: *The Twelve-Factor App* (2011) (cit. on p. 68).

[WJG01] R. Winther, O.-A. Johnsen, B. A. Gran. "Security assessments of safety critical systems using HAZOPs." In: *SAFECOMP: International Conference on Computer Safety, Reliability, and Security*. Springer. 2001, pp. 14–24 (cit. on pp. 10, 11).

[WRH+12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012 (cit. on pp. 3, 11–15, 47, 53, 56).

[Yin98] R. K. Yin. "The abridged version of case study research: Design and method." In: (1998) (cit. on p. 11).

Appendix B

Online References

[Chaa] ChaosToolkit. URL: https://chaostoolkit.org (cit. on pp. 2, 20, 34).

[Chab] ChaosToolkit. URL: https://docs.chaostoolkit.org/reference/api/experiment/ (cit. on pp. 20, 91).

[Cha19] ChaosIQ. *ChaosIQ*. 2019. URL: https://chaosiq.io/ (cit. on p. 20).

[Doc] Docker Inc. URL: https://www.docker.com (cit. on p. 18).

[Fei18] N. V. Feito Matias. *On the lessons learnt by approaching chaos and resilience engineering in a midsize organisation*. 2018 (cit. on p. 44).

[Fou] T. L. Foundation. URL: https://kubernetes.io/ (cit. on pp. 18, 19, 62, 66).

[Grea] Gremlin Inc. *Chaos Conf*. Gremlin Inc. URL: https://chaosconf.splashthat.com (cit. on p. 41).

[Greb] Gremlin Inc. 2018. Gremlin Inc. 2018. URL: https://help.gremlin.com (cit. on pp. 1, 29, 32, 41, 42).

[Grec] Gremlin Inc. 2018. Gremlin Inc. 2018. URL: https://https://www.gremlin.com/community/tutorials/how-to-run-a-gameday/ (cit. on pp. 36, 37).

[Hoc] L. Hochstein. *Resources on Resilience Engineering*. URL: https://github.com/lorin/resilience-engineering (cit. on p. 42).

[IBM] IBM. *IBM WebSphere Application Server*. IBM. URL: https://www.ibm.com/de-de/marketplace/java-ee-runtime/details (cit. on p. 59).

[IT11] Y. Izrailevsky, A. Tseitlin. *The Netflix Simian Army*. 2011. URL: https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116 (cit. on pp. 29–31, 42).

[Jon] N. Jones. *Performing Chaos at Netflix Scale*. URL: https://www.youtube.com/watch?v=LaKGx0dAUlo (cit. on p. 43).

[Led17] A. Ledenev. *Chaos Testing for Docker Container*. 2017. URL: https://alexei-led.github.io/post/pumba_containercamp/ (cit. on pp. 33, 44).

[Med18] A. Medina. *Getting Started with Chaos Engineering*. 2018 (cit. on pp. 26, 32).

[Mica] Microsoft. Microsoft. URL: https://azure.microsoft.com/de-de/ (cit. on p. 90).

[Micb] Microsoft. Microsoft. URL: https://azure.microsoft.com/de-de/services/kubernetes-service/ (cit. on p. 90).

[Mil18] R. Miles. *Chaos Engineering Case Study : HomeAway*. ChaosIQ, 2018. URL: https://medium.com/chaosiq/chaos-engineering-case-study-homeaway-f3fa26b5ede5 (cit. on pp. 44, 45).

[Mon] MongoDB Inc. MongoDB Inc. URL: https://www.mongodb.com (cit. on p. 60).

[Net14] Netflix Tech Blog. Netflix, 2014. URL: https://medium.com/netflix-techblog/fit-failure-injection-testing-35d8e2a9bb2 (cit. on p. 29).

[Obj] Object Management Group. *OMG Unified Modeling Language*. Object Management Group. URL: https://www.omg.org/spec/UML/2.5/PDF (cit. on p. 63).

[Oraa] Oracle. *Java*. Oracle. URL: https://java.com/de/about/whatis_java.jsp (cit. on p. 7).

[Orab] Oracle. *Java EE*. Oracle. URL: https://www.oracle.com/technetwork/java/javaee/overview/index.html (cit. on p. 59).

[Orac] Oracle. *Java Platform Standard Edition 7 Documentation*. Oracle. URL: https://docs.oracle.com/javase/7/docs/api/java/sql/Blob.html (cit. on p. 60).

[Pat] T. Patric. URL: https://github.com/titpetric/docker-chaos-monkey (cit. on p. 34).

[Piva] Pivotal Software. URL: https://spring.io/ (cit. on p. 17).

[Pivb] Pivotal Software Inc. *Spring Boot Documentation*. Pivotal Software Inc. URL: https://docs.spring.io/spring-boot/docs/current/reference/html/_learning_about_spring_boot_features.html (cit. on p. 67).

[R F] R Foundation. R Foundation. URL: https://www.r-project.org/ (cit. on p. 99).

[Ric17] C. Richardson. *Microservice patterns*. 2017 (cit. on p. 6).

[Sob] A. Sobti. URL: https://github.com/asobti/kube-monkey (cit. on p. 36).

[Sys] Systems Safety. Department of Defense. URL: https://www.system-safety.org/Documents/MIL-STD-882E.pdf (cit. on pp. 7, 69).

[Thea] The Apache Software Foundation. The Apache Software Foundation. URL: http://cassandra.apache.org (cit. on p. 60).

[Theb] The Apache Software Foundation. The Apache Software Foundation. URL: http://couchdb.apache.org (cit. on p. 60).

[Thec] The PostgresSQL Global Development Group. *PostgresSQL Documentation*. The PostgresSQL Global Development Group. URL: https://www.postgresql.org/about/ (cit. on p. 61).

[The19a] The Apache Software Foundation. *Apache JMeter*. 2019. URL: https://jmeter.apache.org/ (cit. on p. 19).

[The19b] The Apache Software Foundation. *Apache JMeter User Manual – Test Plan*. 2019. URL: https://jmeter.apache.org/usermanual/test_plan.html (cit. on p. 19).

[Wil18] B. Wilms. *Chaos Monkey for Spring Boot Reference Guide*. 2018. URL: https://codecentric.github.io/chaos-monkey-spring-boot/2.0.1/ (cit. on pp. 35, 88, 121).

All links were last followed on August 22, 2019.

Declaration

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

---

place, date, signature