HAW HAMBURG

MASTER THESIS
Ante Škorić

Towards Zero-Downtime Distributed Systems: Intelligent Error Detection and Automated Recovery

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Ante Škorić

Towards Zero-Downtime Distributed Systems: Intelligent Error Detection and Automated Recovery

Master thesis submitted for examination in Master´s degree
in the study course *Master of Science Informatik*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Stefan Sarstedt
Supervisor: Prof. Dr. Marina Tropmann-Frick

Submitted on: 10.04.2025

Ante Škorić

Thema der Arbeit

Towards Zero-Downtime Distributed Systems: Intelligent Error Detection and Automated Recovery

Stichworte

Microservice-Systeme, Verteilte Systeme, Chaos Engineering, Lasttests, MAPE-K-Modell, Maschinelles Lernen, Resilienz, Kubernetes, Spring Boot, XGBoost, Self-Healing, Klassifikation, Resilienzstrategien, Systemzuverlässigkeit

Kurzzusammenfassung

Microservice-Systeme sind aufgrund ihrer verteilten Natur und dynamischen Umgebungen anfällig für Ausfälle. Diese Arbeit präsentiert den Entwurf, die Implementierung und die Evaluierung eines Tools, das Chaos Engineering, Lasttests, das MAPE-K-Modell und maschinelles Lernen integriert, um die Systemresilienz zu verbessern. Das Tool wurde an zwei Microservice-Architekturen – dem Student Management System (SMS) und dem E-Commerce Order Management System (EOMS) – über zwölf Fehlerszenarien hinweg evaluiert, wobei die Fehlerraten mit und ohne sein Eingreifen verglichen wurden. Die experimentellen Ergebnisse zeigen eine allgemeine Fehlerreduktion von 57,96 %, mit einer signifikanten Verbesserung von 95 % bei der Abschwächung Kubernetes-bezogener Fehler. Trotz seiner Effektivität wies das Tool aufgrund von Ungenauigkeiten bei der Klassifizierung Einschränkungen bei der Behebung bestimmter Spring-bezogener Fehler auf. Durch die Nutzung von XGBoost für die prädiktive Fehlerklassifizierung sowie automatisierte Strategien zur Fehlerbegrenzung zeigt das Tool Potenzial für Self-Healing-Microservices. Zukünftige Forschungsarbeiten sollten sich auf die Verbesserung der Klassifizierungsgenauigkeit, die Erweiterung der Resilienzstrategien und die Optimierung der Tool-Architektur für eine höhere Zuverlässigkeit konzentrieren...

Ante Škorić

Title of Thesis

Towards Zero-Downtime Distributed Systems: Intelligent Error Detection and Automated Recovery

Keywords

Microservice systems, Distributed systems, Chaos engineering, Load testing, MAPE-K model, Machine learning, Resilience, Kubernetes, Spring Boot, XGBoost, Self-healing, Classification, Resilience strategies, System reliability

Abstract

Microservice systems are prone to failures due to their distributed nature and dynamic environments. This thesis introduces the design, implementation, and evaluation of a tool that integrates chaos engineering, load testing, the MAPE-K model, and machine learning to enhance system resilience. The tool was evaluated on two microservice architectures—the Student Management System (SMS) and the E-commerce Order Management System (EOMS)—across twelve failure scenarios, assessing error rates with and without its intervention. Experimental results indicate an overall error reduction of 57.96%, with a significant 95% improvement in mitigating Kubernetes-related failures. Despite its effectiveness, the tool exhibited limitations in addressing certain Spring-related errors due to classification inaccuracies. By leveraging XGBoost for predictive error classification and automated failure mitigation strategies, the tool demonstrates the potential for self-healing microservices. Future research should focus on refining classification accuracy, expanding resilience strategies, and optimizing the tool's architecture for improved reliability...

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation, Objectives and Research Questions

Reliability, operational efficiency, and minimizing downtime are crucial for all distributed systems, including those based on microservice architectures. The ability of microservices to recover from failures and continue functioning is essential for cloud providers [46]. Various resilience patterns, such as retry, fail-fast, and circuit breaker can enhance microservice robustness. Research has also explored sophisticated techniques like model checking for these patterns [32]. This research prompts the question of whether self-healing mechanisms can automatically apply these patterns within systems.

Studies in self-healing systems highlight the importance of efficient debugging for maintaining reliable, high-quality software. As software complexity grows, the likelihood of failure increases [47].
The 2024 survey ″A Survey on Self-healing Software System″ indicates limited research on self-healing systems that perform architectural-level repairs [47].

The objective of this thesis is to leverage self-healing and self-adaptive methodologies to enhance microservice architecture stability and resilience, aiming for zero downtime. The majority of self-adaptive system solutions employ the MAPE-K reference model for implementation. Example of those implementations are ″A Self Healing Microservices Architecture: A Case Study in Docker Swarm Cluster″ [29], ″Self-adaptation in Microservice Architectures: A Case Study″ [11], ″Self-adaptive, Requirements-driven Autoscaling of Microservices″ [24] and ″A MAPE-K Approach to Autonomic Microservices″ [12]. These studies demonstrate the MAPE-K model's effectiveness in self-healing systems but do not integrate XGBoost for error prediction or chaos engineering for testing.

This thesis seeks to bridge this gap by developing a tool that combines the MAPE-K model with XGBoost for log classification and chaos engineering for testing. This

approach enables the testing of systems in a more realistic manner, as detailed in the paper titled ″CHESS: A Framework for Evaluation of Self-adaptive Systems based on Chaos Engineering″, which evaluates system resilience and fault recovery capabilities.
Two systems will be utilized for testing: the Student Management System (SMS) and the E-commerce Order Management System (EOMS).
The analytical component of the MAPE-K approach has already been implemented and analyzed in a previous project for my computer science studies, titled ″Classifier Development for Log Analysis in Spring Boot and Kubernetes″ [40]. This component will be further discussed in the Classification Algorithms section of the Fundamentals chapter.

The central research question of this thesis is as follows: ″Can the combination of the MAPE-K reference model, XGBoost-driven error prediction, and chaos engineering testing be utilized to implement a self-healing system that significantly improves the reliability and resilience of microservice architectures?″.

1.2 Structure

This thesis is structured into eight chapters.
Chapter one outlines the motivation, objectives, and research questions that form the foundation of this work.
Chapter two provides the theoretical background necessary to understand the subsequent chapters.
A review of related work in the field of self-healing systems and the MAPE-K framework follows in chapter three. Chapter four introduces the conceptual framework, including the problem definition and requirements analysis.
The implementation of the proposed tool is described in chapter five, beginning with an overview of the system architecture and the rationale behind architectural decisions, followed by an explanation of the integration of MAPE-K, XGBoost, and decision trees.
Experimental methodology, including setup and implementation, is covered in chapter six.
Chapter seven presents and analyzes the experimental results, concluding with a discussion of the findings.
Finally, chapter eight summarizes the key contributions of this thesis, provides a conclusion, and offers an outlook on potential future research directions.

2 Fundamentals

2.1 MAPE-K

In 2003, IBM published a paper in the journal Computer (Volume: 36, Issue: 1) titled ″The vision of autonomic computing″ [25]. This publication introduced the MAPE-K control loop.
In the fourth edition of their white paper ″An architectural blueprint for autonomic computing″ IBM states:
″This paper has presented a high-level architectural blueprint to assist in delivering autonomic computing in phases. The architecture reinforces that self-management uses intelligent control loop implementations to monitor, analyze, plan and execute, leveraging knowledge of the environment. These control loops can be embedded in resource run-time environments (in the form of self-managing resources) or delivered in management tools. The control loops collaborate using an enterprise service bus (one of the five architectural building blocks) that integrates the remaining four architectural building blocks: autonomic managers, manual managers, manageability endpoints, and knowledge sources.″ [16]

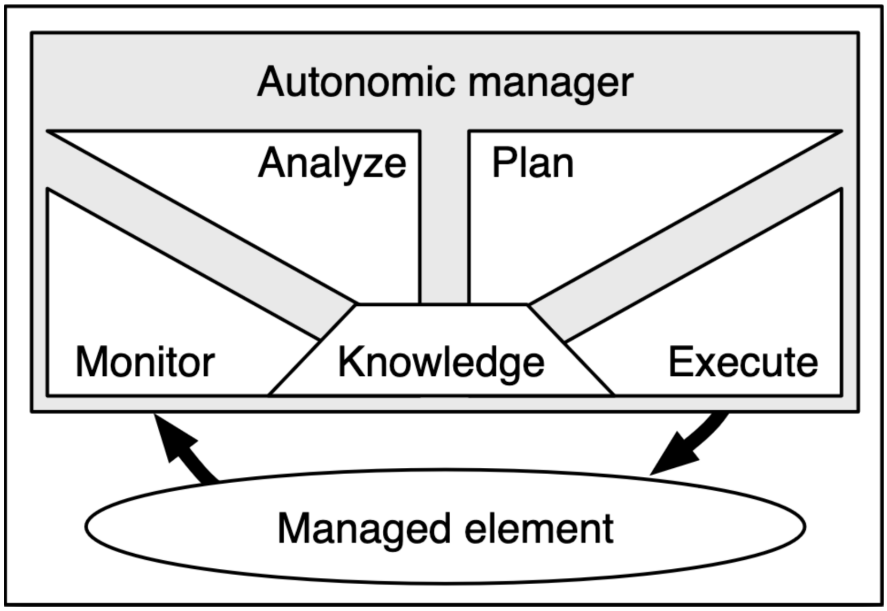

Figure 2.1: The MAPE-K self-adaptive cycle as proposed by IBM [25], figure taken from paper ″Achieving Cost-Effective Software Reliability Through Self-Healing″ [20]

The figure 2.1 illustrates the MAPE-K control loop model reference. This model is utilized for self-adaptive systems and autonomic computing. IBM introduced it to aid in the design and development of self-managing systems [16].
The K in MAPE-K represents Knowledge, serving as the central repository that stores data, metrics, policies, and topology information. This knowledge base is shared among all other components.
M stands for Monitor, which collects data from the managed system or environment, gathering information such as system metrics, performance data, and configuration properties. It subsequently transmits relevant data to the Analyze component.
A denotes Analyze, which processes the monitored data to detect patterns and identify potential issues. It evaluates whether modifications are necessary based on the system's current state and, if required, triggers the Plan component.
P represents Plan, which formulates an action strategy when adaptations are deemed necessary. It also generates a sequence of actions to be executed.
Finally, E stands for Execute, which is responsible for carrying out the modifications specified in the action plan [16].

2.2 Microservices

Martin Fowler and James Lewis define the microservice architecture as follows:
″In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.″ [28]
In a microservices architecture, services are developed within clearly delineated business boundaries.
Each service is responsible for a distinct functionality, ensuring that the codebase remains manageable and does not grow excessively large [35].
Microservices offer several advantages over traditional monolithic software architectures:

- Resilience – The architecture promotes fault isolation. If a service encounters an issue or fails, the failure remains contained within that specific service, preventing disruptions across the entire system [35].
- Independent Deployments - Each service can be deployed independently, allowing for greater flexibility and reducing the risk associated with system-wide updates [35].
- Technology Diversity - Since services are developed autonomously, different technologies and programming languages can be employed to best meet the specific requirements of each service [35].

However, microservices also introduce several challenges:

- Testing Overhead - Testing microservices is inherently more complex than testing a monolithic application due to the need to validate interactions between multiple independent services [35].
- Monitoring Complexity - Given the distributed nature of microservices, comprehensive monitoring is essential to maintain system health and detect potential issues [35].

2.3 Container Orchestration

The container orchestration in the paper ″Container Orchestration: A Survey″ is defined as: ″Container orchestration allows cloud and application providers to define how to select, to deploy, to monitor, and to dynamically control the configuration of multi-container packaged applications in the cloud. Container orchestration is concerned with the management at runtime to support the deploy, run, and maintain phases. Container orchestrator usually offers the following main features: resource limit control, scheduling, load balancing, health check, fault tolerance, and autoscaling.″ [13]
In this thesis, the Kubernetes orchestration tool is utilized.
″Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.″ [26]
To operate Kubernetes, a cluster is deployed, consisting of worker machines referred to as nodes. Nodes execute pods, which serve as hosts for containers. The Control Plane oversees cluster management, providing an API for deploying and managing containers. Kubernetes objects, defined in YAML, specify applications, resources, and policies to ensure the cluster maintains its desired state [26].

2.4 Performance Testing

Performance testing is defined in the AWS documentation as follows: ″Determines the responsiveness and stability of a system as it performs under a particular workload. Performance testing also is used to investigate, measure, validate, or verify other quality attributes of the system, such as scalability, reliability, and resource usage. Types of performance tests might include load tests, stress tests, and spike tests. Performance tests are used for benchmarking against predefined criteria.″ [8] Another critical type of testing relevant to this work is load testing. According to the AWS documentation: ″Load tests are done to gain reliable information on whether your application is delivering the expected qualities. Although the most common approach is to generate load on your applications, there are different ways you can understand load testing.″ [17]
Performance testing is a fundamental process during various stages of the software development lifecycle, particularly before an application is deployed into production.
It serves the following purposes:

- Ensuring Reliability and Stability – Regular performance testing verifies that the application can consistently handle expected workloads without experiencing performance degradation [6].
- Validating Scalability – Testing assesses whether the application can scale efficiently while maintaining performance standards as user demand increases [7].
- Optimizing Resource Utilization – Performance testing evaluates the efficiency of resource usage, including CPU, memory, and network bandwidth, ensuring cost-effective and optimal system operations [5].

2.5 Chaos Engineering

Netflix defines the principles of Chaos Engineering on its website ″Principles of Chaos Engineering″ as:
″Chaos Engineering is the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production.″ [14]
Modern distributed systems enable flexible development and rapid deployment; however, they also introduce complexity and uncertainty. Even if individual services function correctly, their interactions can lead to unpredictable failures, particularly during rare real-world disruptions. Identifying potential weaknesses at an early stage is crucial, as issues such as improper fallback mechanisms, retry storms, or cascading failures can significantly impact system stability. To ensure resilience and maintain confidence in production environments, it is essential to proactively uncover and mitigate these vulnerabilities. Chaos Engineering addresses this challenge by conducting controlled experiments on systems to analyze their behavior under stress, thereby enabling teams to develop more robust and reliable distributed architectures [14]. These experiments generally follow four key steps:

- Define the system's steady state—a measurable indicator of normal operational behavior [14].
- Formulate a hypothesis that the steady state will remain unchanged under both normal and experimental conditions [14].
- Introduce real-world failure scenarios, such as server crashes, hardware malfunctions, or network disruptions [14].

- Observe deviations from the steady state to identify system weaknesses [14].

The greater the difficulty in disrupting the steady state, the higher the confidence in the system's resilience.
Identifying vulnerabilities through these experiments highlights areas for improvement, allowing potential failures to be addressed before they escalate into critical production issues [14].

2.6 XGBoost

In the paper ″XGBoost: A Scalable Tree Boosting System″, XGBoost was introduced as an efficient and scalable gradient boosting framework [15]. ″In this paper, we described the lessons we learnt when building XGBoost, a scalable tree boosting system that is widely used by data scientists and provides state-of-the-art results on many problems.″ [15]
XGBoost incorporates several key features that enhance its performance:

- Sparsity-aware learning – Efficiently handles sparse data, a common characteristic of real-world datasets [15].
- Weighted quantile sketch – Facilitates approximate tree learning, improving computational efficiency and scalability [15].
- Cache optimization – Enhances data access patterns, reducing latency and accelerating computations [15].
- Data compression and sharding – Reduces memory consumption and distributes computational workloads, enabling the processing of large-scale datasets [15].

These features contribute to XGBoost's widespread adoption among data scientists for various machine learning tasks, including:

- Classification – Assigning categorical labels to data points [15].
- Regression – Predicting continuous numerical values [15].
- Ranking – Ordering items based on relevance or importance in a given context [15].

The effectiveness of XGBoost has been demonstrated across diverse applications, including web text classification, ad click-through rate prediction, and high-energy physics event classification [15].

3 Related Work

The thesis addresses an intersection of topics, including self-healing and self-adaptive systems, XGBoost, MAPE-K, and chaos engineering. This chapter reviews key contributions in those domains and papers in which the combination of the topics is discussed.

The paper ″A Survey on Self-healing Software System″ [47] provides a comprehensive survey on self-healing software systems, emphasizing the challenges posed by increasing software complexity and the necessity for systems capable of autonomous recovery from failures. The study categorizes various self-healing methods and highlights the importance of self-diagnosis and automatic repair mechanisms to maintain system reliability without human intervention.

In 2003 IBM released their paper called ″The vision of autonomic computing″ [25] which introduces the MAPE-K control loop. Prior to this, in 2001, IBM published a manifesto highlighting a looming software complexity crisis. This crisis was driven by the increasing scale and complexity of software systems, often consisting of tens of millions of lines of code, and the growing challenge of managing heterogeneous environments across enterprise-wide and internet-based systems. The manifesto emphasized that the difficulty of integrating and managing these systems posed a threat to continued progress in computing. As a response, autonomic computing was proposed as a promising approach to address this complexity. Autonomic systems aim to manage themselves based on high-level objectives set by administrators, reducing the need for manual intervention. These systems are envisioned to seamlessly integrate new components, much like a new cell integrating into a living organism. While ambitious, these concepts laid the foundation for the development of self-managing computing systems, representing a significant research challenge in the pursuit of reducing operational complexity [25].

Later, in 2006, IBM published a white paper titled ″An Architectural Blueprint for Autonomic Computing″ [16]. This document outlines a comprehensive framework for creating self-managing computing systems, drawing inspiration from the human body's autonomic

nervous system. The blueprint introduces key components such as the Autonomic Manager, which employs a control loop to monitor and adjust system operations, and Manageability Endpoints, which facilitate communication between resources and managers. By advocating for systems capable of self-configuration, self-optimization, self-healing, and self-protection, this work has significantly influenced subsequent research and development in autonomic and self-adaptive systems.

The paper ″Feedback Control as MAPE-K Loop in Autonomic Computing″ [38] examines the MAPE-K loop through the lens of control theory, exploring both continuous and discrete (supervisory) control techniques and their application in the feedback control of computing systems. It offers detailed interpretations of feedback control loops as MAPE-K loops, supported by various case studies.

Microservices architecture offers benefits like flexibility and scalability, with auto scaling being key to adjusting resources as needed. However, existing auto scaling solutions often misallocate resources, leading to inefficiencies [24]. To address this, MS-RA, a self-adaptive, requirements-driven auto scaling solution was proposed in the paper ″Self-adaptive, Requirements-driven Autoscaling of Microservices″ [24], which utilizes service-level objectives (SLOs) for real-time decision-making. Built on the MAPE-K self-adaptive loop, MS-RA has been evaluated using an open-source microservice-based application.

In the context of microservices, the development of self-healing architectures has been explored to enhance system robustness [29]. The paper ″A Self Healing Microservices Architecture: A Case Study in Docker Swarm Cluster″ [29] has examined the implementation of self-healing mechanisms within Docker Swarm clusters, demonstrating how microservices can autonomously detect and recover from failures, thereby improving overall system reliability.

In the following paper ″Self-Adaptive Microservice-based Systems - Landscape and Research Opportunities″[18] a systematic mapping of 21 primary studies was made. The idea was to understand the application of self-adaptation in microservice-based systems. Findings indicate that most studies focus on the monitoring phase (28.57%) of the adaptation control loop, emphasize self-healing properties (23.81%), employ reactive adaptation strategies (80.95%) at the system infrastructure level (47.62%), and utilize centralized approaches (38.10%).

Recent advancements have introduced frameworks that integrate machine learning techniques to enhance self-configuration and self-healing capabilities. One such framework

employs an extreme gradient boosting (XGBoost) classifier to predict and mitigate system anomalies, thereby improving overall system resilience [19]. In the paper ″Self-Configuration and Self-Healing Framework Using Extreme Gradient Boosting (XGBoost) Classifier for IoT-WSN″ [19] IoT traffic is categorized into various classes using the XGBoost classifier. During the self-configuration phase, IoT devices are allocated transmission slots and contention access periods based on their priority levels. In the self-healing phase, the system initiates localized route recovery when a node's residual power drops below a certain threshold or when a node moves beyond a predefined range. Simulation results demonstrate that this framework achieves a higher packet delivery ratio and reduced packet drops compared to existing methods, while also lowering computational costs.

The CHESS framework was introduced in the paper ″CHESS: A Framework for Evaluation of Self-adaptive Systems based on Chaos Engineering″[30]. The framework offers a structured approach to evaluating self-adaptive systems. By employing chaos engineering techniques, CHESS enables systematic fault injection and monitoring, providing insights into system behavior under adverse conditions and facilitating the enhancement of self-healing capabilities.

Collectively, these studies contribute to the advancement of self-healing and self-adaptive systems, offering diverse methodologies and frameworks to enhance the resilience and efficiency of modern software architectures.

4 Concept

This chapter outlines the scope of the master's thesis and defines the problem to be addressed, implemented, and validated. Additionally, it presents the requirements analysis, detailing the necessary criteria for the tool's implementation.

4.1 Problem Definition

As outlined in the introduction, this thesis focuses on developing a tool based on the MAPE-K reference model, enhanced with a classification algorithm, to analyze and resolve errors in Spring Boot services running on Kubernetes.
The MAPE-K reference model is chosen for its structured approach, dividing concerns into distinct phases—Monitoring, Analysis, Planning, and Execution—which improves the manageability and scalability of complex adaptive systems. This modularity makes it well-suited for the intended use case [29].

The tool's scope includes system monitoring at both the log level of Spring Boot services and the Kubernetes cluster. For error analysis, it must classify errors to ensure appropriate solutions can be applied. Solution execution will be determined based on environmental monitoring and system metrics, with fixes applied at either the Kubernetes cluster level or the Spring Boot service level. Additionally, the tool should support rollback functionality to revert any changes made to the system.
To validate its effectiveness, a chaos engineering approach will be employed, proactively identifying and addressing vulnerabilities. This will enhance the system's resilience and reliability in dynamic and unpredictable environments [30].

4.2 Requirements Analysis

As outlined in the problem definition chapter, the proposed tool is designed to address two distinct categories of errors: those occurring within a Spring Boot service and those arising at the Kubernetes infrastructure level.
To systematically determine which errors the tool should resolve and which corresponding solutions should be applied, a structured approach has been devised. This chapter presents the methodology used to identify and classify these errors and their respective solutions.

For errors related to Spring Boot services, an extensive review of the existing literature was conducted to identify the most prevalent issues and their potential resolutions. The most commonly encountered errors in Spring Boot services are as follows:

- Database connection error - Occurs when an application fails to establish or maintain a connection to its database service, disrupting data access operations [48].
- Configuration errors - ″are also one of the major causes of today's system failures. Many configuration issues manifest themselves in ways similar to software bugs such as crashes, hangs, silent failures. It leaves users clueless and forced to report to developers for technical support, wasting not only users' but also developers' precious time and effort. Unfortunately, unlike software bugs, many software developers take a much less active, responsible role in handling configuration errors because "they are users" faults.″ [45]
- Service down - Represents the complete unavailability of a system component, preventing it from processing requests or performing its designated functions [10].
- Request error - Occurs when a client sends a malformed, invalid, or incorrectly structured request to a service, resulting in rejection or improper processing. These errors typically manifest as client-side (4xx) HTTP status codes in web services [31].
- Third party service error - Occurs when an external service integrated into a system fails to operate correctly. This can result from various issues, such as the third-party service being unavailable, experiencing internal errors, or having compatibility issues with the primary system [34].
- Timeout error - Occurs when an operation fails to complete within its allocated time window, potentially leaving system state uncertain [33].

Following the identification of these errors, further research was conducted to determine effective mitigation strategies.
The following solutions were identified:

- Circuit Breaker pattern - ″is used to detect and handle service failures gracefully. It prevents a system from repeatedly attempting to invoke a failing service, which can lead to cascading failures. By temporarily stopping the invocation of a failing service, the Circuit Breaker pattern allows the system to recover and maintain stability. This pattern helps to mitigate the risk of system-wide outages by isolating failures and providing mechanisms to recover from them. The Circuit Breaker pattern often operates in three states: Closed, Open, and Half-Open.″ [39]
  This approach is employed to detect and manage service failures, effectively preventing cascading failures and ensuring system stability. As a result, it enhances overall system availability and reduces failure rates [39].

- Retry pattern - ″automatically re-attempts a failed operation a specified number of times before giving up. This pattern is particularly useful in transient failure scenarios where a temporary issue might resolve itself after a short delay. When implementing the Retry pattern, it is crucial to define the retry logic carefully, including the number of retry attempts and the delay between retries. Implementing exponential backoff, where the delay increases exponentially with each retry, can help reduce the load on the failing service and avoid overwhelming it with requests.″ [39]
  This mechanism automatically retries failed operations, increasing their likelihood of success. Consequently, it enhances the overall success rate of operations while reducing error occurrences [39].

- Fallback pattern - ″provides an alternative response or action when a service fails. This pattern ensures that the system can continue to function, albeit with reduced capability, even when some services are unavailable. Fallback mechanisms can vary from returning cached data or default responses to redirecting requests to alternative services. The choice of fallback strategy depends on the criticality of the service and the nature of the failure. For instance, in a shopping application, a fallback might provide a static″service unavailable″ page, while in a financial application, it might offer a default value or last known good data to maintain functionality.″ [39]
  This mechanism offers alternative responses during failures, allowing the system to

maintain functionality, albeit with reduced capabilities. As a result, it enhances overall system reliability and improves the user experience [39].

- Timeout pattern - ″specifies a maximum duration for a service request. If the request exceeds this duration, it is aborted, preventing long-running operations from consuming excessive resources and affecting system performance. Timeouts can be configured at various levels, including the client-side, server-side, and network level. Configuring appropriate timeout values is essential to balance between allowing sufficient time for legitimate operations and preventing excessive resource consumption or blocking.″ [39]
  This mechanism defines a maximum duration for service requests, preventing long-running operations from degrading system performance. As a result, it enhances response times and optimizes resource utilization [39].

- Graceful Degradation Strategy - ″is the ability of a system to maintain basic functionality even when some components or services are not fully operational. This might involve prioritizing essential services during high-load or failure scenarios.″ [9]

- Isolating the Service (Bulkhead pattern) - ″demonstrated its value in isolating different components of the system to prevent single points of failure from affecting unrelated services. By segmenting resources and managing them independently, the Bulkhead pattern ensured that failures in one part of the system did not propagate to other areas. This isolation not only improved system resilience but also enhanced overall availability. Our metrics revealed a significant increase in system availability, from 85% to 95%, indicating that the Bulkhead pattern was successful in containing failures and preventing them from impacting the entire application. This result underscores the importance of resource isolation in maintaining robust system performance under varying load conditions.″ [39]
  This approach isolates system components to prevent single points of failure, thereby enhancing system resilience and availability. Consequently, it improves overall fault isolation and ensures greater system stability [39].

- Failover pattern - Is a fault-tolerance mechanism that automatically switches to a redundant or standby system upon the failure of the primary system. This ensures continuous availability and reliability in distributed systems. In microservices architectures, implementing failover mechanisms is crucial for maintaining service continuity. Strategies such as active-passive and active-active configurations are commonly employed to achieve this [1].

- Exponential backoff - Is a strategy where the time between retry attempts increases exponentially after each failure. This approach helps manage retries for failed operations, reducing network congestion and enhancing system reliability [41].
- Pod deletion - This is one of the strategies that was taken over from the Kubernetes solutions, it will be described in more detail in the Kubernetes solutions part of this section.
- Load balancing - ″is the process of redistribution of workload in a distributed system like cloud computing ensuring no computing machine is overloaded, under-loaded or idle. Load balancing tries to speed up different constrained parameters like response time, execution time,system stability etc. thereby improving performance of cloud.″ [2]
- YML formatting - One more strategy that was taken over from the Kubernetes solutions, it will also be described in more detail in the Kubernetes solutions part of this section.
- Rollback config - One more strategy that was taken over from the Kubernetes solutions, it will also be described in more detail in the Kubernetes solutions part of this section.
- Client-Side Throttling - ″When a client detects that a significant portion of its recent requests have been rejected due to ″out of quota″ errors, it starts self-regulating and caps the amount of outgoing traffic it generates. Requests above the cap fail locally without even reaching the network″ [10]
- Request Queuing Mechanism - Allows a service to temporarily store incoming requests when a downstream service is unavailable. These queued requests are processed later when the downstream service is back online. This strategy ensures reliability and resilience, preventing data loss during service interruptions [41].

The second category of errors pertains to issues within the Kubernetes cluster.
Given that this domain is less extensively researched compared to microservice resilience, a different approach was employed to identify prevalent errors and their corresponding solutions.
Specifically, monitoring data from the company xChange Solutions [43], collected via Datadog, was analyzed to determine the most frequently occurring Kubernetes-related errors.

The analysis revealed the following common Kubernetes errors:

- Pod Unhealthy - Occurs when the readiness probe fails, indicating that the pod is not executing.
- Node problem - Happens when the node is not in a ready state.
- Network error - Occurs when the network is not ready or the CNI plugin is not initialized.
- Mount fail – Happens when the pod is in a FailedMount state.
- Low CPU or memory - Occurs when there is insufficient CPU or memory for the pod to run.
- Invalid image error – Happens when there is an invalid reference to the required container image.
- Configuration error – Arises due to misconfigured Kubernetes settings.
- Kubernetes failed – Encompasses instances where Kubernetes generates failed events.

Unlike Spring Boot service errors, for which established solutions exist, Kubernetes-related error resolution required a different methodology. Since limited research has been conducted in this area, solutions were identified by manually analyzing GitLab hotfixes, Kubernetes deployments and Datadog logs/events within the company.
For certain errors where no established solutions were found using this approach, small proof-of-concept (PoC) implementations were developed to explore viable fixes.

Based on this research, the following solutions were identified:

- YML formatting – Fixed Kubernetes configuration errors by reformatting the YAML file.
- Rollback configuration – Resolved issues by reverting to a previous version of the configuration.
- Check registry – Addressed failures related to inaccessible container registries.
- Verify image – Resolved invalid image reference issues.
- Pod CPU and memory limitations – Increased CPU and memory limits to address resource constraints.

- Node scaling – Allocated additional resources to nodes.
- Free disk space – Freed or expanded disk space to resolve storage-related failures.
- Pod deletion – Restarted pods to resolve transient failures.
- Enable CNI – Addressed network issues by enabling the CNI plugin.
- Fix storage class/access modes – Adjusted storage class and access modes to resolve mount failures.

Due to certain technical limitations and the complexity of error handling mechanisms, not all identified solutions were implemented in the final tool.
The rationale for selecting specific solutions, the implementation details, and the mapping between errors and their corresponding solutions will be discussed in the Solution Classification section of the Implementation chapter.

5 Implementation

This chapter details the implementation of the tool, including its architectural framework, the algorithms utilized, and the overall operational workflow.

5.1 Architecture

The architectural design was derived from a systematic analysis of core requirements, established through evaluation:

- Scalability - The tool must exhibit robust scalability to accommodate the potential expansion of the monitored system and efficiently process large volumes of data.
- Performance - Given the significant volume of logs requiring monitoring and classification, the tool must maintain high processing efficiency.
- Asynchronous - The tool must support asynchronous processing, as the monitored system generates extensive log data. Parallel processing is essential to ensure that logs do not queue, preventing delays in resolving errors during runtime.
- Decoupling - The tool must operate independently of the node in which the monitored service is running. As nodes may become unavailable, reliance on them would prevent the tool from addressing errors effectively. Furthermore, the system's services must not depend on the tool, although the tool may depend on the services.

To ensure scalability, the tool was developed as a stateless service, utilizing MongoDB for data storage. An event-driven approach was adopted for inter-service communication, leveraging a message broker. This approach also facilitated asynchronous processing, allowing multiple logs to be handled concurrently without interdependencies.

Python was selected as the implementation language due to its efficiency and built-in support for asynchronous programming [36]. To enhance processing speed, the XGBoost library was integrated directly into the tool as a module. In the initial version, XGBoost operated as a separate service, but performance constraints necessitated its direct integration. To achieve decoupling, the tool operates externally to the service's execution node, ensuring continued functionality even if a node becomes unavailable. Dependency minimization was a critical objective, leading to the incorporation of Spring Boot's dynamic configuration, which will be elaborated upon later.

Considering these requirements, the following architecture 5.1 was designed:

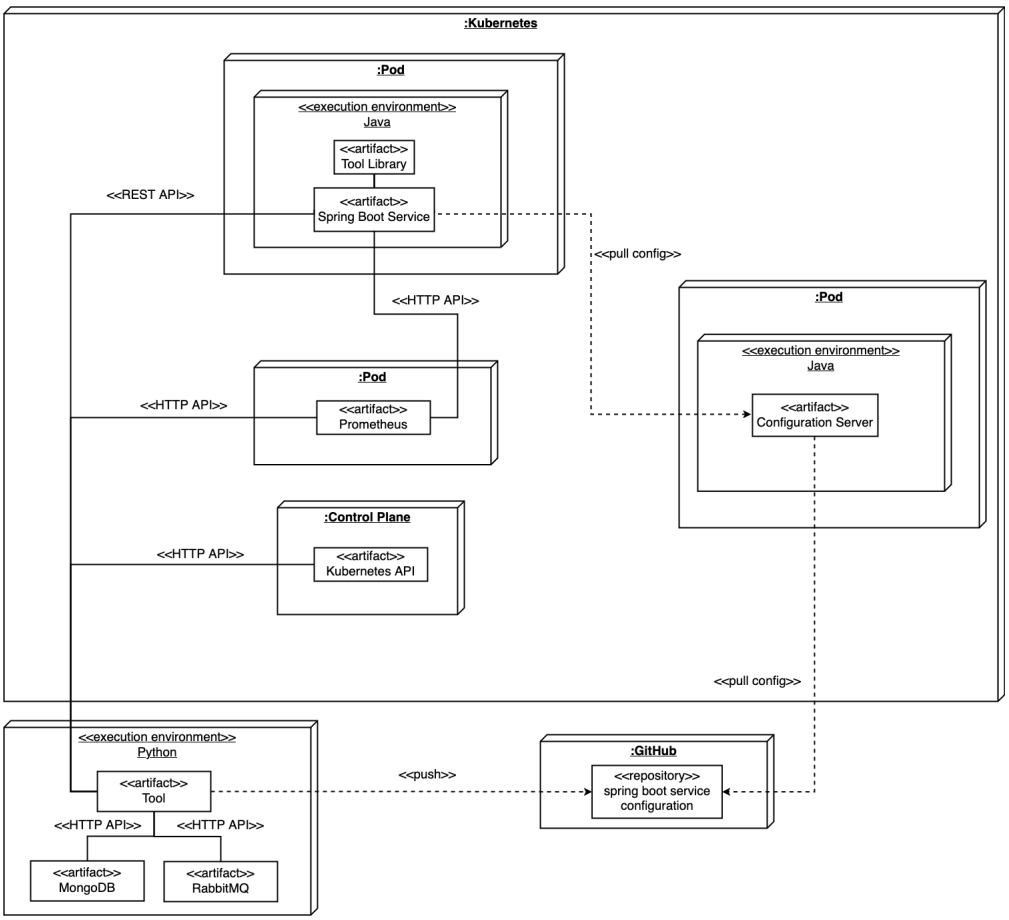

Figure 5.1: Deployment diagram showing the dependencies between the tool and system that is monitored

As depicted in the architecture, MongoDB serves as the primary data store, while RabbitMQ facilitates inter-component communication. The subsequent sections will elaborate on the individual components, their interactions, and their alignment with the MAPE-K reference model.

To communicate with the monitored system, a communication tunnel needs to be established. For evaluation purposes, this was achieved through Kubernetes port forwarding mechanism. The tool extracts data from the Kubernetes API, Prometheus, and Spring Boot endpoints, utilizing this information for decision-making in the analysis phase.

SDKs were employed within Python to facilitate data retrieval from Kubernetes and Prometheus, as well as for executing actions via the Kubernetes API. For Spring Boot-related interactions, dynamic configuration was leveraged through a dedicated configuration server managing application YAML files stored in a Git repository. When a service is initiated, it requests its configuration file from the server. This approach allows runtime configuration modifications without necessitating service restarts, as only a refresh operation via an endpoint is required.

Additionally, a Spring Boot library was developed as an integral component of the tool. This library employs aspect-oriented programming (AOP) to integrate resilience patterns such as circuit breakers, fallbacks, queued requests, retries, and backoff mechanisms. It's implemented in Java using aspect-oriented programming in Spring Boot, which means it can only be used in Spring Boot services. The configuration of these patterns remains dynamic, determined at runtime via the Spring Boot configuration framework. This abstraction layer ensures that services remain independent of the tool while benefiting from its resilience features. Moreover, the library provides an endpoint to track service latency, feeding data into the tool's analysis phase. The resilience mechanisms are integrated into Feign clients, ensuring that external service calls are handled with the fault tolerance strategies. The reasons for selectively implementing resilience patterns, as explained in the Requirements Analysis section, will be further explored in the Solution Classification section. When the library is integrated into the service, its resilience patterns are not applied uniformly across all endpoints. Only those endpoints explicitly annotated with @Resilience utilize the library's functionality.

5.2 Algorithms

This section provides an in-depth analysis of the tool's core components, their functionalities, and their alignment with the MAPE-K reference model.

5.2.1 MAPE-K

The MAPE-K reference model is a feedback control loop comprising four primary components: Monitoring, Analysis, Planning, and Execution [25].

The following figure 5.2 illustrates the tool's system architecture:

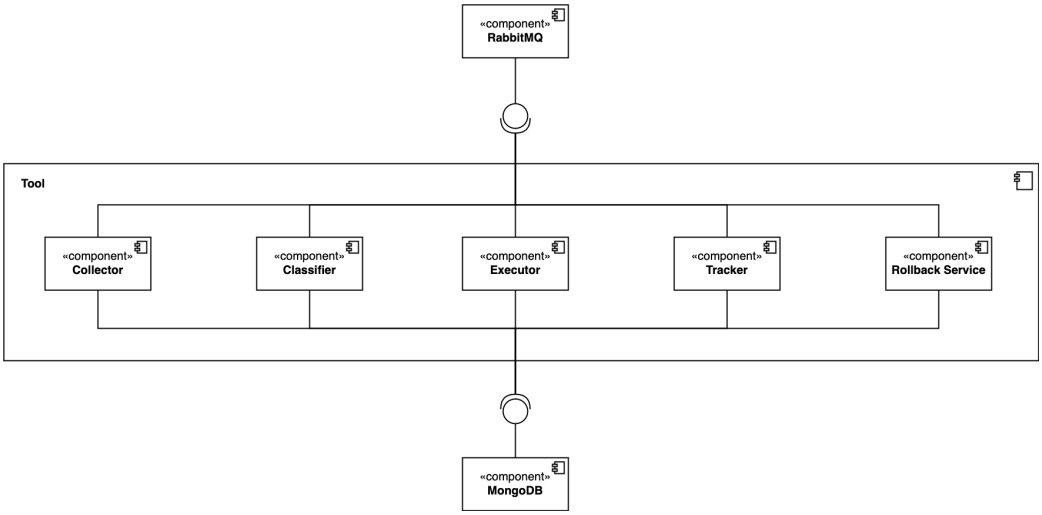

Figure 5.2: Component diagram of the tool

The flow of the tool is described in the sequence diagram 5.3:

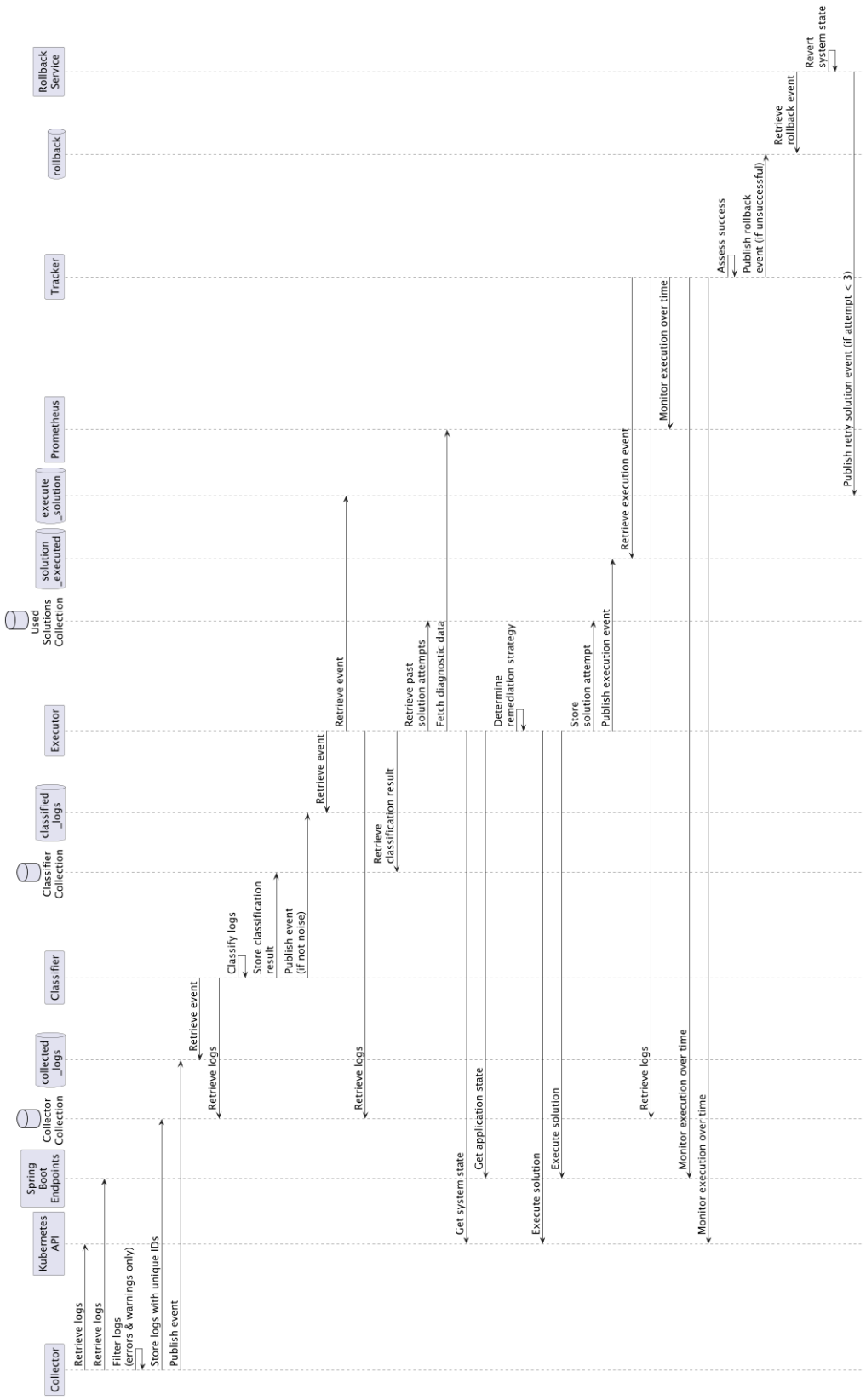

Figure 5.3: Sequence diagram of the tool

The system consists of the following core components:

- Log Collector
- Log Classifier
- Solution Executor
- Solution Tracker
- Rollback Service

The names in the diagram have been modified to fit within the available space, resulting in slight differences from those in the text.
The Log Collector component is responsible for aggregating logs from both the Kubernetes cluster and Spring Boot services. To achieve this, the tool employs the Kubernetes API to retrieve logs. The component initially filters logs, ensuring that only those categorized as errors or warnings are subjected to further processing. Each retrieved log is assigned a unique identifier and subsequently stored in MongoDB. Following this, an event is generated and placed in the collected_logs queue.
The Log Classifier component retrieves events from the queue and attempts to classify them. If a log is determined to be noise, it is discarded; otherwise, the classification result is stored in MongoDB, and a new event is created in the classified_log queue.
The Solution Executor component fulfills two primary responsibilities: determining the appropriate remediation strategy and executing the selected solution. It processes events from the classified_log queue, utilizing data from MongoDB, Prometheus, the Kubernetes API, and Spring Boot endpoints to make informed decisions. The methodology for selecting an optimal solution is detailed in the Solution Classification section of this chapter. Once a solution has been identified, it is executed, and a corresponding event is placed in the solution_executed queue. This event includes an attempt counter, which tracks the number of times different solutions have been attempted to resolve the error. A retry mechanism is integrated into the tool, ensuring that if an initial remediation attempt is unsuccessful, subsequent attempts are made. Each retry iteration benefits from additional diagnostic data, as the system continues to operate under error conditions. However, if the number of attempts exceeds three, the Solution Executor ceases further execution.
The Solution Tracker component monitors solution execution by reading events from the solution_executed queue. The tracking methodology is solution-specific; for certain solutions, error logs are compared against new logs using predefined analytical formulas,

while alternative solutions may employ different tracking mechanisms. These methodologies are further elaborated in the Solution Classification section.
Tracking is conducted over a predefined time interval, configurable within the system settings (e.g., 60 seconds). Upon expiration of this interval, the Solution Tracker determines whether the solution has been successful. If the error has been resolved, the tool logs the result and terminates the process. Conversely, if the solution is deemed unsuccessful, an event is created in the rollback queue.
The Rollback Service is responsible for reverting changes implemented by the Solution Executor. Upon reading an event from the rollback queue, the Rollback Service utilizes the embedded information to restore the system to its previous state. If the rollback is successful and the attempt counter remains below three, a new event is generated and placed in the execute_solution queue. This queue is processed by the Solution Executor, thereby enabling an iterative loop in which the tool continues retrying error resolution until a predefined termination condition is met.

Each component aligns with MAPE-K as follows: the Log Collector corresponds to the Monitoring phase, the Log Classifier represents the Analysis phase, and the Solution Executor serves as the Planning and Execution component, in conjunction with the Rollback Service. Knowledge is encapsulated within MongoDB, which stores logs, classifications, decisions, and executed solutions.

5.2.2 Log Classification

As previously mentioned in the Introduction chapter, the log classification mechanism was initially developed and validated in an earlier project, ″Classifier Development for Log Analysis in Spring Boot and Kubernetes″ [40]. This section provides a concise summary of data collection, preprocessing, model selection, implementation, training, and evaluation. For a comprehensive discussion, refer to the cited project.

As already seen in the architecture, the log classification is a fundamental component of the tool, determining whether logs warrant further processing. The goal of the log classification in its core, is to be able to accurately categorize Java service logs and Kubernetes events.
As already mentioned in the Requirements Analysis chapter, the tool should be able to classify the logs into the following categories:

- spring_timeout_error
- spring_third_party_service_error
- spring_down_error
- spring_configuration_error
- spring_request_error
- spring_db_connection_error
- kubernetes_pod_unhealthy
- service_down_kubernetes_error
- kubernetes_node_problem
- kubernetes_network_error
- kubernetes_mount_fail
- kubernetes_low_cpu_memory
- kubernetes_invalid_image_error
- kubernetes_failed
- kubernetes_configuration_error

The classification model outputs a probability distribution over these categories. If confidence falls below a predefined threshold (currently 85%), the log is deemed noise and disregarded.
The dataset used for training the model comprises 9,600 data points, with approximately 600 data points per category. The data collection process for this project was particularly challenging due to the lack of publicly available datasets suited for this purpose. Consequently, two primary methods were employed to gather the necessary data.
Following an extensive investigation into the most frequently occurring errors in Spring Boot services that impact resilience, latency, and system stability, relevant errors were extracted using the BigQuery Stack Overflow dataset. A combination of post querying and tag filtering was applied to compile a comprehensive dataset of Spring Boot errors [40].
For Kubernetes-related errors, as outlined in the Requirements Analysis chapter, data was gathered with the assistance of xChange Solutions [43]. Datadog monitoring was

leveraged, utilizing specific filters and the Datadog API to extract relevant error data. [40] Given that both datasets were initially unprocessed and contained raw data, preprocessing steps were undertaken to enhance their usability.

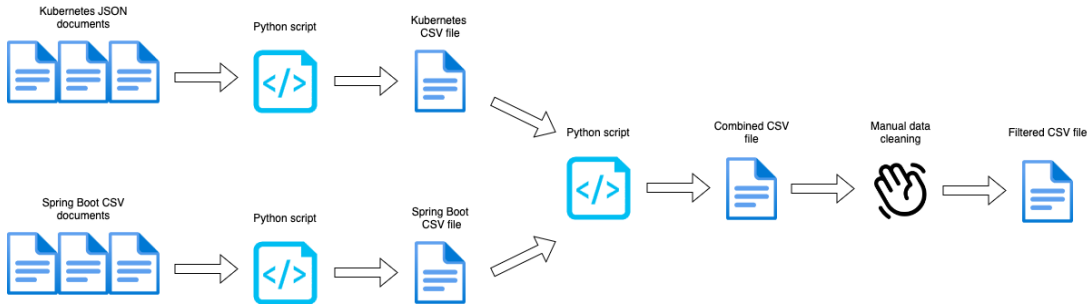

Figure 5.4: Processing of the dataset used for training the classification model

The figure 5.4 shows the processing of the dataset used for training. The Kubernetes data was in JSON format, while the Spring Boot data was in CSV, containing Stack Overflow questions. Preprocessing was handled by Python scripts [40]:

- Kubernetes Script: Converted JSON logs to CSV, cleaned messages, categorized errors, analyzed label frequency, and balanced data by capping labels at 600 entries.
- Spring Boot Script: Processed CSV logs similarly, extracting and categorizing error messages, splitting data, and adjusting label counts by reducing or duplicating entries as needed.

Both scripts produced cleaned CSV files, which were merged into a single dataset. A final manual inspection ensures data quality before training.

For the implementation, four different classification approaches were explored and validated: Those were semantic embeddings, LSTM, XGBoost and Fine-tuning using SetFit. The semantic embeddings approach was primarily exploratory and served as a foundation for the subsequent models.
The training results indicate that semantic embeddings, particularly those derived from fine-tuning a pretrained Sentence Transformer model with SetFit, were not well-suited for this classification task. While initially promising, the observed performance was significantly lower than that of LSTM and XGBoost models. Moreover, the training time for SetFit was considerably high, further diminishing its practicality for this application [40].

Among the evaluated models, XGBoost demonstrated the highest accuracy (98.21%) while also exhibiting the shortest training time. The LSTM model achieved an accuracy of 95.04% on Kubernetes data, which is slightly lower than XGBoost but still satisfactory. But it suffered a severe performance drop when incorporating Spring Boot data, with accuracy declining to 6.01%, a level equivalent to random guessing. This suggests that the LSTM model struggled with the heterogeneity of the dataset [40].
The superior performance of XGBoost can be attributed to its inherent characteristics, as documented in prior research [23]. XGBoost has been shown to outperform LSTM in certain applications due to its efficiency, stability, and scalability, particularly when handling large structured/tabular datasets. Although system error logs exhibit sequential properties, which typically favor LSTM-based approaches, the findings of this study suggest that preprocessing techniques and architectural choices significantly impact model performance [23].
The LSTM model in this study employed a simple architecture with a single LSTM layer and a dense layer, which may have contributed to its suboptimal performance. Notably, comparable results were achieved in a related master's thesis on Ericsson system logs, where a Bag-of-Words (BOW) representation combined with XGBoost yielded superior classification performance [22]. This further supports the conclusion that, despite the sequential nature of system logs, non-sequential models such as XGBoost can be more effective in certain contexts.

5.2.3 Solution Classification

Solution classification is a critical component of the Solution Executor, determining the appropriate resolution strategy for detected errors. As previously mentioned, the solution selection is based on information derived from classified logs, Prometheus metrics, Kubernetes API data, and Spring Boot endpoints. A rule-based system was implemented to facilitate decision-making regarding the appropriate solution. The choice of a decision tree model was motivated by the constraints of this thesis, including time limitations and the complexity of the overall implementation. Given these constraints, a decision tree represented the most suitable approach for this initial version of the tool.

The decision tree used for solution classification is illustrated in the following five figures 5.5 to 5.9.

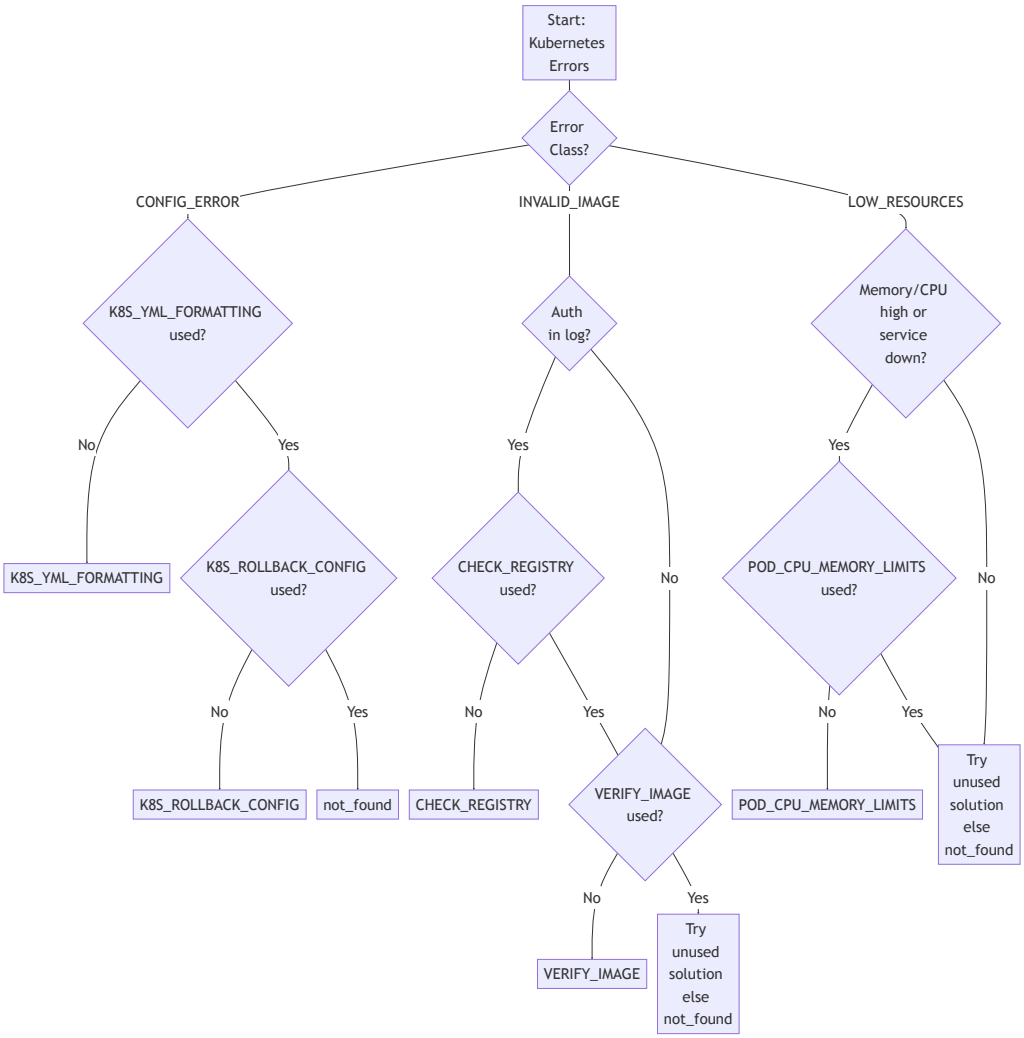

Figure 5.5: Kubernetes error part of the decision tree

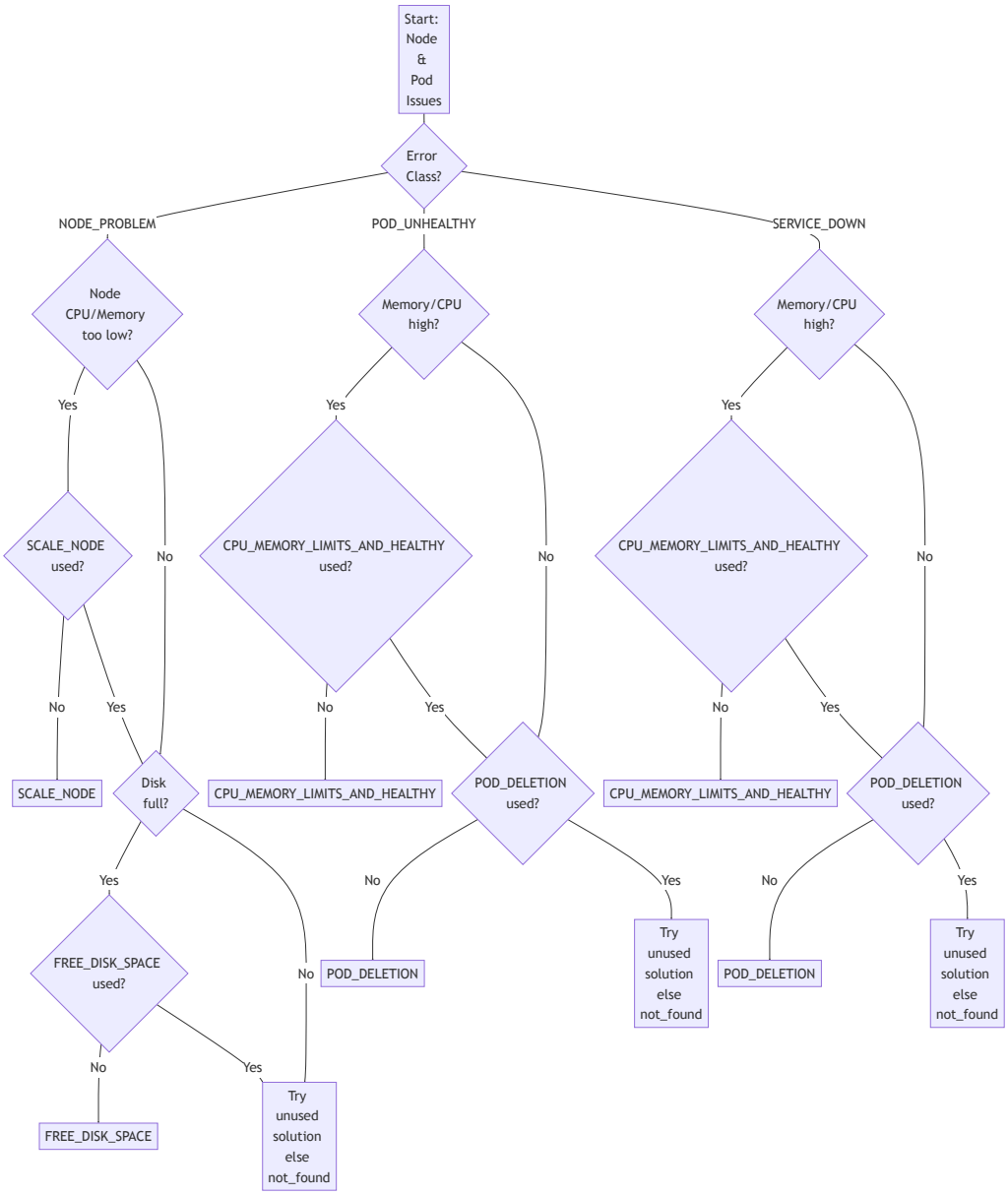

Figure 5.6: Node and pod issues part of the decision tree

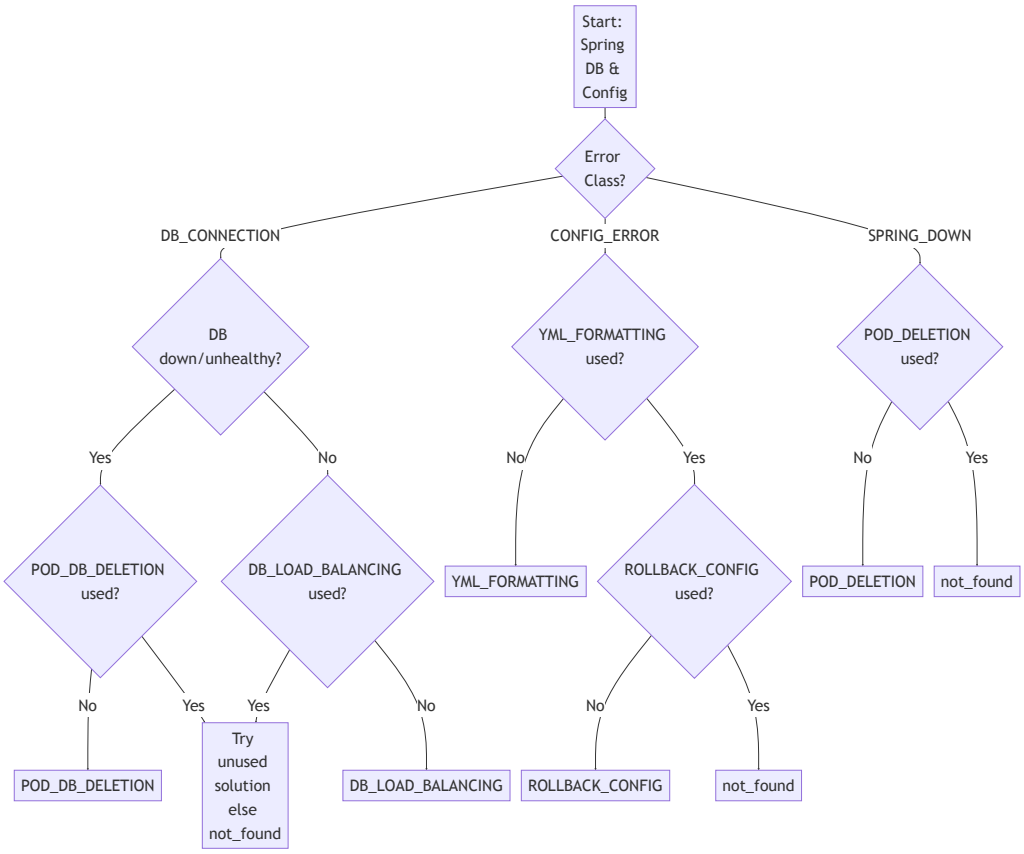

Figure 5.7: Spring DB and configuration part of the decision tree

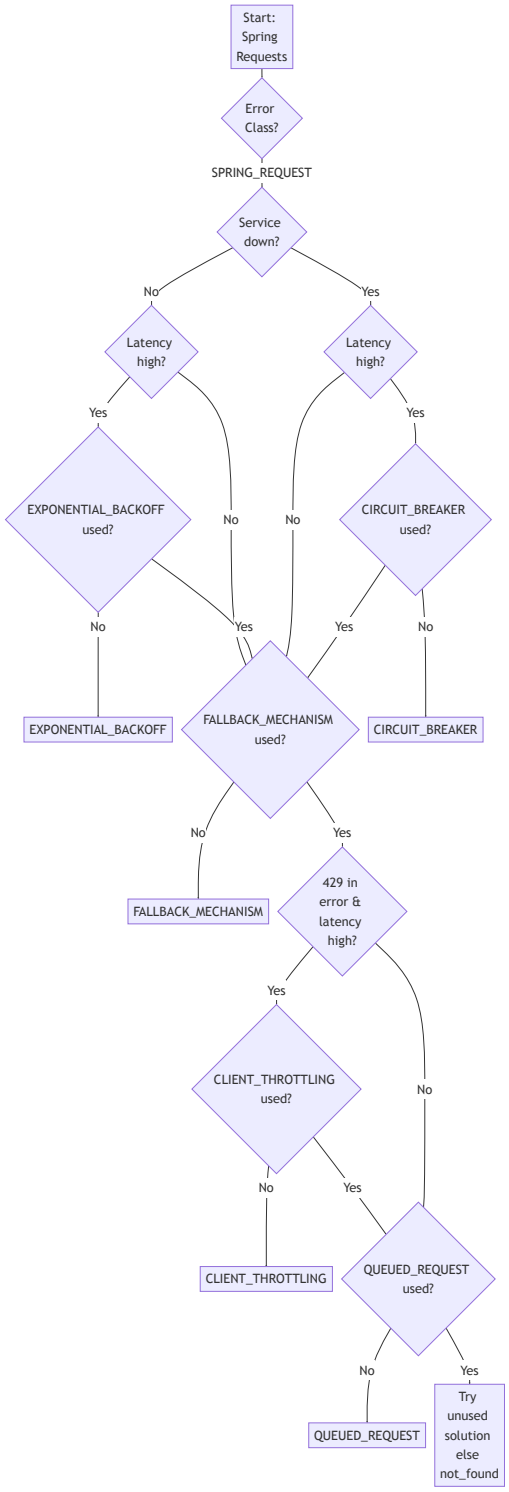

Figure 5.8: Spring requests part of the decision tree

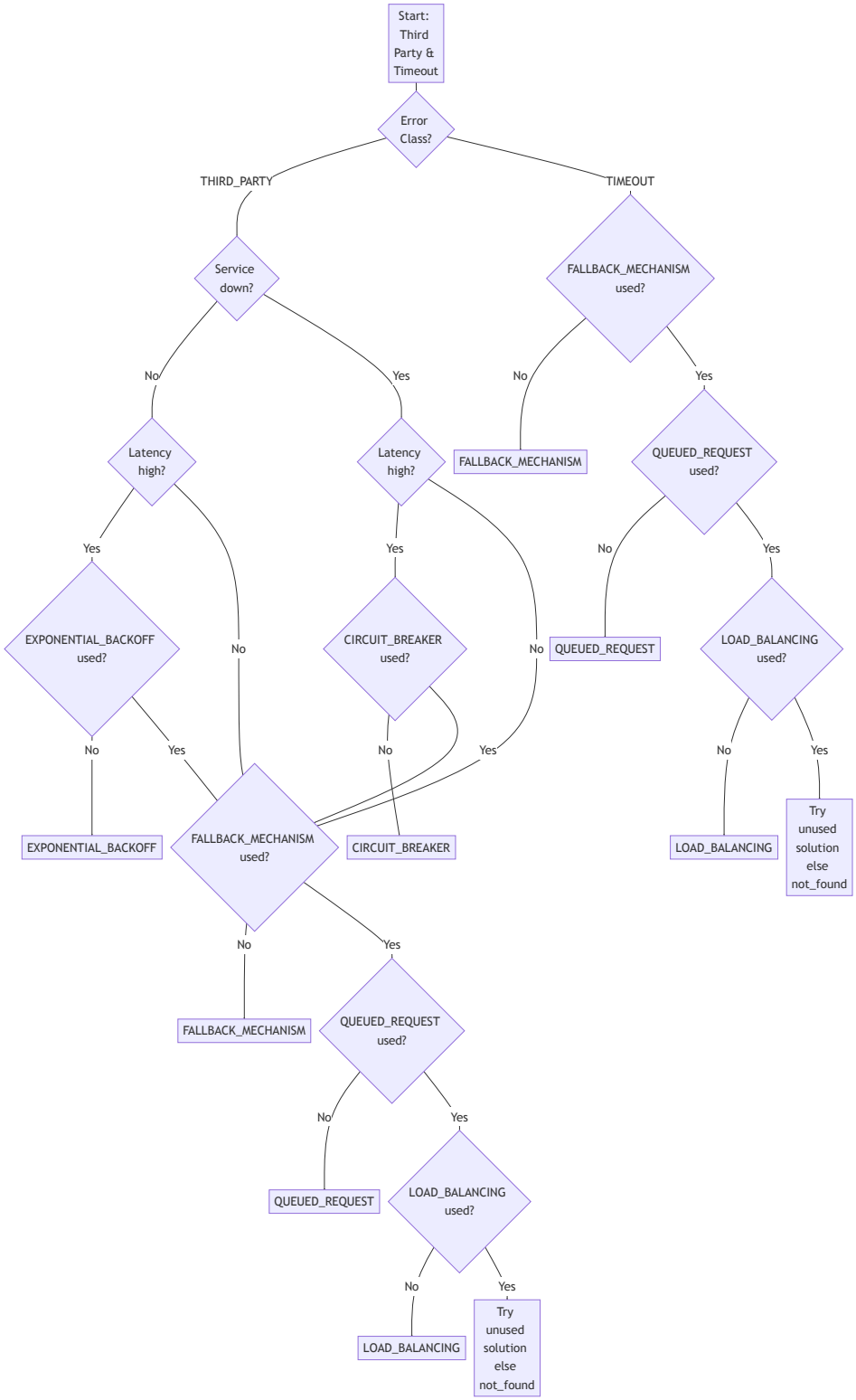

Figure 5.9: Spring third party and timeout part of the decision tree

In the implementation there is only one decision tree, but for the sake of clarity, it has been divided into five figures, else it could not be displayed properly.
The terminology and the meaning of the decision tree is defined in the chapter Requirements Analysis.
For Kubernetes-related errors, the decision tree considers multiple failure scenarios. In the case of a Kubernetes configuration error, the system verifies whether YAML formatting has already been applied before proceeding to rollback configuration if necessary. If neither solution is available, the classification results in not found. For a Kubernetes invalid image error, the tree assesses authentication logs, potentially applying check registry or verify image, with fallback ensuring that unused solutions are attempted before concluding that no viable resolution exists.

Resource-related issues such as Kubernetes low resources (CPU and memory limitations) and Kubernetes node problems follow a similar structured approach. The classification process evaluates whether CPU or memory constraints contribute to the failure, applying pod CPU and memory limits, scale node, or free disk space where applicable. If no solution proves effective, the system attempts alternative strategies before marking the issue as unresolved. Similarly, when a Kubernetes pod unhealthy or service down Kubernetes condition arises, the tree assesses CPU and memory usage, determining whether CPU memory limits and healthy or pod deletion should be applied to restore functionality.

Errors related to Spring Boot services, such as a Spring database connection error, Spring configuration error, and Spring service down, are handled in a similar way. For database-related failures, database health check is performed, solutions such as load balancing and DB pod deletion are considered. Configuration errors undergo a similar assessment, applying YAML formatting or rollback configuration where appropriate. If a service is detected as down, the system attempts pod deletion as a corrective measure.

Service degradation scenarios, including request error, third-party service error, and timeout errors, are addressed by evaluating latency and availability. Solutions such as exponential backoff, fallback mechanism, and circuit breaker are prioritized, followed by load balancing and queued request where necessary.

For unclassified or invalid errors, the system defaults to not found, ensuring that unrecognized failure patterns do not disrupt the classification process. This structured decision-making approach enables an adaptive response to diverse failure conditions, optimizing solution selection based on predefined criteria.

In addition to decision-making, tracking the applied solutions and their rollback mechanisms is essential.

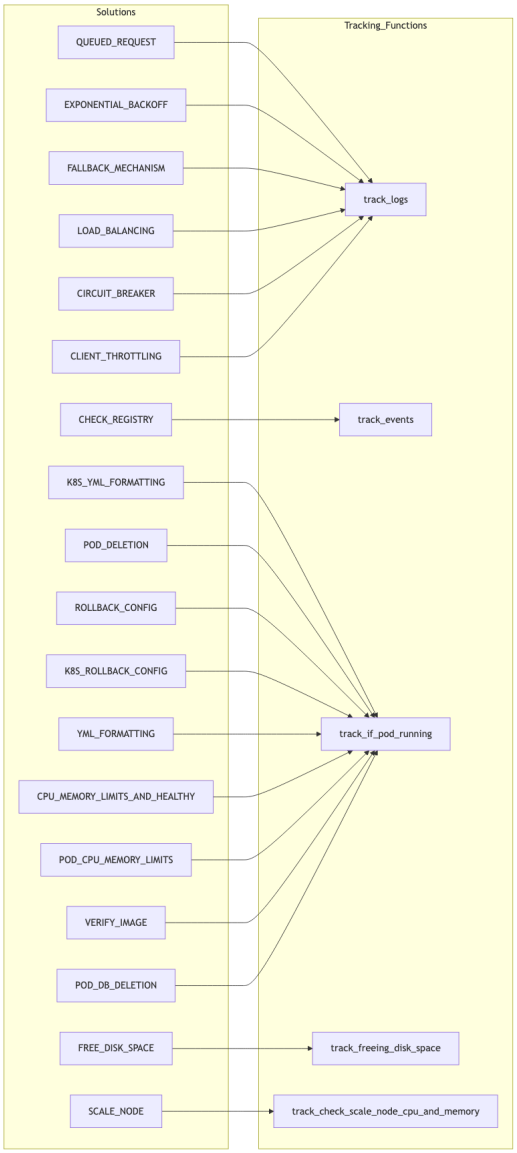

Figure 5.10: Mapping of solutions to the tracking functions

The tracking process ensures that solution execution is monitored effectively, providing insights into the impact and success of each corrective action. The tracking system consists of multiple functions, each responsible for monitoring a specific aspect of the

applied solutions. The track logs function is linked to connection-handling solutions such as queued request, exponential backoff, fallback mechanism, load balancing, circuit breaker, and client throttling, ensuring that latency-related issues are properly observed. The track events function monitors registry-related solutions, specifically check registry, to validate their application. For Kubernetes and Spring Boot configuration-related issues, the track if pod running function is responsible for overseeing solutions like YAML formatting, rollback configuration, pod deletion, CPU and memory limits, pod CPU and memory limits, verify image, pod database deletion etc. This ensures that structural modifications, including configuration corrections and resource allocation changes, are tracked in real time. Resource management tracking is handled by specialized functions. The track freeing disk space function monitors the nodes free disk space, ensuring that disk-related interventions are properly executed and evaluated. Similarly, the track check scale node cpu and memory function is responsible for tracking the application of scale node, ensuring that scaling operations are performed based on observed resource constraints. By systematically linking solutions to their corresponding tracking functions, this approach provides a structured mechanism for assessing the effectiveness of each corrective action. This enables better rollback strategies and facilitates continuous improvement in failure resolution.

In this isolated setup, rollback is only applied to the YAML formatting solution, as other solutions do not require rollback because the system is quite isolated. This selective rollback approach simplifies recovery while maintaining the integrity of applied solutions.

Not all errors and solutions have been implemented in the tool. This is due to the complexity of certain issues and their corresponding solutions, which would require a significant amount of time beyond the scope of this thesis. Additionally, technical constraints further limited the range of implementations.

While the decision tree approach was effective for this thesis, a more sophisticated algorithm should be implemented in a production-ready version of the tool. The rule-based system, though well-suited for well-defined problems, becomes increasingly complex as new rules are added and lacks the flexibility to adapt to novel scenarios [27].
To address these limitations, further research was conducted to identify a more scalable solution for selecting the appropriate resolution strategy. The classification process should incorporate both the error class and relevant metrics obtained from Prometheus, Kubernetes API, and Spring Boot endpoints.
XGBoost was identified as a highly suitable alternative due to several key advantages:

- Effective handling of both numerical and categorical data [44]
- Built-in regularization to mitigate overfitting [15]
- Fast training and inference times, enabling real-time responses [15]
- Feature importance insights, enhancing interpretability [15]
- Proven reliability in production environments [15]

A crucial requirement of the system is its ability to accommodate new errors and solutions over time. To achieve this, the tool should support incremental classification improvements without requiring complete model retraining. A plugin-based architecture would further enhance its extensibility. XGBoost's support for incremental learning and transfer learning techniques allows for continuous expansion of the classification model without the need for complete retraining [42]. This design establishes a solid foundation for an automated incident response system that can adapt to evolving challenges while maintaining robust and reliable performance.

6 Experimental Setup and Implementation

This chapter outlines the methodology used to evaluate the tool, including the test environment setup and the implementation of chaos engineering setup.

6.1 Evaluation Methodology

For the evaluation of the tool, the CHESS framework, as described in the paper ″CHESS: A Framework for Evaluation of Self-adaptive Systems based on Chaos Engineering″[30] was utilized.
″CHESS is a systematic approach for evaluating self-adaptive and self-healing systems using chaos engineering principles. It perturbs systems to observe their responses and includes predefined fault injection scenarios, a self-monitoring service for logging behaviors, and a managing system service that restores stability″[30].

The original implementation of CHESS, as presented in the paper, was not directly used due to its limitation of supporting only five chaos engineering injections/methods. The requirements of this thesis require the execution of approximately 40 different chaos engineering methods/injections.
As emphasized in the CHESS study and based on fundamental chaos engineering principles, a target system is required for executing chaos engineering experiments [30]. In this work, two distinct systems were selected: the SMS and the EOMS systems. A detailed description of these systems is provided in the subsequent section of this chapter.

In the CHESS framework, system state monitoring is performed using a system monitor [30]. However, in this thesis, an alternative approach was adopted, leveraging performance testing. Specifically, k6 performance tests were executed concurrently with chaos engineering experiments, and the resulting performance data was analyzed to assess the system state post-experimentation.

The evaluation methodology is structured as follows: In the first experiment, the SMS system was executed without the proposed tool, and chaos engineering was applied. The chaos engineering process was divided into 12 distinct scenarios, with each scenario encompassing one to six chaos engineering injections/methods. Concurrently, performance testing was conducted. In the second experiment, the EOMS system was tested under similar conditions, without the tool, and with both chaos engineering and performance testing applied. The third experiment involved executing the SMS system with the tool, followed by chaos engineering and performance testing. Finally, in the fourth experiment, the EOMS system was tested with the tool under the same conditions.
A total of 48 scenarios were executed, distributed across the four experiments, with each scenario consisting of one to six chaos engineering injections/methods. Throughout each test, approximately seven requests per second were sent to the system. The SMS system included five endpoints targeted by these requests, whereas the EOMS system contained 11 endpoints. Each experiment was conducted over a duration of approximately 12 hours. All experiments and scenarios were executed in isolation within a local Minikube environment. This approach was selected due to the constraints of time and resources associated with this thesis. Deploying the entire setup on a cloud provider would have been prohibitively time-consuming and financially impractical.

6.2 Test Environment Setup

As previously mentioned in this chapter, two systems were developed in which chaos engineering, combined with performance testing, was employed to evaluate the tool. The first system is the Student Management System (SMS), a web application designed for managing schools and their students. The second system is the E-commerce Order Management System (EOMS), which facilitates order placement in an online shop. Both systems were developed using the Spring Boot framework and are deployed in a Kubernetes cluster.
The source code for both projects was originally obtained from GitHub but was heavily modified to meet the requirements [3, 4]. The following figure 6.1 illustrates the architecture of the SMS system:

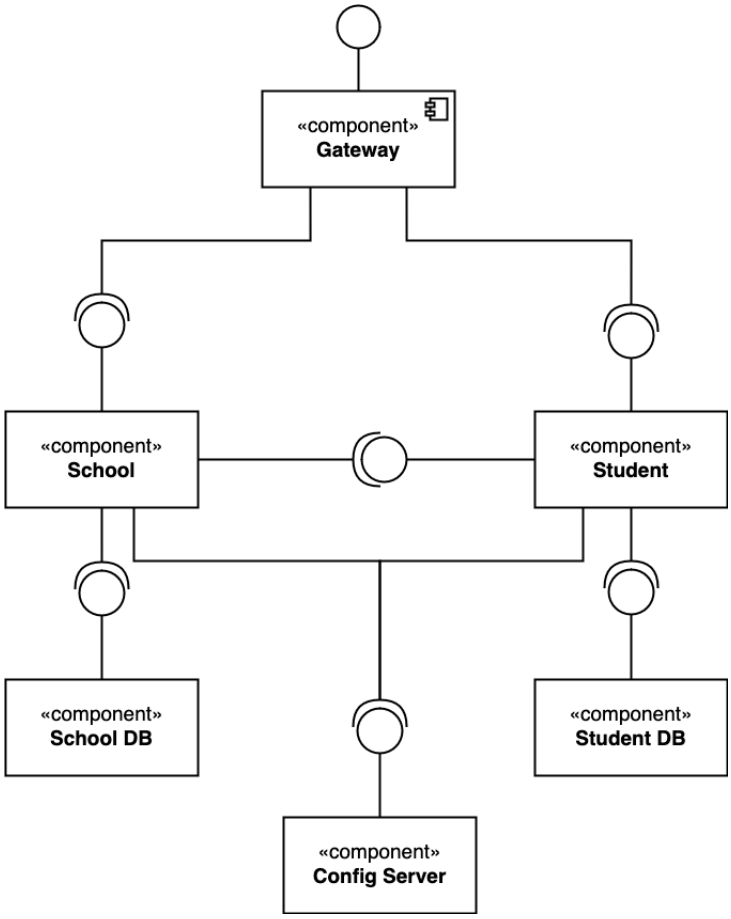

Figure 6.1: Component diagram of the SMS system

The SMS system consists of four services: the gateway, school, student, and config-server services. The gateway service serves as the system's entry point, routing requests to the appropriate services. The school service is responsible for managing schools within the system and provides three endpoints.

- POST /api/v1/schools - Creates a school
- GET /api/v1/schools - Returns all schools
- GET /api/v1/schools/with-students/school-id - Returns the school with all the students in it

Similarly, the student service manages student records and also exposes three endpoints.

- POST /api/v1/students - Creates a student
- GET /api/v1/students - Returns all students
- GET /api/v1/students/school/school-id - Returns the students with the given school id

A key characteristic of the system is that the school service communicates synchronously with the student service. Specifically, the GET /api/v1/schools/with-students/school-id endpoint invokes the GET /api/v1/students/school/school-id endpoint from the student service. Both the school and student services utilize separate PostgreSQL databases.
Additionally, the system includes deployments for Zipkin, Prometheus, Loki, PgAdmin, and Postgres-Exporter, which are used for system monitoring. These components are omitted from the architectural figure due to visualization constraints.
The config service retrieves configuration settings from a Git repository and facilitates dynamic configuration changes across all services at runtime. As discussed in the implementation chapter, this is achieved through Spring Cloud Config.

The library forming part of the tool, as described in the implementation chapter, was incorporated into the school service during the SMS system experiments involving the tool deployment. The subsequent section describes the chaos engineering script, which leverages Spring Cloud Config to introduce chaos into the system via the student service.

The architecture of the EOMS system is shown in the following figure 6.2:

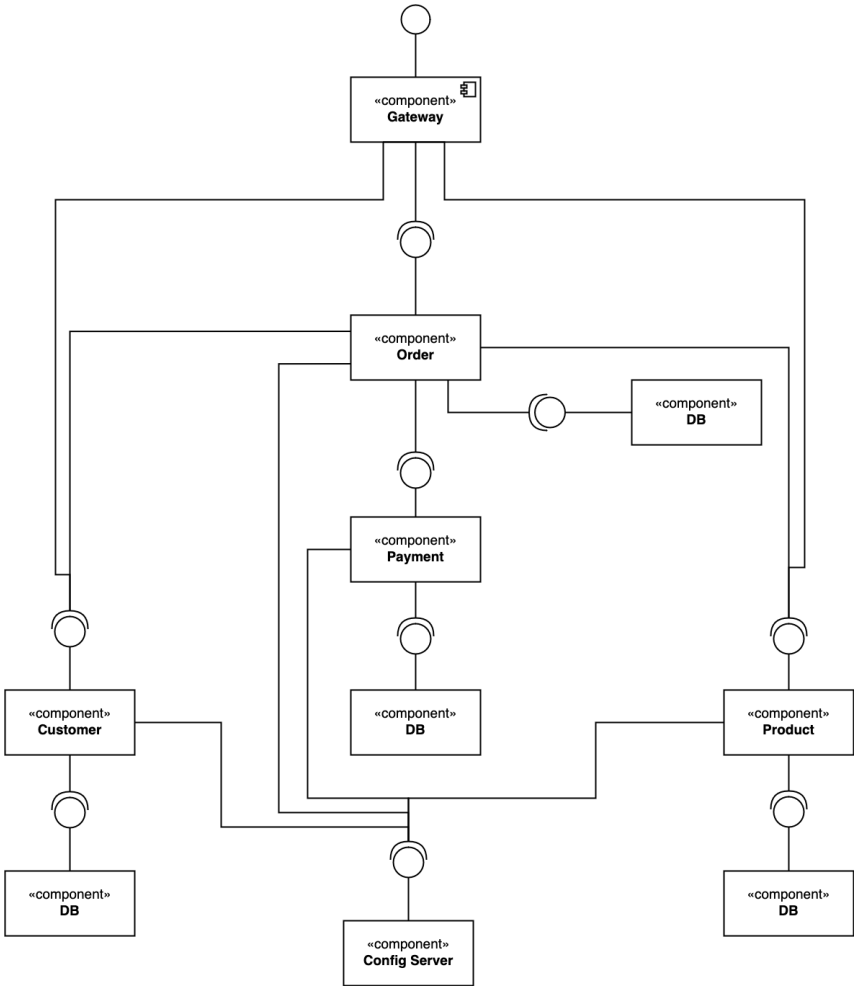

Figure 6.2: Component diagram of the EOMS system

The EOMS system comprises six services: gateway, config-server, order, product, customer, and payment services. The gateway and config-server function similarly to their counterparts in the SMS system. The order service manages customer orders and exposes four endpoints.

- GET /api/v1/orders - Returns all orders
- GET /api/v1/orders/id - Returns the order with the given id
- GET /api/v1/order-lines/order/id - Returns the order lines with the given order id

- POST /api/v1/orders - Creates an order

The product service, upon initialization, pre-populates its database with 25 different products and offers four endpoints.

- POST /api/v1/products - Creates a product
- GET /api/v1/products - Gets all products
- GET /api/v1/products/id - Gets the product with the given id
- POST /api/v1/products/purchase - Creates a purchase

The customer service provides five endpoints for managing customer data.

- POST /api/v1/customers - Creates a customer
- PUT /api/v1/customers - Updates customer information
- GET /api/v1/customers - Returns all customers
- DELETE /api/v1/customers/id - Deletes the customer with the given id
- GET /api/v1/customers/id - Returns the customer with the given id

The payment service handles payment transactions and includes a single endpoint.

- POST /api/v1/payments - Creates a payment

The EOMS system is inherently more complex than the SMS system, not only due to the greater number of services and endpoints but also due to the increased complexity of inter-service communication. The POST /api/v1/orders endpoint in the order service triggers the GET /api/v1/customers/id endpoint in the customer service, followed by the POST /api/v1/products/purchase endpoint in the product service. After that the order is saved and an order-line is created in the database. Once the order is processed, an order-line is created in the database, after which the POST /api/v1/payments endpoint in the payment service is invoked.
Although the inter-service communication remains synchronous, the complexity is significantly higher than in the SMS system. Furthermore, the order service represents a single point of failure in the architecture. The customer service employs MongoDB as its database, while the product, order, and payment services each utilize separate PostgreSQL databases.
For monitoring purposes, the same stack as in the SMS system—Zipkin, Prometheus,

Loki, PgAdmin, and Postgres-Exporter—was deployed. However, these components, along with MongoDB and MongoDB-Exporter, were omitted from the figure due to visualization limitations. MongoDB-Exporter was specifically used for monitoring the MongoDB database.
The tool's library was also integrated into this system, specifically into the order service during one of the experiments. In addition, the chaos engineering script needed to be capable of injecting failures into the system. For this purpose, the product service was selected as the target of chaos injections.

6.3 Chaos Engineering Implementation

For the validation of the tool, a chaos engineering script was developed in conjunction with a performance testing script. This section provides a detailed description of both the chaos engineering script and the performance testing script.
The following figure 6.3 illustrates the architecture of the chaos engineering setup:

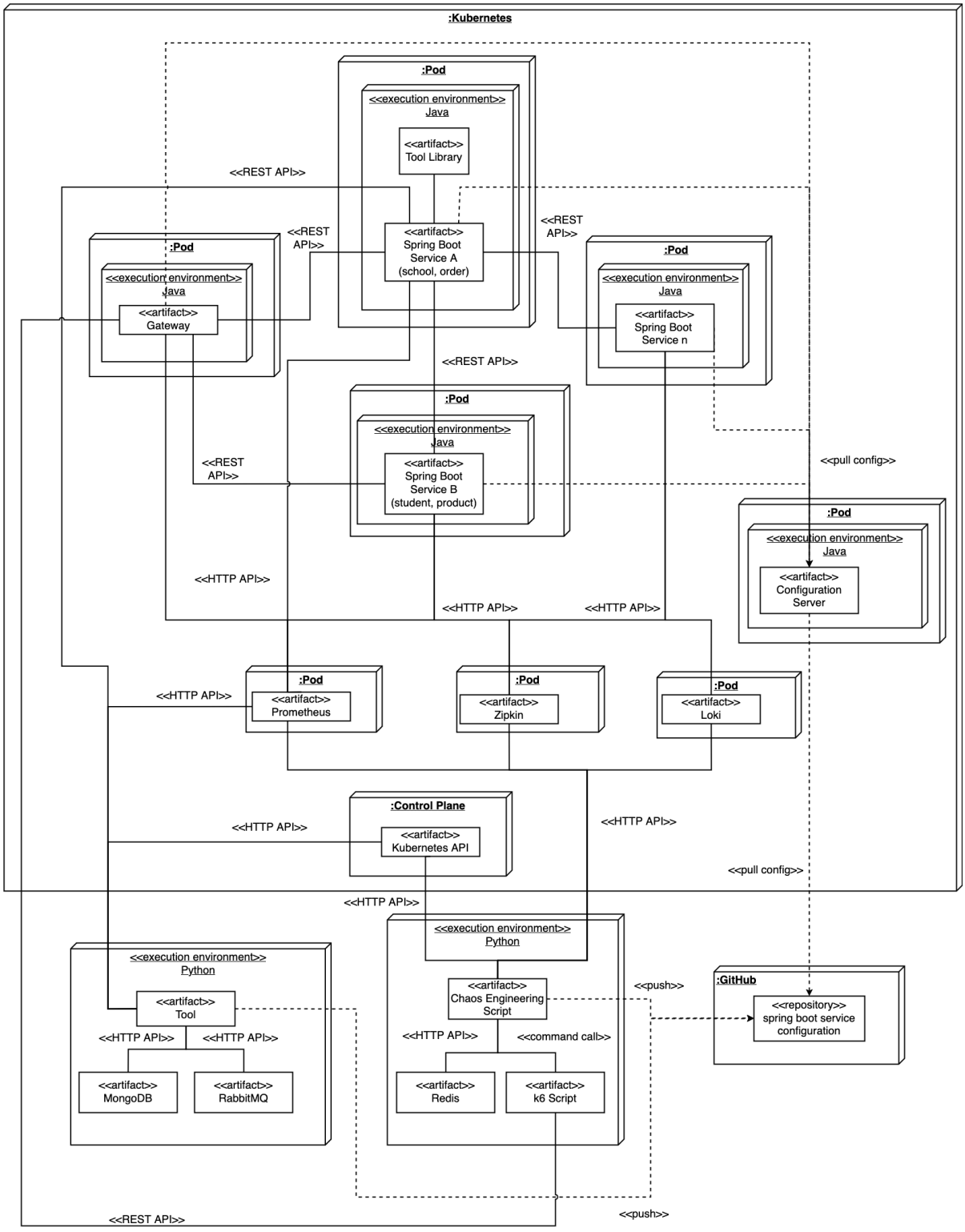

Figure 6.3: Deployment diagram of the chaos engineering setup

Although the figure should ideally show the interaction between the chaos engineering script and other Spring Boot services, these interactions could not be effectively represented and were therefore omitted from the figure.
The chaos engineering script is the core component of the testing system, defining all experimental scenarios. These scenarios simulate real-world failure conditions, including service timeouts, HTTP error responses, configuration inconsistencies, resource constraints, and infrastructure failures. The script interacts with the Kubernetes API to manipulate deployments, pods, and other resources, thereby enabling infrastructure-level chaos experiments. It can simulate various Kubernetes-specific failure modes, such as incorrect image configurations, pod health issues, node failures, and resource limitations.
Additionally, the script injects application-level failures, including artificial latency, predefined HTTP error codes, and dynamic configuration modifications. This is accomplished using the Spring Cloud Configuration Server and GitHub, which enable runtime configuration changes to introduce controlled failure conditions.
The script also integrates k6 load testing, which operates continuously throughout the experiment to simulate realistic user traffic. This ensures that system behavior under failure conditions can be observed in a high-load environment.
As previously mentioned, the framework collects extensive telemetry data during the experiments. This includes Kubernetes logs and events, distributed traces from Zipkin, metrics from Prometheus, and logs from Loki, providing a comprehensive observability framework for analyzing system behavior under failure conditions. Additional metrics are gathered from both the k6 script and the chaos engineering script in the form of logs.
The script fully automates the experimental lifecycle, encompassing infrastructure provisioning (Minikube cluster setup and microservices deployment), execution of experiments, data collection, and cleanup. This ensures reproducibility and consistency across chaos experiments.

For each failure type identified in the Requirements Analysis section of the Concept chapter, a corresponding scenario was designed and executed. The implemented scenarios are as follows:

Spring Timeout Scenario:
In this scenario, requests are continuously sent to the system via the Gateway for five minutes before any fault injection occurs. Subsequently, an incremental wait time is introduced in the Student/Product service at four different intervals, each lasting five minutes, to simulate timeout conditions. Once the timeout phase is completed, the wait time modification is removed, and the service resumes normal operation for five minutes.

Following this, another wait time is introduced into the Student/Product service, inducing a new timeout condition. After two minutes, the entire service is forcibly terminated, enabling an evaluation of queued requests, fallback mechanisms, and load balancing across multiple service instances. This phase continues for 15 minutes. Finally, a rollback is executed, restoring normal service operation for five minutes. A five-minute waiting period is maintained between scenarios.

Spring Third-Party Service Scenario:
Requests are initiated through the Gateway five minutes before the experiment begins. The Student/Product service is then configured to return HTTP 500 error responses for a duration of 20 minutes. Following this phase, the service is restored to normal operation for five minutes.
The HTTP 500 error responses are then reintroduced for two minutes to simulate a third-party service failure. Subsequently, the entire service is forcibly stopped, allowing for an evaluation of queued requests, fallback mechanisms, load balancing strategies, and the circuit breaker pattern. This phase lasts 15 minutes. The rollback phase ensures that error responses cease, and the service returns to normal operation for five minutes. A five-minute waiting period is enforced between scenarios.

Spring Request Scenario:
Requests are continuously sent through the Gateway five minutes before the experiment begins. The Student/Product service then starts responding with HTTP 400 error codes to the School/Order service for 20 minutes. After this phase, the HTTP 400 error is removed, and normal service operation resumes for five minutes.
Subsequently, the Student/Product service is reconfigured to return HTTP 400 responses for an additional two minutes to simulate a request error. Following this, the entire service is terminated to assess the impact on queued requests, fallback mechanisms, load balancing, and the circuit breaker pattern. This phase lasts 15 minutes, after which normal operation resumes for five minutes.
Next, the Student/Product service begins returning HTTP 429 responses for 20 minutes. After another five-minute normal operation period, the School/Order service is misconfigured to call an incorrect URL for 20 minutes. A rollback is then performed to restore the correct configuration, allowing the service to resume normal operation. A five-minute waiting period is enforced between scenarios.

Spring Down Scenario:
Requests are continuously sent to the system via the Gateway for five minutes before

any fault injection occurs. The Student/Product service is then deliberately stopped by deleting the corresponding pod, ensuring that the service ceases operation entirely. This phase lasts for 20 minutes.
Subsequently, the Student/Product service is restored, becoming accessible again, and requests continue for an additional five minutes before the scenario concludes. A five-minute waiting period is enforced before the next scenario.

Spring Configuration Scenario:
Requests are initiated through the Gateway five minutes before the fault is introduced. An incorrect application.yaml configuration file is then deployed to the Student/Product service, specifying an invalid port, thereby inducing a configuration-related failure. This phase persists for 20 minutes.
Following this, a rollback is performed to restore the correct configuration, allowing the service to resume normal operation. Requests continue for an additional five minutes before the scenario concludes. A five-minute waiting period is enforced between scenarios.

Spring Database Connection Scenario:
Requests are continuously sent to the system via the Gateway for five minutes before any failure is induced. The database schools/order is then made inaccessible without being terminated, simulating a database connection failure. This phase persists for 20 minutes. Subsequently, the database is restored, resuming normal operation for five minutes before being completely shut down. This phase lasts an additional 20 minutes, enabling an evaluation of fallback mechanisms and load balancing across multiple service endpoints. The system then returns to normal operation for another five minutes. A five-minute waiting period is observed between scenarios.
An additional failure condition was initially considered for this scenario, involving an excessive number of database connections. However, due to technical challenges in implementing this condition, it was ultimately omitted.

Kubernetes Pod Unhealthy Scenario:
Requests are continuously sent to the system via the Gateway, starting five minutes before any faults are introduced. The Student/Product pod is then artificially degraded by significantly reducing CPU and memory limits, rendering it unhealthy. This phase lasts for 20 minutes.
Afterward, the service resumes normal operation for five minutes before the pod is explicitly deleted to simulate a pod failure. This phase also persists for 20 minutes. Finally, a

rollback is executed to restore the system to its normal state, which lasts for five minutes. A five-minute waiting period is enforced before the next scenario.

Service Down Kubernetes Scenario:
This scenario mirrors the Kubernetes Pod Unhealthy Scenario. However, it is included separately because, in some cases, the failure injected by the chaos engineering script is classified as a service down kubernetes class by the XGBoost classifier depending on the event/log.

Kubernetes Low CPU/Memory Scenario:
Requests are continuously sent to the system via the Gateway, beginning five minutes prior to any fault injection. The allocated CPU and memory resources are then drastically reduced, allowing for an evaluation of the system's resource allocation mechanisms. This phase lasts for 20 minutes.
Following this, a rollback is performed to restore normal system operation, which continues for five minutes. A five-minute waiting period is enforced before the next scenario.

Kubernetes Invalid Image Scenario:
Requests are initiated through the Gateway five minutes before the fault is introduced. The container image is then replaced with an invalid URL, leading to image verification failures. This phase persists for 20 minutes.
Subsequently, a rollback is executed to restore the system to its normal state, which lasts for five minutes. A five-minute waiting period is enforced before the next scenario.

Kubernetes Configuration Scenario:
Requests are continuously sent to the system via the Gateway, beginning five minutes before any fault is introduced. The Kubernetes deployment is then deliberately misconfigured to attempt access to a non-existent secret, inducing a configuration failure. This phase persists for 20 minutes.
Following this, a rollback is executed to restore the system to its normal state, which lasts for five minutes. A five-minute waiting period is enforced before the next scenario.

The Kubernetes Network and Kubernetes Mount Failure scenarios were also planned but were ultimately not implemented. The network failure scenario was deemed infeasible as it required disabling the CNI plugin, which would necessitate a Minikube cluster restart. This restart would terminate all running services and erase all database data, which was not an acceptable outcome. The mount failure scenario was also not implemented, as persistent volumes in Kubernetes have static access modes. The intended

approach—modifying the volume to read-only to trigger a mount failure—was not technically possible.

Before each scenario, a Minikube cluster is initialized, and all required services are deployed. Additionally, the script deploys a Redis instance, which is utilized by the k6 scripts.
Each scenario begins with the execution of the k6 script, which runs throughout the scenario's duration. This ensures continuous monitoring of system behavior and provides insights into which services remain operational under failure conditions. Concurrently, the chaos engineering script executes and collects relevant data. Upon scenario completion, all collected data is stored, the Minikube cluster is stopped and deleted, and the Redis instance is also shut down.
The data collection process leverages the Kubernetes API to retrieve Kubernetes events and Spring Boot service logs. Additionally, logs, traces, and metrics are collected from Loki, Zipkin, and Prometheus via their respective APIs. This setup ensures that each experiment is executed in isolation, preventing interference between scenarios.

Two distinct k6 scripts were developed. One for the SMS system and another for the EOMS system. Both scripts utilize Redis for storing response data, as this data is required in subsequent requests. Since k6 executes multiple virtual users (VUs) in parallel, a shared storage mechanism such as Redis is necessary to maintain data consistency across requests. Traditional in-script data structures are insufficient, as VUs cannot share memory.
All requests are directed to the system's primary entry point, the Gateway service. The chaos engineering script itself is not deployed within the Minikube cluster; instead, it operates externally, transmitting requests using Kubernetes port forwarding.

The k6 script for the SMS system sends the following requests:

- `POST /api/v1/students`, `POST /api/v1/schools`, and `GET /api/v1/schools/with-students/{school-id}` — 1 request per second.
- `GET /api/v1/students` and `GET /api/v1/schools` — 2 requests per second.

The script includes a setup phase in which a school ID is first stored in Redis. This is essential to ensure that the `POST /api/v1/students` and `GET /api/v1/schools/with-`

`students/{school-id}` requests can be executed successfully, as they rely on a valid school ID. Other request payloads are generated dynamically using the k6 Faker library.

The k6 script for the EOMS system has a similar structure, but is more complex due to the complexity of the EOMS system. It sends requests at the following rates:

- `PUT /api/v1/customers` and `DELETE /api/v1/customers/{id}` — 1 request every 40 seconds.
- `POST /api/v1/customers` and `POST /api/v1/products` — 1 request every 10 seconds.
- `GET /api/v1/customers`, `POST /api/v1/orders`, `GET /api/v1/orders`, `GET /api/v1/orders/{id}`, `GET /api/v1/order-lines/order/{id}`, `GET /api/v1/products`, and `GET /api/v1/products/{id}` — 1 request every 2 seconds.

The setup phase of this script involves sending 10 initial requests to both the `POST /api/v1/customers` and `POST /api/v1/products` endpoints. This ensures that sufficient data is preloaded into both the database and Redis for subsequent requests. Additionally, some endpoints introduce a slight delay before execution to guarantee that the necessary data is available in Redis for constructing valid request payloads.

7 Experimental Results and Analysis

This chapter provides a detailed explanation of the experiment and presents the results of four experiments conducted on the SMS and EOMS systems, both with and without the tool. Finally, the chapter discusses the findings, draws conclusions, and addresses the research question.

7.1 Experimental Procedure

The four experiments were conducted on a MacBook Pro 2021 with an Apple M1 Pro chip, 8-core CPU (6 performance cores and 2 efficiency cores), 16 GB of memory, and running macOS 15.3. Each experiment was executed approximately 10 times, as errors encountered during the runs changes were necessary. The execution of all experiments and scenarios required a total of approximately 480 hours. The final set of experiments each required approximately 12 hours of execution time, resulting in a joint runtime of approximately 48 hours. To prevent the system from entering sleep mode during execution, the caffeinate library was used. Caffeinate ensures that the Mac remains active, thereby preventing interruptions due to system sleep, which is particularly beneficial for long-running computational tasks [37].

Minikube was initialized using the following command:

minikube start –driver=docker –cpus=4 –memory=8192 –disk-size=40g –cni=calico

This command configures Minikube to utilize the Docker driver, allocating 4 CPU cores, 8192 MB of memory, and 40 GB of disk space.

As outlined in the previous chapter, data collection was performed using two distinct methods. The first method involved gathering logs from both the k6 load testing tool and the chaos engineering script. During k6 script execution, logs were recorded in three formats: a plaintext file containing the raw execution logs, time-series data capturing key

metrics alongside timestamps, stored in JSON and CSV formats. The second data collection method involved extracting metrics and logs from Prometheus, Zipkin, Kubernetes events, and system logs.

Following data acquisition, two Python scripts were developed for visualization and analysis. The first script processes k6 execution performance metrics, including response times, error rates, and user activity over time, and generates a summary report. The second script analyzes Prometheus metrics from the Kubernetes cluster, visualizing various system performance indicators such as JVM metrics, deployment statuses, HTTP request metrics, and memory consumption. These visualizations provide insights into system health and performance trends.

7.2 Performance Analysis

In the following sections, the results of the four experiments are presented. The figures illustrate the experimental scenarios; however, the scenarios that are on the Y axis do not correspond to those outlined in Chapter Experimental Setup and Implementation. Instead, the scenarios showed in the figures Y axis represent k6 load-testing scenarios, which are defined as follows: ″Scenarios configure how VUs and iteration schedules in granular detail. With scenarios, you can model diverse workloads, or traffic patterns in load tests.″ [21]

7.2.1 SMS without tool

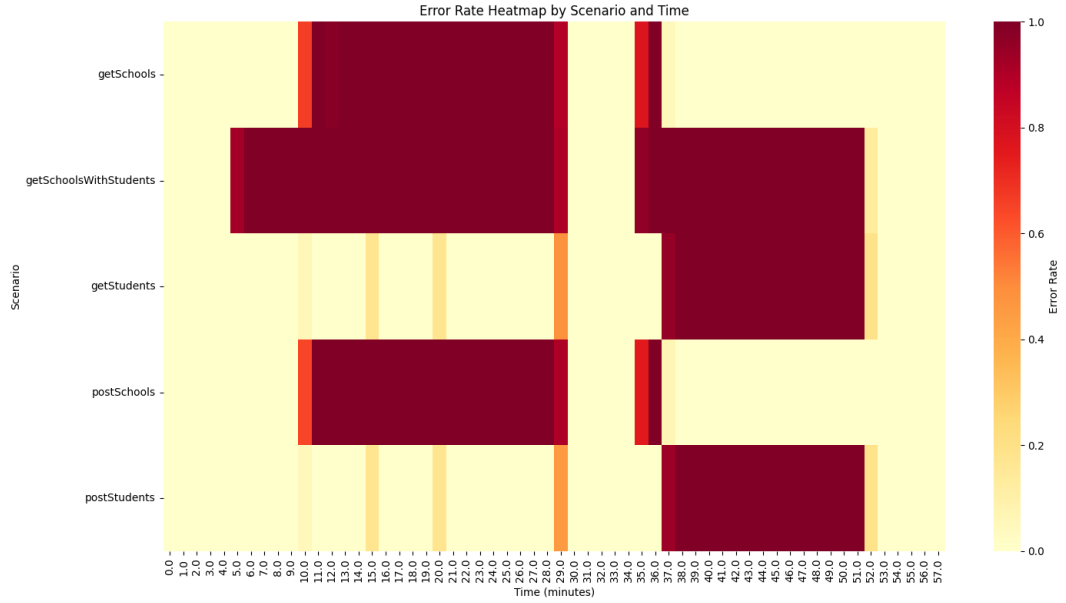

Figure 7.1: Heatmap of the Spring Timeout scenario

The figure 7.1 presents the heat map for the Spring Timeout scenario. During the initial five minutes of the experiment, no errors were observed. This is because no chaos engineering actions were applied during this period. Subsequently, both the getStudents and postStudents endpoints operated as expected, with only minor errors. This behavior arises because the chaos engineering timeout error action affects only the GET /api/v1/students/school/{school-id} endpoint, while the other endpoints do not depend on it. In contrast, the getSchoolsWithStudents endpoint fails because it relies on the response from GET /api/v1/students/school/{school-id}, which experiences excessive delays due to the injected failure. As a result, the getSchools and postSchools endpoints also fail, as the schools service experiences high resource consumption, primarily due to getSchoolsWithStudents waiting indefinitely for responses from the delayed endpoint. Following this period, the system resumed normal operation for five minutes, with no errors, as the chaos engineering action was rolled back. Subsequently, a new chaos engineering action was introduced, leading to minor errors in the postSchools and getSchools endpoints. This behavior is attributed to the implementation of the k6 script, which necessitates the deletion of outdated Redis entries (IDs) because they no longer exist in

the database. New entries must replace the outdated ones, but this process incurs a short delay since new POST requests must be issued to retrieve fresh IDs from the endpoints and overwrite the previous ones. During this transition, errors occur in the GET services as they attempt to access outdated IDs that are no longer present in the database. Subsequently, the students service becomes unavailable due to a chaos engineering action that causes a system failure.
A total of 23,362 requests were executed, of which 8,603 resulted in errors, yielding a failure rate of 36.82%.

The following table A.2 presents the response time statistics.

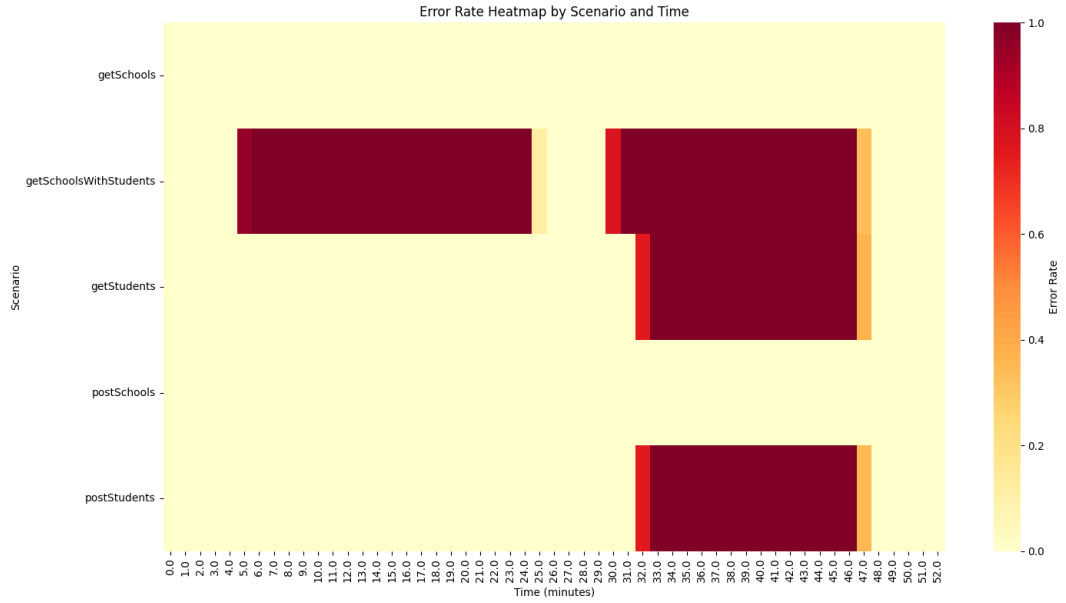

Figure 7.2: Heatmap of the Spring Third-Party Service scenario

The figure 7.2 shows the Spring Third-Party Service scenario. As in the previous scenario, the system operated without errors for the first five minutes, as no chaos engineering actions were executed during this time. In this case, the getSchoolsWithStudents endpoint failed because GET /api/v1/students/school/{school-id} returned an HTTP 500 error. However, all other endpoints remained functional because there was no excessive load on the service. Unlike the Spring Timeout scenario, where the getSchoolsWithStudents endpoint caused significant load by waiting indefinitely for responses, in this case, the failure was immediate due to the HTTP 500 error response. Following this period, the system

returned to normal operation for five minutes after the chaos engineering action was rolled back. In the subsequent chaos engineering action, the students service became unavailable. As a result, all students service endpoints, as well as getSchoolsWithStudents, failed. This occurred because the students service was down, and getSchoolsWithStudents depended on it.
A total of 21,899 requests were executed, with 4,857 errors, corresponding to a failure rate of 22.18%.
The following table A.3 presents the response time statistics.

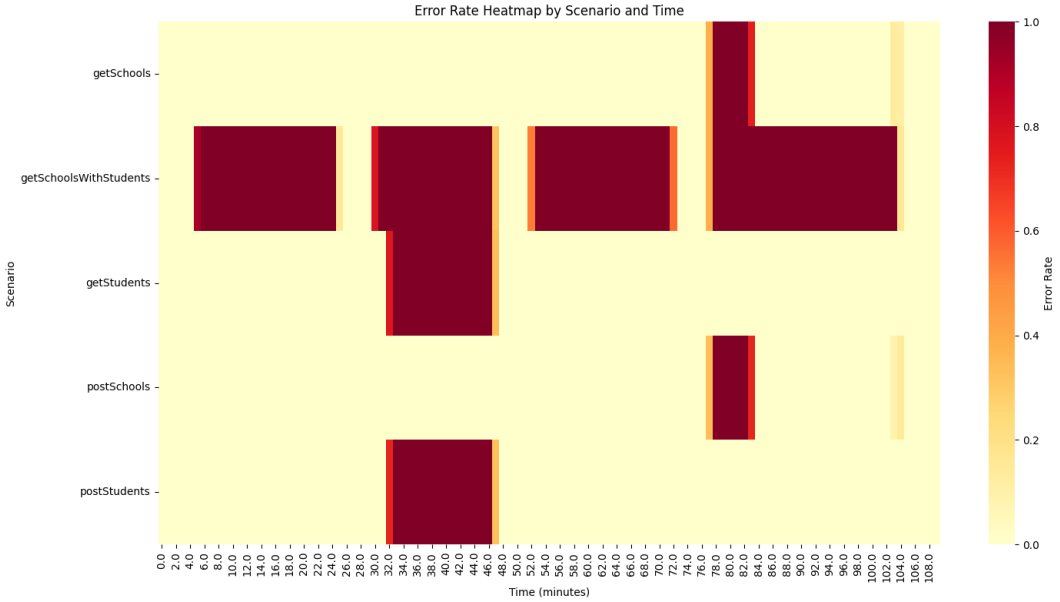

Figure 7.3: Heatmap of the Spring Request scenario

The figure 7.3 presents the heat map for the Spring Request scenario.
As observed in all scenarios, the first five minutes of execution proceed without errors. The getSchoolsWithStudents endpoint fails because GET /api/v1/students/school/{school-id} returns an HTTP 400 error. However, all other endpoints function correctly since the service is not experiencing excessive load; the error is isolated to this specific endpoint, which fails immediately.
Following this, the action is rolled back, and the system operates normally for the next five minutes. When the students service becomes unavailable, all endpoints related to this service, including getSchoolsWithStudents, fail. Since getSchoolsWithStudents depends on the students service, its failure is a direct consequence of the service outage. Once

again, after rolling back the action, the system stabilizes for five minutes without errors. A subsequent failure occurs when GET /api/v1/students/school/{school-id} returns an HTTP 429 error. Similar to the previous failure, the getSchoolsWithStudents endpoint is affected, while the other endpoints remain functional due to the absence of additional service load. After rolling back the action, the system resumes normal operation for another five minutes. During the final chaos engineering action, referred to as wrong_url, the getSchoolsWithStudents endpoint fails as expected, since it cannot retrieve student data from the students service.
Additionally, the getSchools and postSchools endpoints encounter errors due to outdated IDs in Redis, which must be updated with new entries.
A total of 45,617 requests were sent to the system, of which 8,650 resulted in errors, corresponding to a failure rate of 18.96%.

The following table A.4 presents the response time statistics.

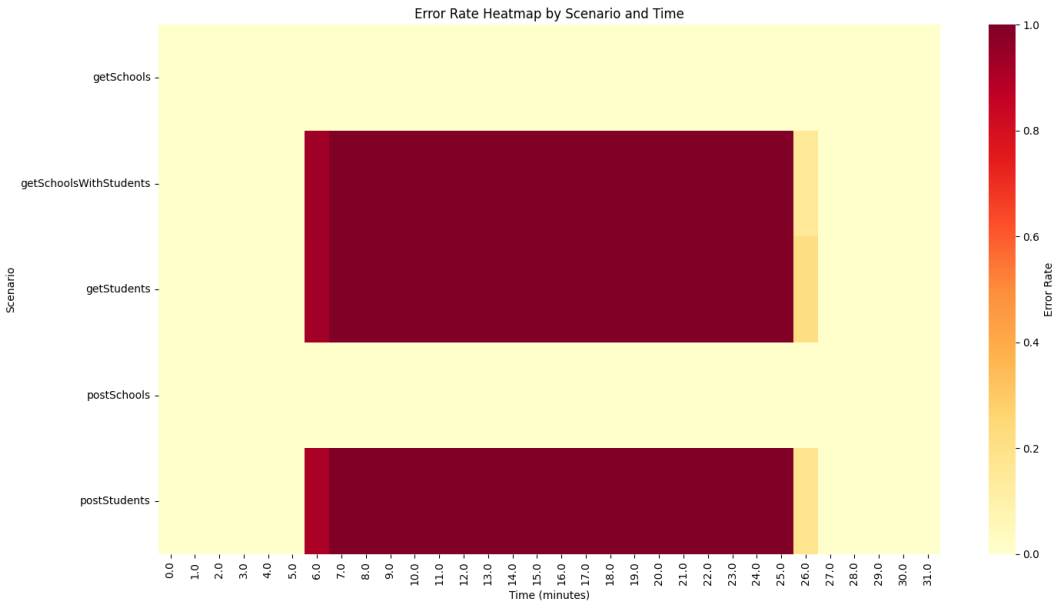

Figure 7.4: Heatmap of the Spring Down scenario

The figure 7.4 illustrates the heat map for the Spring Down scenario.
Similar to previous scenarios, the first five minutes of execution do not exhibit any errors. Since the students service is unavailable, all of its endpoints are non-functional. Additionally, the getSchoolsWithStudents endpoint fails due to its dependency on the

students service.
A total of 12,947 requests were executed, of which 4,662 resulted in errors, yielding an error rate of 36.01%.

The following table A.5 presents the response time statistics.

The figure A.1 presents the heat map for the Spring Configuration scenario.
The results of this scenario closely resemble those of the Spring Down scenario. The students service remains unavailable due to a misconfigured system, leading to the failure of all students service endpoints. As a consequence, the getSchoolsWithStudents endpoint also fails due to its dependency on the students service.
A total of 15,891 requests were sent, of which 6,560 resulted in errors, corresponding to a failure rate of 41.28%.

The table A.6 presents the response time statistics.

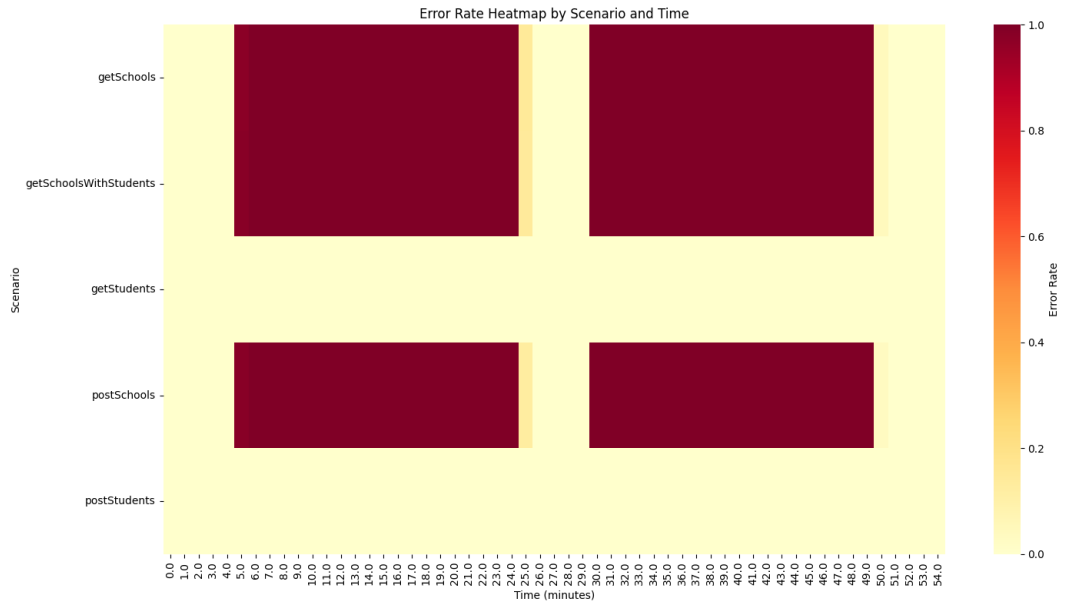

Figure 7.5: Heatmap of the Spring Database Connection scenario

The figure 7.5 illustrates the heat map for the Spring Database Connection scenario. As in previous cases, the first five minutes exhibit no errors. However, as the schools database becomes inaccessible, the schools service is unable to function, causing the failure of all related endpoints.

A total of 22,483 requests were executed, with 9,006 errors, resulting in a failure rate of 40.06%.

The following table A.7 presents the response time statistics.

The figure A.2 shows the Kubernetes Pod Unhealthy scenario.
The initial five minutes are free of errors, consistent with prior scenarios. The results are comparable to those observed in the Spring Database Connection scenario. The schools service becomes unhealthy or down, leading to the failure of all its endpoints.
A total of 22,732 requests were sent, of which 9,386 resulted in errors, yielding an error rate of 41.29%.

The table A.8 presents the response time statistics.

The figure A.3 presents the Service Down Kubernetes Error scenario.
The first five minutes of execution remain error-free, consistent with previous scenarios. The results closely resemble those of the Spring Database Connection scenario. Due to the unavailability or failure of the schools service, all associated endpoints become non-functional.
A total of 22,691 requests were sent, with 9,340 errors, corresponding to a failure rate of 41.16%.

The table A.9 presents the response time statistics.

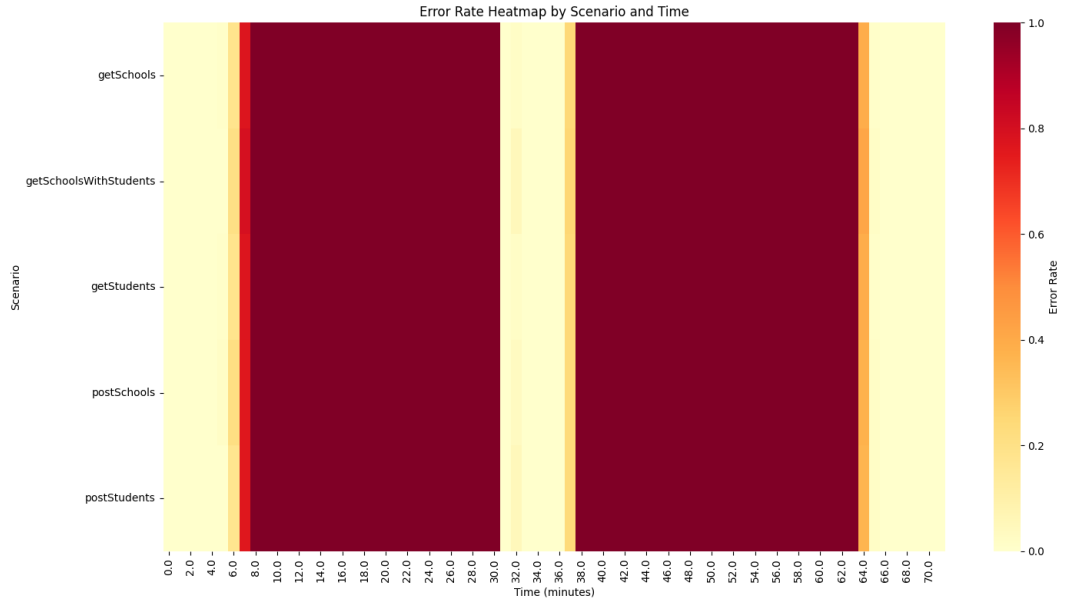

Figure 7.6: Heatmap of the Kubernetes Node Problem scenario

The figure 7.6 illustrates the Kubernetes Node Problem scenario.
As in previous scenarios, the first five minutes of execution proceed without errors. During both chaos engineering actions, all services become unavailable due to disruptions in the Kubernetes node on Minikube.
A total of 27,718 requests were executed, of which 19,236 resulted in errors, yielding an error rate of 69.40%.

The table A.10 presents the response time statistics.

Unfortunately, the Kubernetes Mount Fail Scenario could not be implemented. It is not possible to modify the database mount at runtime. The intended goal was to make the mount read-only; however, this approach is not possible. Once a Persistent Volume (PV) is created, its access modes remain static. Additionally, the ReadOnlyMany mode does not permit write operations, and an init container cannot modify a read-only filesystem.

Disabling the Container Network Interface (CNI) could not be implemented, that's why Kubernetes Network Error Scenario was not executed. If the CNI is disabled, kubectl also becomes inoperable, requiring the termination of all pods and their subsequent redeployment.

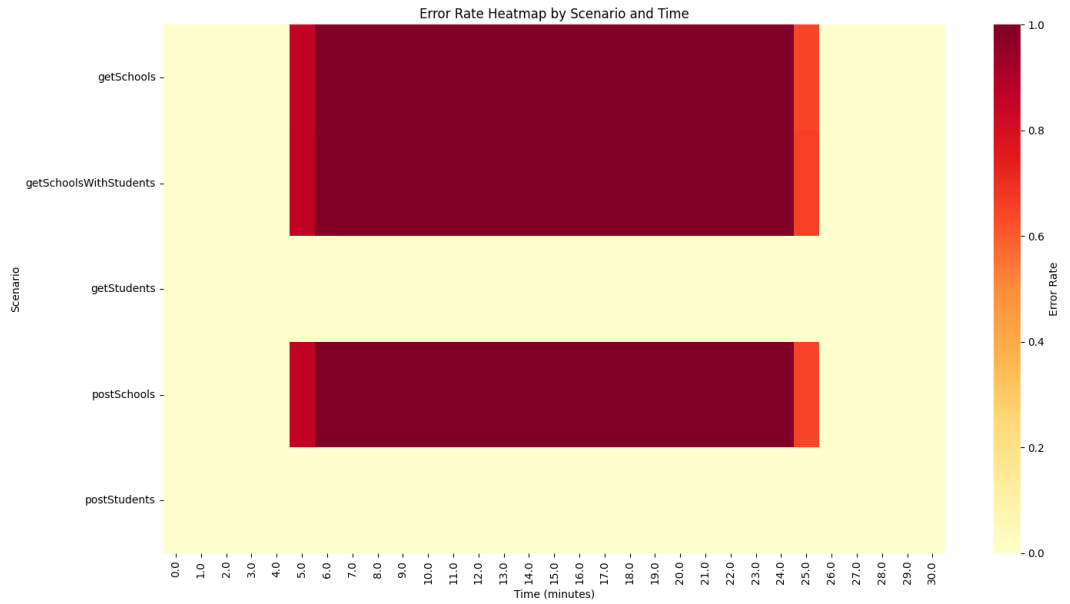

Figure 7.7: Heatmap of the Kubernetes Low CPU Memory scenario

The figure 7.7 presents the Kubernetes Low CPU Memory scenario.
Similar to previous scenarios, the first five minutes exhibit no errors. The schools service experiences resource constraints, causing the service to become unavailable, thereby affecting all its endpoints.
A total of 12,488 requests were processed, with 4,716 errors, resulting in a failure rate of 37.76%.

The following table A.11 presents the response time statistics.

The figure A.4 illustrates the Kubernetes Invalid Image scenario.
This scenario closely resembles the Kubernetes Low CPU Memory scenario. Since the schools service is unavailable, all its endpoints fail.
A total of 12,352 requests were sent, of which 4,581 resulted in errors, corresponding to a failure rate of 37.09%.

The table A.12 presents the response time statistics.

The figure A.5 presents the Kubernetes Configuration scenario.
The results of this scenario are similar to those observed in the Spring Configuration scenario. The students service fails due to a misconfiguration, causing all its endpoints

to be non-functional. As a consequence, the getSchoolsWithStudents endpoint also fails, as it depends on the students service.
A total of 12,575 requests were executed, with 4,659 errors, yielding a failure rate of 37.05%.

The table A.13 presents the response time statistics.

7.2.2 EOMS without tool

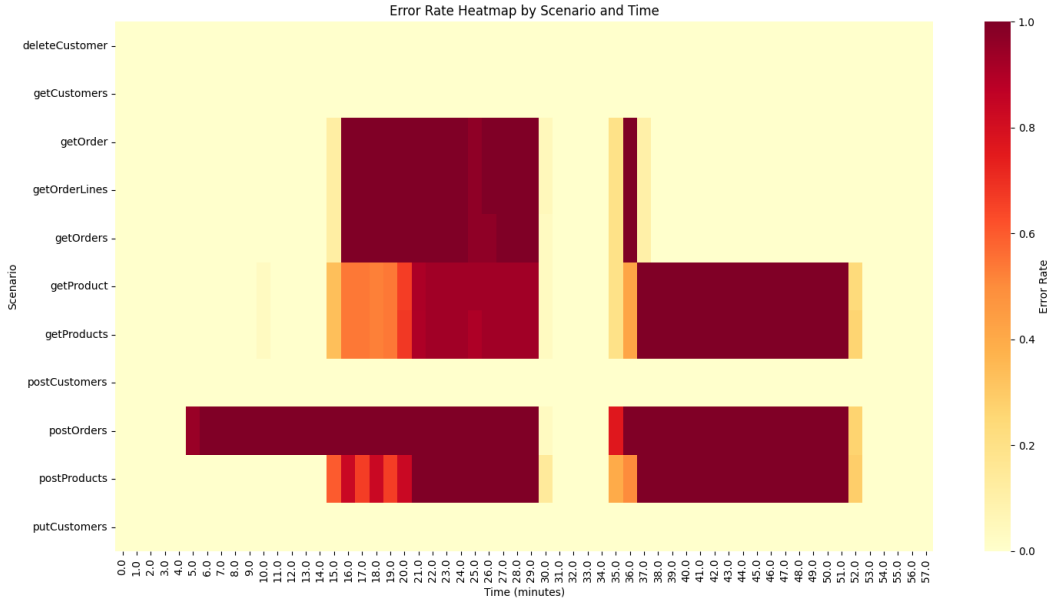

Figure 7.8: Heatmap of the Spring Timeout scenario

The figure 7.8 illustrates the heat map for the Spring Timeout scenario.
During the initial five minutes of execution, no errors are observed, as no chaos engineering actions have been introduced. Subsequently, errors appear in the getProduct, getProducts, and postProducts endpoints.
This is expected, as only POST /api/v1/products/purchase is directly affected by the timeout error, while the aforementioned endpoints do not invoke it. The postOrders endpoint fails because it waits too long for a response from POST /api/v1/products/purchase. As a result, excessive load accumulates on the products service, leading to failures in getProduct, getProducts, and postProducts. A similar pattern is observed in the orders

service, as it experiences excessive load due to the failing postOrders service. Following this phase, the chaos engineering action is rolled back, and the system resumes normal operation for five minutes. A subsequent chaos engineering action leads to minor errors in getOrders, getOrderLines, and getOrders, attributed to the previously discussed k6 script. Eventually, the products service fails completely due to the chaos engineering action, rendering it non-functional.
A total of 20,153 requests were processed, with 4,778 errors, corresponding to a failure rate of 23.71%.
The following table A.14 presents the response time statistics.

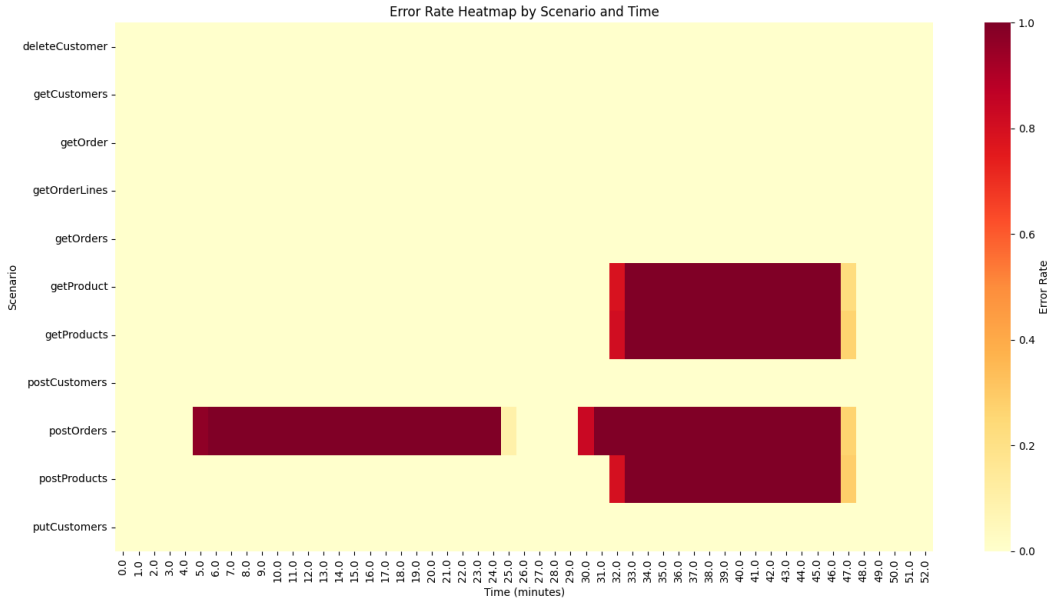

Figure 7.9: Heatmap of the Spring Third-Party Service scenario

The figure 7.9 illustrates the Spring Third-Party Service scenario.
As observed in other scenarios, the first five minutes proceed without errors, as no chaos engineering actions are applied. The postOrders endpoint fails because POST /api/v1/products/purchase returns an HTTP 500 error. Since the service is not under significant load, all other endpoints continue to function normally. However, the postOrders request fails immediately upon receiving the HTTP 500 response. Following this failure, the chaos engineering action is rolled back, and the system operates normally for five minutes.
In the second chaos engineering scenario, we observe that the products service is down.

As a result, all product-related endpoints, as well as the postOrders endpoint, fail. This occurs because postOrders depends on the products service, which is unavailable.
A total of 20,627 requests were sent, with approximately 2,488 errors, resulting in an error rate of 12.06%.
The following table A.15 presents the response time statistics.

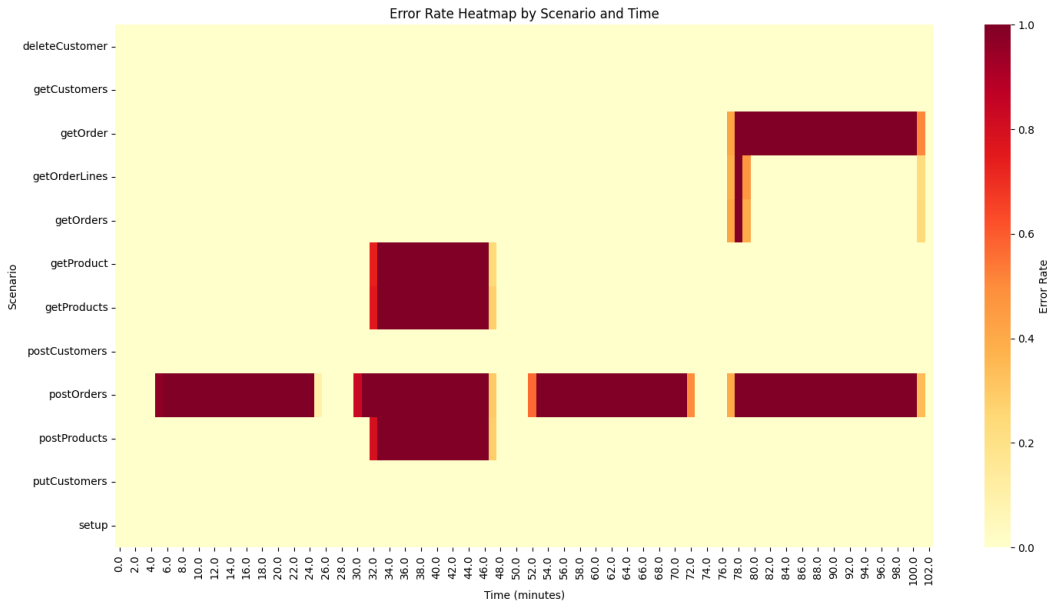

Figure 7.10: Heatmap of the Spring Request scenario

The figure 7.10 illustrates the Spring Request scenario.
As in every scenario, the first five minutes proceed without errors. The postOrders endpoint fails because POST /api/v1/products/purchase returns an HTTP 400 error. However, since the service is not under load, all other endpoints continue to function normally.
Following this failure, the chaos engineering action is rolled back, and the system operates normally for five minutes.
While the products service remains down, all product-related endpoints and postOrders remain non-functional, as postOrders depends on the products service.
Once again, the action is rolled back, and the system runs without errors for five minutes.
Later in the experiment, the postOrders endpoint fails again, this time due to POST /api/v1/products/purchase returning an HTTP 429 error. All other endpoints continue to function normally as there is no significant load on the service. After another rollback,

the system operates error-free for five minutes.
During the wrong_url scenario, the postOrders endpoint fails as expected because it cannot retrieve customer data from the customers service. Additionally, getOrderLines and getOrders exhibit errors because outdated IDs in Redis must be replaced with new ones.
The getOrder endpoint remains non-functional throughout the scenario. This is because the orders service is restarted after the URL change. Restarting is necessary to apply the configuration change, but doing so results in the loss of all database entities. Furthermore, since postOrders continues to fail, no new orders are created, preventing old orders in Redis from being overwritten. This behavior was deliberately left unchanged in the experiment, as it represents a realistic scenario in real-world applications.
A total of 41,025 requests were sent, of which 5,451 resulted in errors, yielding an error rate of 13.29%.
The following table A.16 presents the response time statistics.

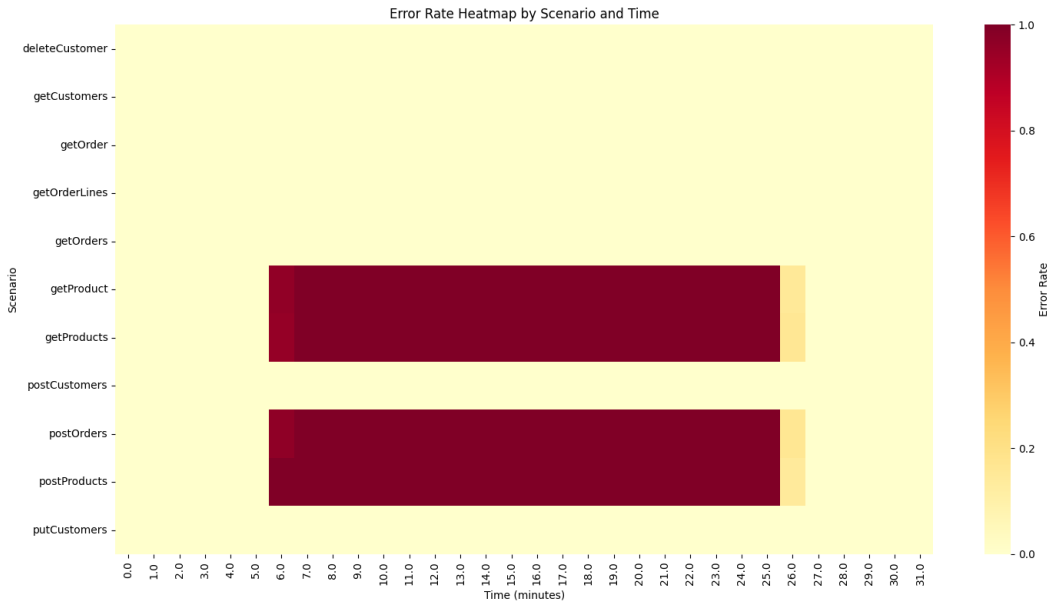

Figure 7.11: Heatmap of the Spring Down scenario

The following figure 7.11 presents the heat map for the Spring Down scenario.
As in other scenarios, the first five minutes proceed without errors. During the chaos engineering action, the products service is down, causing all its endpoints to fail. Consequently, the postOrders endpoint is also non-functional, as it depends on the products

service.
A total of 11,949 requests were sent, with 2,491 errors, yielding an error rate of 20.85%.

The following table A.17 presents the response time statistics.

The figure A.6 illustrates the heat map for the Spring Configuration scenario.
The results of this scenario closely resemble those of the Spring Down scenario. The products service fails due to a misconfiguration, rendering all product-related endpoints non-functional. Since postOrders depends on the products service, it also fails.
In this scenario, 14,347 requests were sent, of which 3,319 resulted in errors, yielding an error rate of 23.13%.

The table A.18 presents the response time statistics.

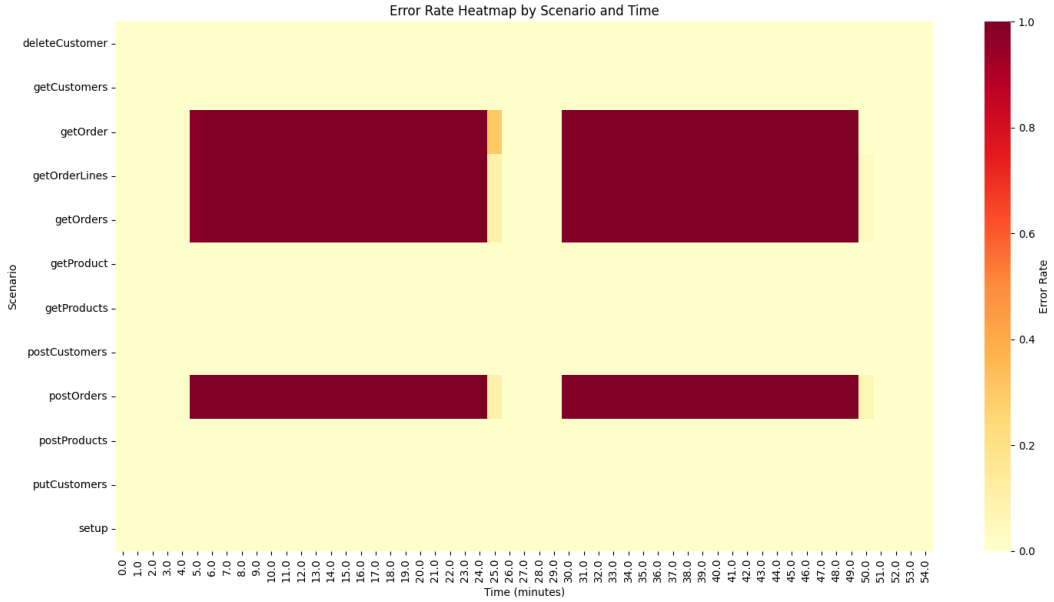

Figure 7.12: Heatmap of the Spring Database Connection scenario

The figure 7.12 illustrates the Spring Database Connection heat map.
As in other scenarios, the first five minutes proceed without errors. Since the orders database is inaccessible and completely unavailable, the orders service cannot function, causing all order-related endpoints to fail.
A total of 19,111 requests were sent, with 5,323 errors, resulting in an error rate of 27.85%.

The following table A.19 presents the response time statistics.

The figure A.7 illustrates the Kubernetes Pod Unhealthy scenario.
As in previous scenarios, the first five minutes proceed without errors. The results closely resemble those of the Spring Database Connection scenario. The orders service becomes unhealthy and stops functioning, causing all its endpoints to fail.
A total of 19,891 requests were sent, with 6,251 errors, resulting in an error rate of 31.43%.

The table A.20 presents the response time statistics.

The figure A.8 illustrates the Service Down Kubernetes scenario.
As in other cases, the first five minutes run without errors. The results closely mirror those of the Spring Database Connection scenario. The orders service is either down or unhealthy, rendering all its endpoints non-functional.
A total of 20,022 requests were sent, with 6,378 errors, yielding an error rate of 31.85%.

The table A.21 presents the response time statistics.

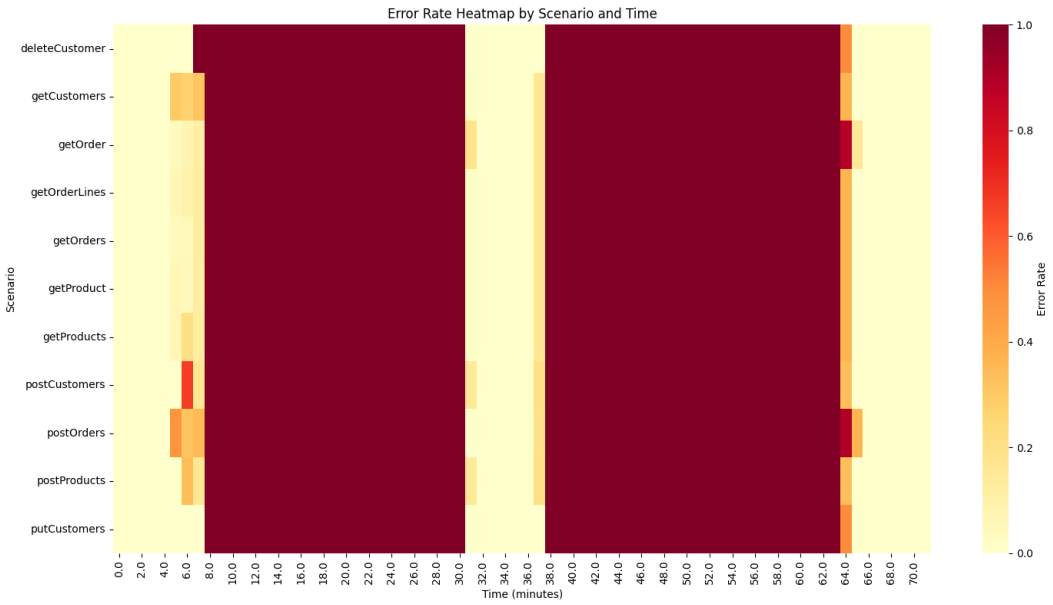

Figure 7.13: Heatmap of the Kubernetes Node Problem scenario

The figure 7.13 illustrates the Kubernetes Node Problem scenario.
The first five minutes are error-free, similar to other scenarios. During both chaos engi-

neering actions, all services go down due to disturbances affecting the Kubernetes node in Minikube.
A total of 24,097 requests were sent, with 15,946 errors, leading to an error rate of 66.17%.

The following table A.22 presents the response time statistics.

As already mentioned the Kubernetes Mount Fail and Kubernetes Network Error Scenarios could not be implemented.

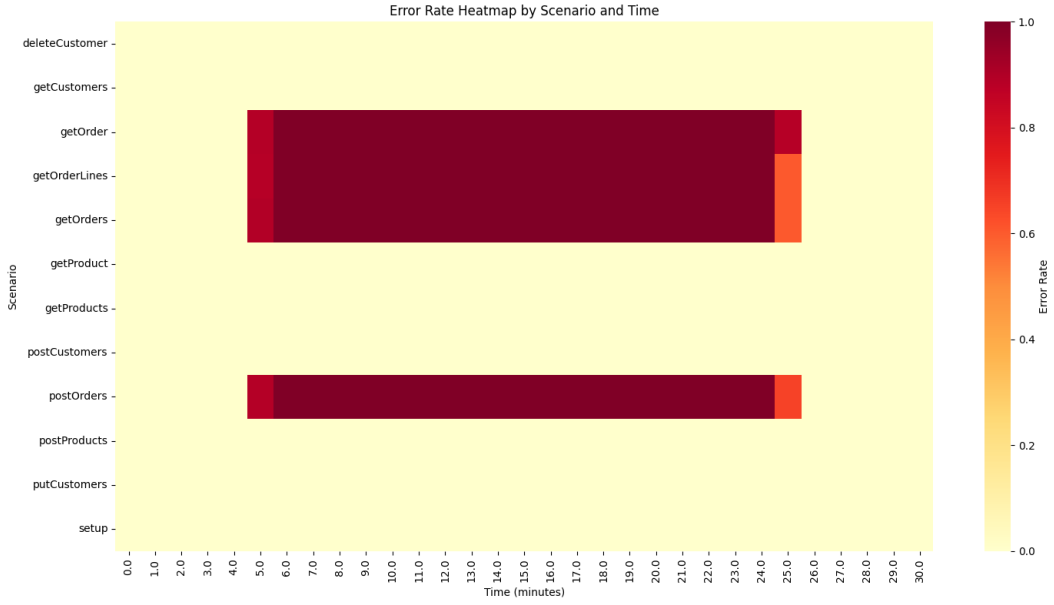

Figure 7.14: Heatmap of the Kubernetes Low CPU and Memory scenario

The figure 7.14 illustrates the Kubernetes Low CPU and Memory scenario.
As in previous cases, the first five minutes are error-free. Due to resource constraints, the orders service becomes unresponsive, causing all its endpoints to fail.
A total of 10,926 requests were executed, with 3,061 errors, yielding an error rate of 28.02%.

The following table A.23 presents the response time statistics.

The figure A.9 illustrates the Kubernetes Invalid Image scenario.
This scenario behaves similarly to the Kubernetes Low CPU and Memory scenario. The

orders service fails due to an invalid image, causing all its endpoints to become non-functional.
A total of 10,623 requests were executed, with 2,752 errors, leading to an error rate of 25.91%.

The table A.24 presents the response time statistics.

The figure A.10 illustrates the Kubernetes Configuration scenario.
The results are similar to those of the Spring Configuration scenario. A misconfiguration in the products service causes it to fail, which in turn prevents the postOrders endpoint from functioning, as it depends on the products service.
A total of 11,373 requests were executed, with 2,254 errors, resulting in an error rate of 19.82%.

The table A.25 presents the response time statistics.

7.2.3 SMS with tool

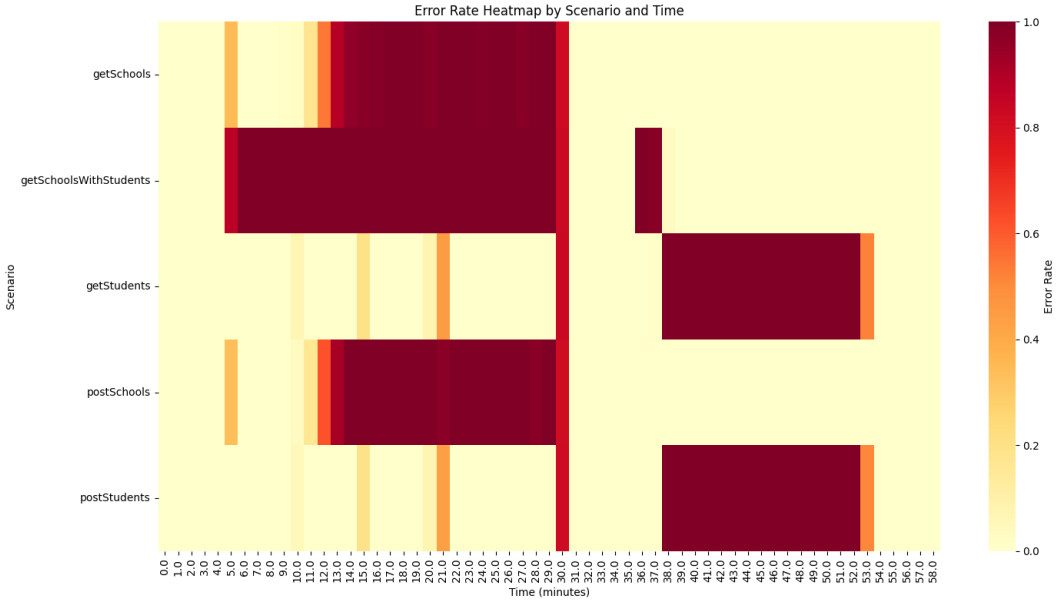

Figure 7.15: Heatmap of the Spring Timeout scenario with tool

The figure 7.15 illustrates the heat map for the Spring Timeout scenario with the tool deployed.
Compared to the scenario without the tool, the getSchoolsWithStudents endpoint remains affected by the timeout. This is because the tool could not apply any pattern that would resolve the error. An analysis of the tool's logs indicates that it attempted to apply the fallback mechanism but was unsuccessful. Further investigation revealed a small bug in the tool that caused this issue in both this scenario and the Spring Timeout scenario in the EOMS system. Since these two scenarios were the last to be executed, and minor modifications were made to the tool before execution, this bug was introduced at that point. This explains why the issue did not occur in other experiments.
During the second chaos engineering action, the student service fails entirely, rendering all student-related endpoints non-functional. At this point, the getSchoolsWithStudents endpoint stops failing because the tool detects the error and successfully executes the fallback mechanism.
A total of 24,019 requests were processed, with 7,596 errors, corresponding to a failure rate of 31.62%. This indicates that the tool was able to reduce the error rate by 14.12%. The response time statistics are presented in table A.26.

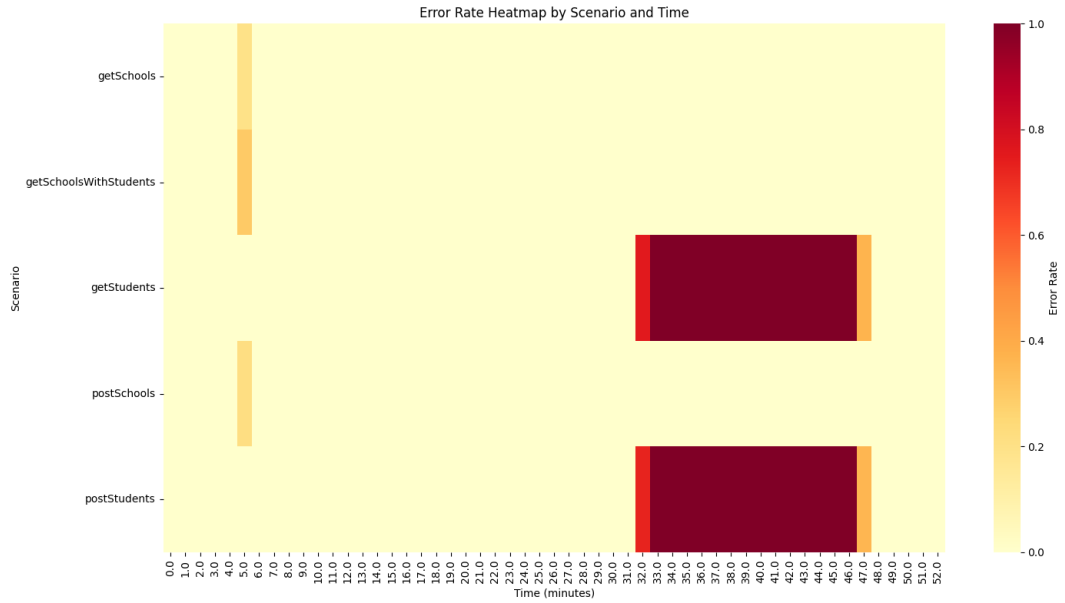

Figure 7.16: Heatmap of the Spring Third-Party Service scenario with tool

The figure 7.16 presents the Spring Third-Party Service scenario combined with the deployed tool.
In comparison to figure 7.2, the results indicate that the getSchoolsWithStudents endpoint remains unaffected by the chaos engineering action. This behavior is attributed to the tool's execution of the fallback mechanism, which successfully handled the error and returned a response to the client. Student, postStudents and getStudents endpoints were non-functional due to the unavailability of the student service.
The tool was unable to redeploy the service, as the chaos engineering action was deliberately designed to render it undeployable. During the experiment, a total of 21,818 requests were issued, of which 2,602 resulted in errors, corresponding to an error rate of 11.93%. This demonstrates that the tool effectively leveraged the fallback mechanism to mitigate 500 errors, thereby reducing the error rate by 46.21%.
The response time statistics are provided in table A.27.

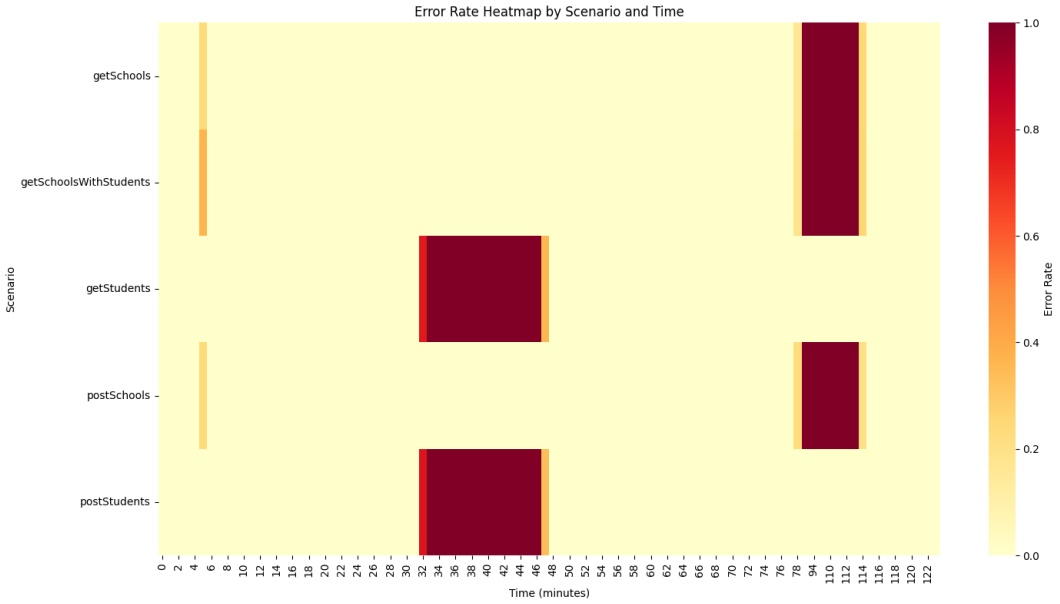

Figure 7.17: Heatmap of the Spring Request scenario with tool deployed

Figure 7.17 depicts the Spring Request scenario in conjunction with the deployed tool. Similar to the Spring Third-Party Service scenario, the getSchoolsWithStudents endpoint remains unaffected by the chaos engineering action. Again, this can be attributed to the tool's execution of the fallback mechanism, which successfully handled the error and provided a response to the client. Notably, this mechanism was effective for both

400 and 429 errors.
The postStudents and getStudents endpoints experienced temporary outages due to the complete unavailability of the student service. As previously mentioned, in this specific context, the tool lacks the capability to recover unavailable services. During the execution of the wrong_url scenario, the getSchoolsWithStudents endpoint initially fails, as expected, since it cannot retrieve data from the GET /api/v1/students/school/{school-id} endpoint. However, after some time, the endpoint resumes functionality as the tool deploys its fallback mechanism to handle the error and return a response to the client. Once again, the tool employs its fallback mechanism.
A total of 38,849 requests were sent, of which 4,033 resulted in errors, yielding an error rate of 10.38%. These results indicate that the tool effectively utilized the fallback mechanism to handle 400 and 429 errors, ultimately reducing the error rate by 45.25%.
The response time statistics are provided in table A.28.

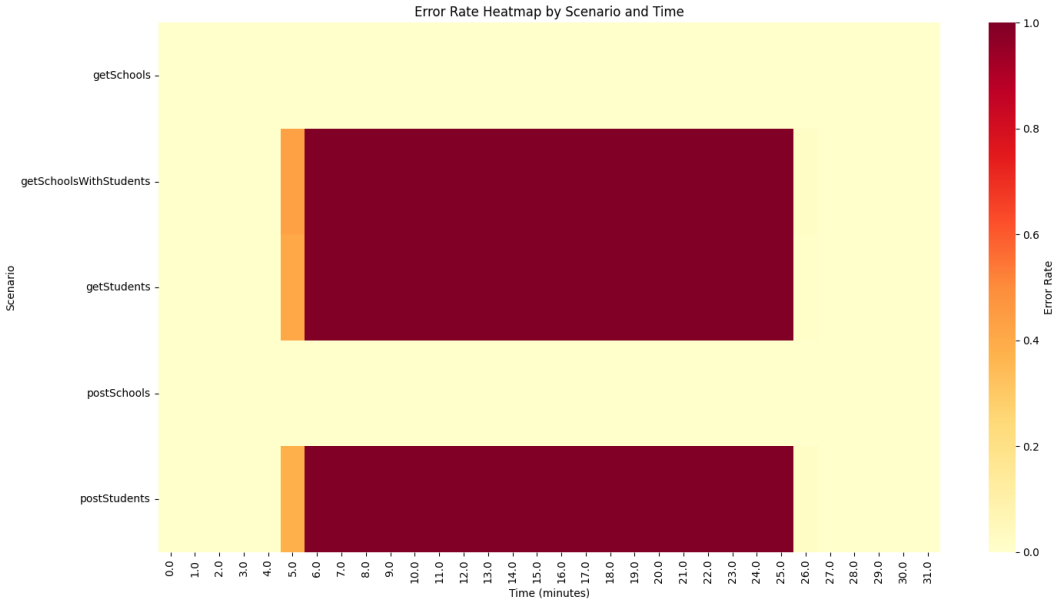

Figure 7.18: Heatmap of the Spring Down scenario with tool deployed

Figure 7.18 illustrates the heatmap for the Spring Down scenario with the tool deployed. A comparison with the corresponding scenario conducted without the tool reveals that the results are nearly identical. In this case, the tool failed to categorize the error using the classification algorithm and was consequently unable to execute a corrective solution. In this scenario, a total of 12,831 requests were issued, with 4,683 resulting in errors,

corresponding to an error rate of 36.50%. Thus, the tool was ineffective in reducing the error rate.
The response time statistics are presented in table A.29.

Figure A.11 depicts the heatmap for the Spring Configuration scenario with the tool deployed.
As observed in the Spring Down scenario, the results closely resemble those obtained in the absence of the tool. Once again, the tool was unable to categorize the error using the classification algorithm and could not execute a corrective solution.
In this case, a total of 15,559 requests were sent, of which 6,232 resulted in errors, yielding an error rate of 40.05%. Similar to the Spring Down scenario, the tool was ineffective in reducing the error rate.
The response time statistics are summarized in table A.30.

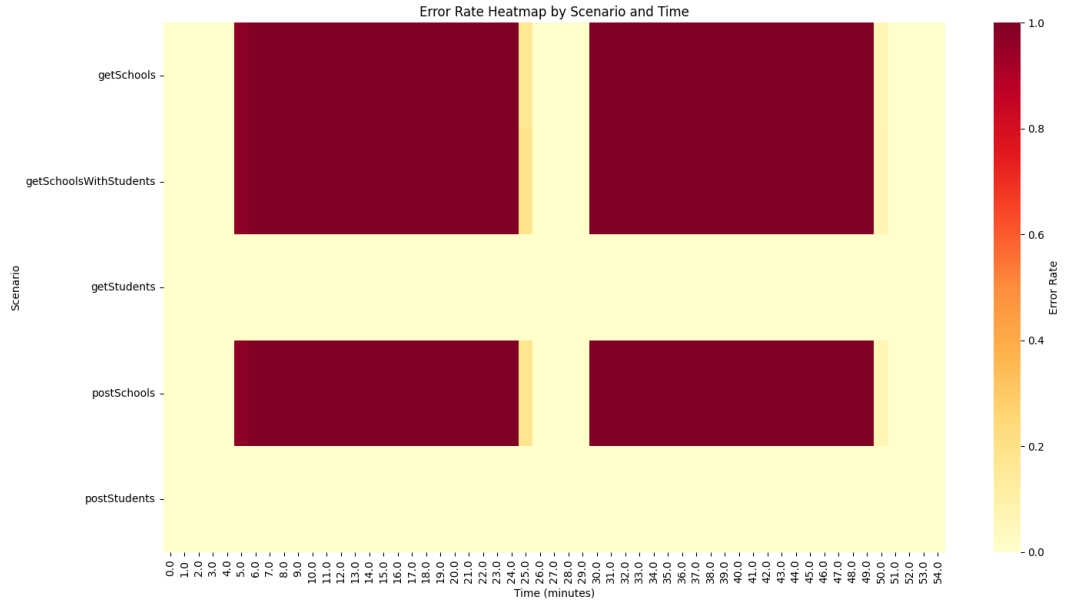

Figure 7.19: Heatmap of the Spring Database Connection scenario with tool deployed

The figure 7.19 illustrates the heat map for the Spring Database Connection scenario with the tool deployed.
Similar to the two previous scenarios, the results closely resemble those obtained without the tool. This is because, in this scenario, the tool also failed to categorize the error using the classification algorithm and was unable to execute a corrective solution.

A total of 22,459 requests were executed, with 8,992 errors, resulting in a failure rate of 40.04%. The tool was ineffective in reducing the error rate.
The response time statistics are presented in table A.31.

Figure 7.20: Heatmap of the Kubernetes Pod Unhealthy scenario with tool deployed

The figure 7.20 illustrates the Kubernetes Pod Unhealthy scenario, demonstrating that all request errors were successfully resolved.
The tool detected high CPU and memory usage on the affected pod and applied the cpu memory limits and health solution. This solution increased the available CPU and memory resources while restoring the health endpoint, thereby making the service operational again. As a result, the schools pod was able to restart and respond to incoming requests from the getSchoolsWithStudents endpoint.
During this scenario, a total of 23,226 requests were issued, with 228 resulting in errors, yielding an error rate of 0.98%. In comparison to the scenario without the tool, the error rate was reduced by 97.62%.
The response time statistics are provided in table A.32.

The figure A.12 presents the Service Down Kubernetes scenario with the deployed tool. The tool detected excessive CPU and memory usage on the pod and applied the cpu memory limits and health solution, to restore service availability. The results closely

resemble those observed in the Kubernetes Pod Unhealthy scenario.
During this scenario, a total of 23,223 requests were issued, with 229 resulting in errors, corresponding to an error rate of 0.99%. Compared to the scenario without the tool, the error rate was reduced by 97.59%.
The response time statistics are presented in table A.33.

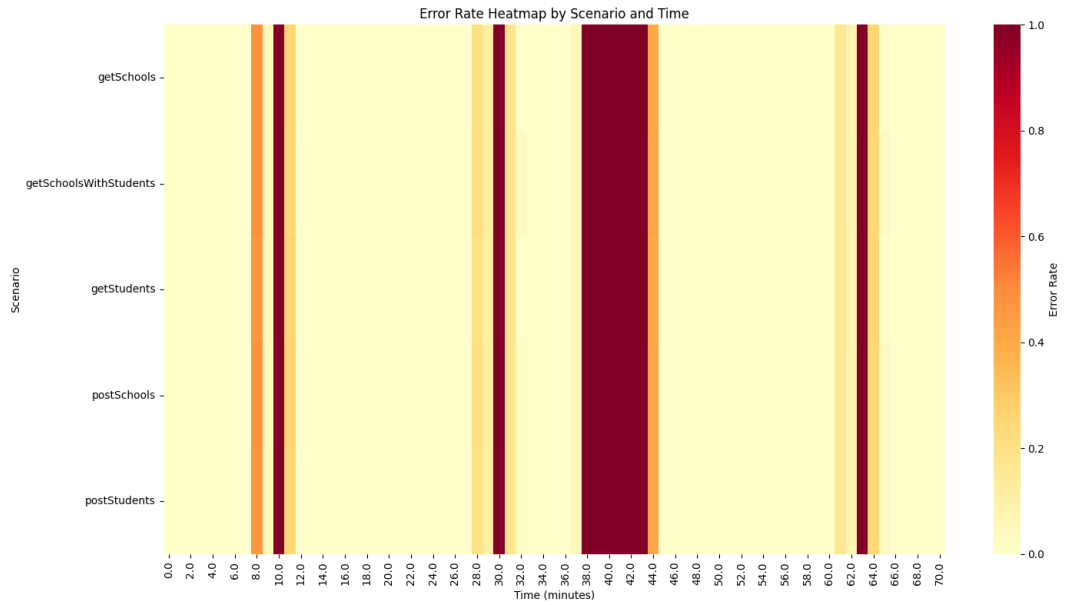

Figure 7.21: Heatmap of the Kubernetes Node Problem scenario with tool deployed

The figure 7.21 illustrates the Kubernetes Node Problem scenario.
The results indicate that the tool successfully mitigated both chaos engineering actions. The first action involved exhausting disk space. The tool detected the low disk availability and executed a free disk space solution to free space. This process required a certain amount of time, as deleting sufficient data to free disk space is not instantaneous. At approximately the 28-minute mark, errors reappeared due to the rollback of the chaos engineering action, which resulted in the termination of all pods, causing temporary failures. The second action involved artificially reducing the available CPU and memory on the node. The tool detected the resource constraints and allocated additional CPU and memory to restore normal operation. Similarly, at approximately 61 minutes, errors reoccurred as the chaos engineering script initiated a system rollback, leading to temporary service disruptions.
A total of 29,755 requests were executed, of which 4,761 resulted in errors, yielding an

error rate of 16.00%. This demonstrates that the tool successfully reduced the error rate by 76.94%.
The response time statistics are provided in table A.34.

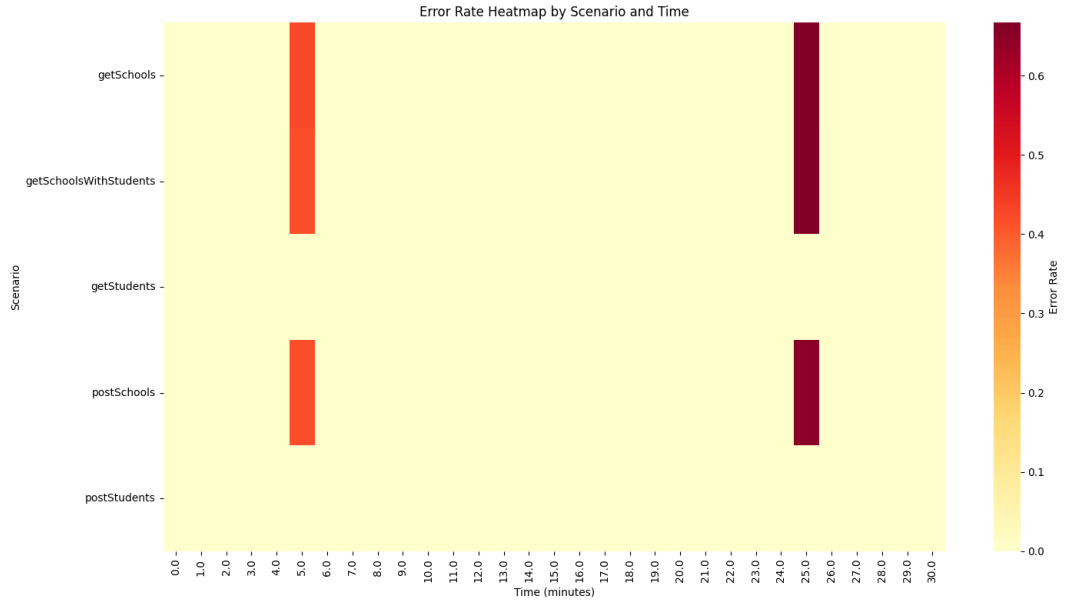

Figure 7.22: Heatmap of the Kubernetes Low CPU Memory scenario with tool deployed

The figure 7.22 presents the Kubernetes Low CPU Memory scenario with the tool deployed.
The results show that all errors were successfully mitigated by the tool. Upon detecting the Kubernetes event, the tool executed the memory, cpu limits solution, increasing the available resources for the affected pod. As a result, the pod was able to handle requests not only from the getSchoolsWithStudents endpoint but also from the getSchools and postSchools endpoints.
A total of 12,757 requests were processed, with 257 resulting in errors, corresponding to a failure rate of 2.01%. This indicates that the tool effectively reduced the error rate by 94.67%.
The response time statistics are summarized in table A.35.

The figure A.13 illustrates the Kubernetes Invalid Image scenario with the tool deployed. The results indicate that all errors were successfully resolved by the tool. Upon detecting the Kubernetes event, the tool executed the verify image solution, which verifies the

image tag and updates it to latest if it is incorrectly specified. As a result, the pod was successfully created and became responsive to all endpoints.
A total of 12,614 requests were processed, with 50 resulting in errors, yielding a failure rate of 0.40%. This demonstrates that the tool effectively reduced the error rate by 98.92%.
The response time statistics are provided in table A.36.

The figure A.14 presents the Kubernetes Configuration scenario.
The results show that all errors were successfully mitigated by the tool. Upon detecting the Kubernetes event, the tool executed the Kubernetes YAML formatting solution, which corrected the YAML configuration format. Following this intervention, the pod was successfully created and became fully responsive across all endpoints.
A total of 12,860 requests were executed, with 135 resulting in errors, corresponding to a failure rate of 1.05%. This indicates that the tool reduced the error rate by 97.16%.
The response time statistics are presented in table A.37.

7.2.4 EOMS with tool

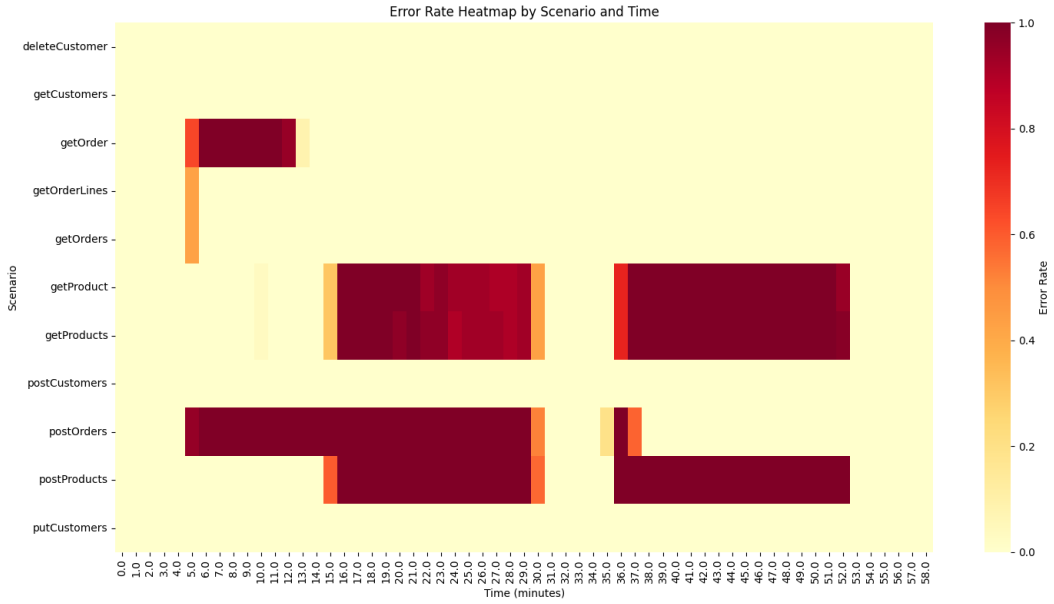

Figure 7.23: Heatmap of the Spring Timeout scenario with tool deployed

The figure 7.23 illustrates the heat map for the Spring Timeout scenario with the tool deployed.
Compared to the scenario without the tool, the postOrders endpoint remains affected by the timeout. This is because the tool had a bug that caused the fallback mechanism to not be applied as already mentioned. However, the getOrder, getOrderLine, and getOrders endpoints remain unaffected by the chaos engineering action. This is due to the tool detecting prolonged request delays when postOrders calls POST /api/v1/products/purchase, and subsequently terminating the request after 10 seconds. By identifying excessive wait times, the tool prevents postOrders from consuming all available resources, thereby avoiding an accumulated overload that could affect other endpoints.
At the 5-minute mark, an anomaly occurs in the getOrder endpoint. This anomaly is likely caused by the k6 script, but the tool was unable to categorize it. During the second chaos engineering action, the product service fails entirely, rendering all product-related endpoints non-functional. At this point, the postOrders endpoint stops failing because the tool detects the error and executes the fallback mechanism.
A total of 21,735 requests were processed, with 3,567 errors, corresponding to a failure rate of 16.41%. This indicates that the tool was able to reduce the error rate by 30.78%. The response time statistics are presented in table A.38.

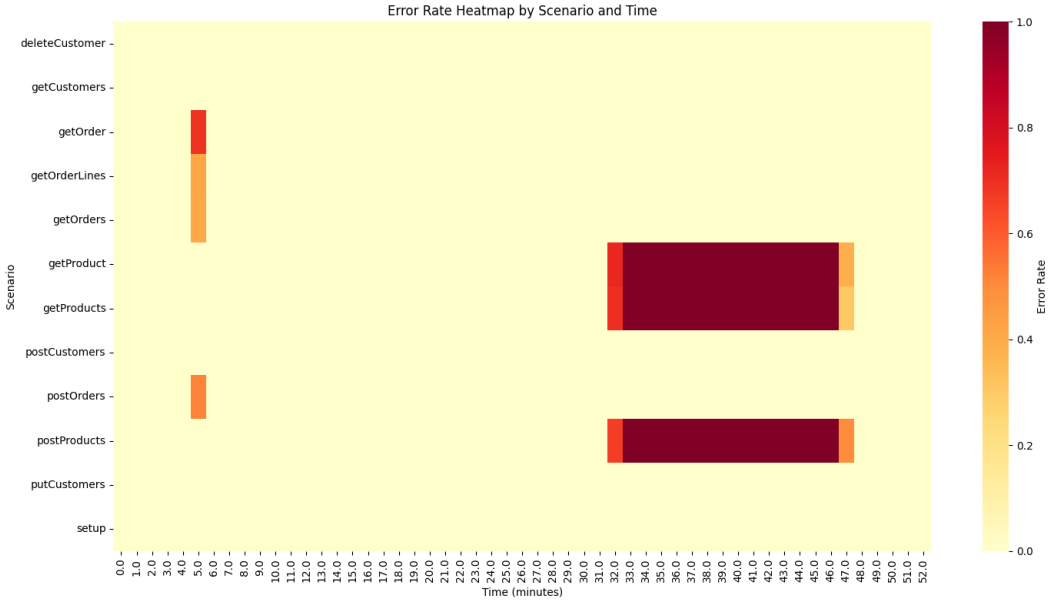

Figure 7.24: Heatmap of the Spring Third-Party Service scenario with tool

The figure 7.24 illustrates the Spring Third-Party Service scenario in combination with the deployed tool.
The results indicate that the postOrders endpoint remains unaffected by the chaos engineering actions throughout the execution. As observed in the SMS case with the tool deployed, the tool successfully detected the error in the Spring Boot service and responded by employing the fallback mechanism. However, the product-related endpoints postProducts, getProduct, and getProducts, remained non-functional due to the unavailability of the product service, which the tool is not designed to recover.
A total of 20,627 requests were sent, of which 1,460 resulted in errors, yielding an error rate of 7.08%. This demonstrates that the tool effectively utilized the fallback mechanism, reducing the error rate by 41.29%.
The response time statistics are presented in table A.39.

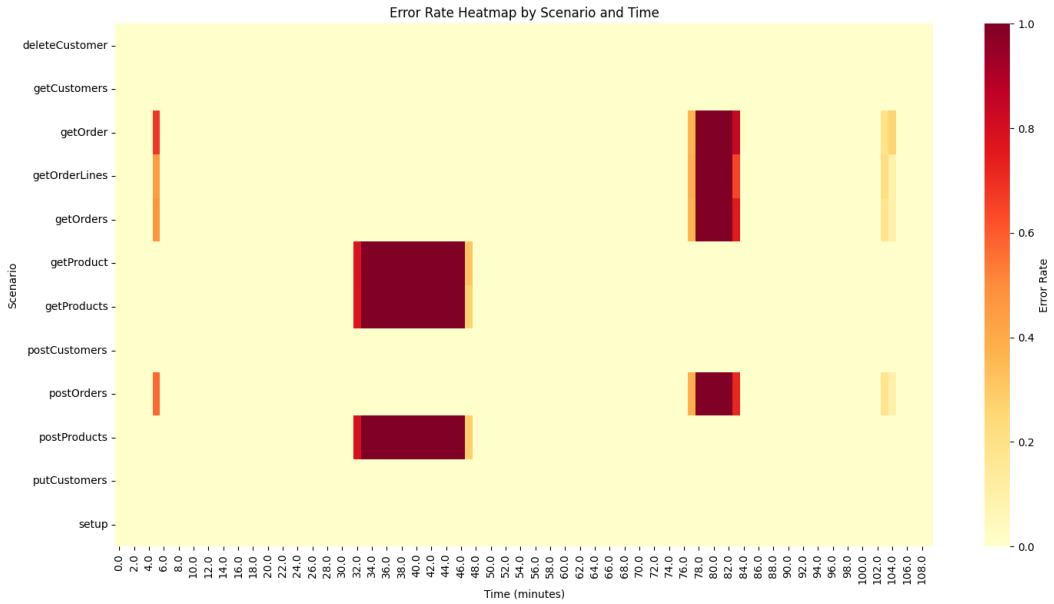

Figure 7.25: Heatmap of the Spring Request scenario with tool deployed

The figure 7.25 illustrates the Spring Request scenario with the deployed tool.
Similar to the Spring Third-Party Service scenario, the postOrders endpoint remains unaffected by the chaos engineering action. This is due to the tool deploying a fallback mechanism capable of handling the error and returning a response to the client. This mechanism was applied for both 400 and 429 errors.
The endpoints postProducts, getProduct, and getProducts experienced downtime, as

the entire product service was unavailable. As previously mentioned, the tool is unable to handle service unavailability in this context. During the execution of the wrong_-url scenario, the postOrders endpoint fails as expected due to its inability to retrieve data. However, after some time, it became operational again because the tool deployed a fallback mechanism to handle the error and provide a response. Once again, the tool's solution in this case was the fallback mechanism.
A total of 43,221 requests were sent, of which 2,460 resulted in errors, yielding an error rate of 5.69%. This demonstrates that the tool effectively mitigated 400 and 429 errors, reducing the error rate by 57.18%.
The response time statistics are provided in table A.40.

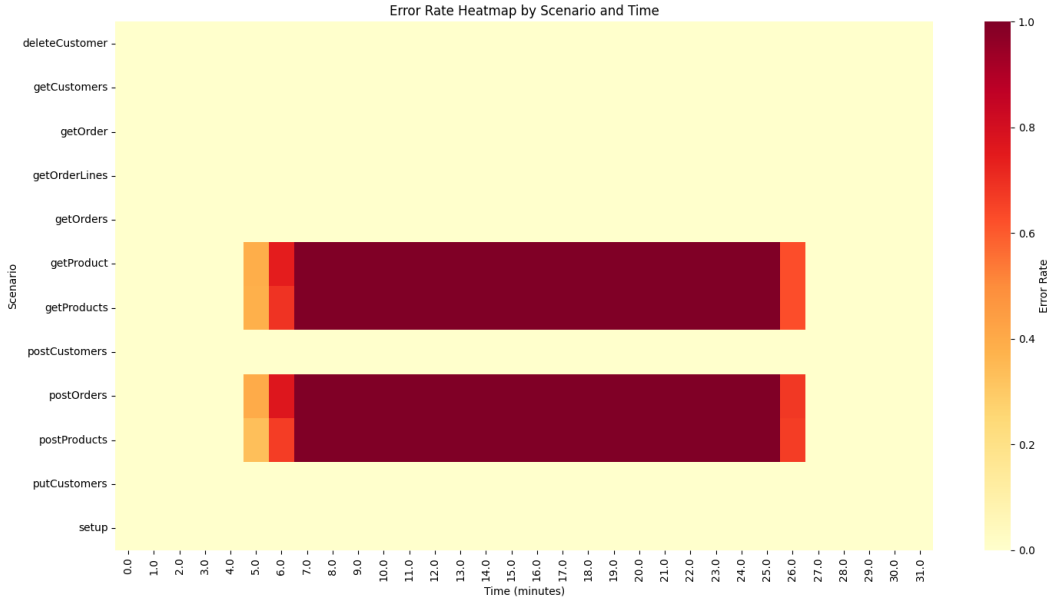

Figure 7.26: Heatmap of the Spring Down scenario with tool deployed

The figure 7.26 presents the heat map for the Spring Down scenario with the tool deployed.
A comparison with the results of the Spring Down scenario without the tool reveals that the outcomes are nearly identical. The tool was unable to categorize the error using its classification algorithm and, therefore, could not execute an appropriate solution to resolve the issue.
In this scenario, 12,316 requests were sent, of which 2,710 resulted in errors, yielding an

error rate of 22.00%. Thus, the tool was not able to reduce the error rate in this case. The response time statistics are detailed in table A.41.

The figure A.15 presents the heat map for the Spring Configuration scenario with the tool deployed.
As in the previously described scenario, the results are nearly identical to those observed in the absence of the tool. The tool was unable to categorize the error using its classification algorithm or execute a corrective solution.
In this scenario, 16,981 requests were sent, of which 4,877 resulted in errors, yielding an error rate of 28.72%. Once again, the tool was not able to reduce the error rate.
The response time statistics are presented in table A.42.

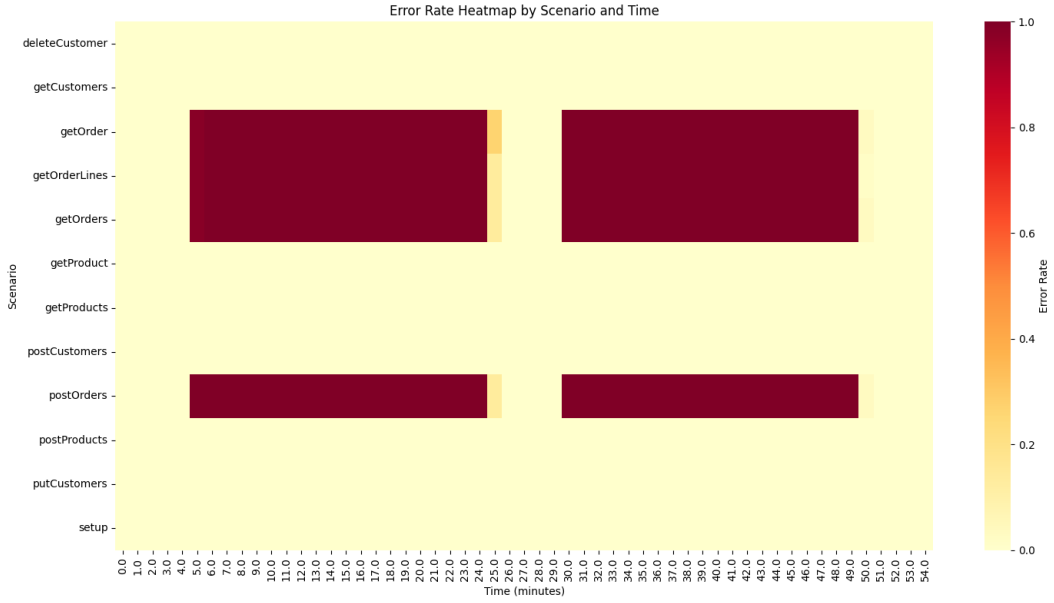

Figure 7.27: Heatmap of the Spring Database Connection scenario with tool deployed

The figure 7.27 illustrates the heat map for the Spring Database Connection scenario with the tool deployed.
Similar to the Spring Database Connection scenarios in the SMS system, the tool was unable to resolve the errors. Once again, the tool failed to categorize the error using the classification algorithm and could not execute a corrective solution.
A total of 19,052 requests were executed, with 5,269 errors, resulting in a failure rate of

27.66%. The tool was ineffective in reducing the error rate.
The response time statistics are presented in table A.43.

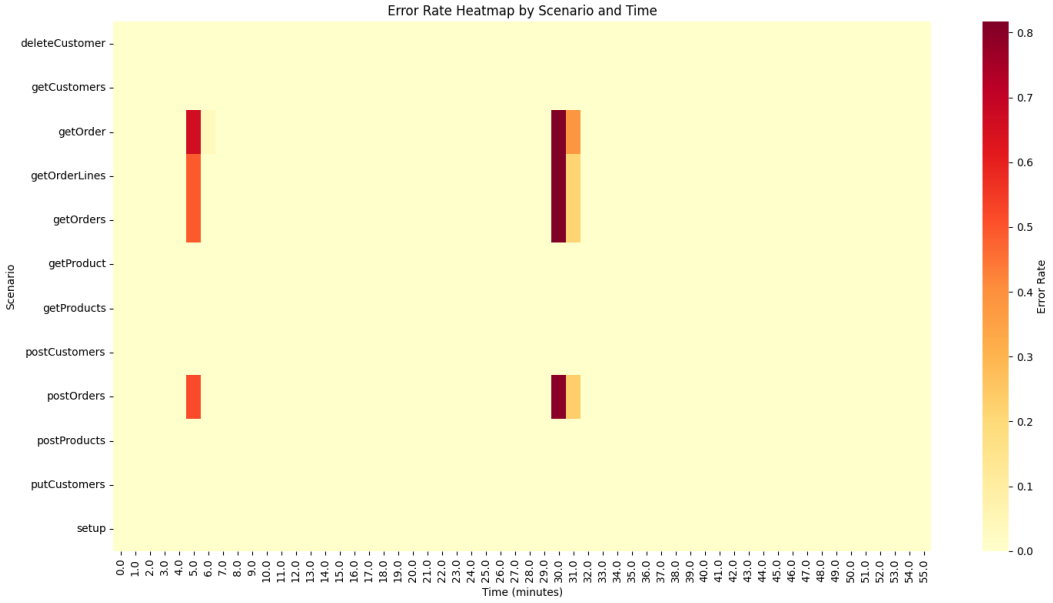

Figure 7.28: Heatmap of the Kubernetes Pod Unhealthy scenario with tool deployed

The figure 7.28 illustrates that all request errors were successfully resolved.
As in the previous Kubernetes Pod Unhealthy scenario, the tool detected high CPU and memory usage in the pod and applied the CPU and memory limits solution. This approach increased the allocated CPU and memory while also restoring the health endpoint, thereby making the service available again. As a result, the order pod was able to restart and respond to incoming requests.
During the scenario, 22,343 requests were sent, of which 341 resulted in errors, yielding an error rate of 1.53%. Compared to the scenario without the tool, this corresponds to a 95.13% reduction in the error rate.
The response time statistics are presented in table A.44.

The figure A.16 presents the Service Down Kubernetes Error scenario with the tool deployed.
Similar to the Kubernetes Pod Unhealthy scenario, the tool detected excessive CPU and memory usage and applied the cpu memory limits and health solution to resolve the issue.

During the scenario, 22,353 requests were sent, of which 272 resulted in errors, yielding an error rate of 1.22%. Compared to the scenario without the tool, the error rate was reduced by 96.16%.
The response time statistics are presented in table A.45.

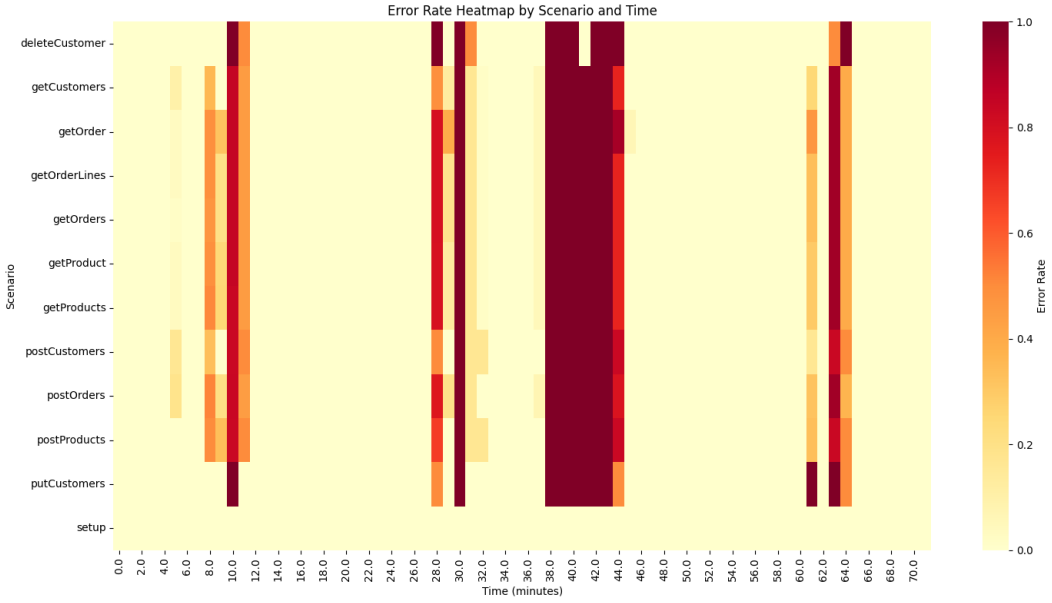

Figure 7.29: Heatmap of the Kubernetes Node Problem scenario with tool deployed

The figure 7.29 illustrates the Kubernetes Node Problem scenario with the tool deployed. The results indicate that the tool was able to resolve both chaos engineering actions. The outcome is identical to that observed in the Kubernetes Node Problem scenario within the SMS use case. In the first phase, the tool detected low disk space and applied the free disk space solution to reclaim storage. In the second phase, the tool identified insufficient memory and CPU resources and allocated additional resources to the node.
A total of 27,371 requests were executed, of which 3,849 resulted in errors, yielding an error rate of 14.06%. This demonstrates that the tool was able to reduce the error rate by 78.75%.
The response time statistics are presented in table A.46.

Figure 7.30: Heatmap of the Kubernetes Low CPU Memory scenario with tool deployed

The figure 7.30 presents the Kubernetes Low CPU Memory scenario with the tool deployed.
The results indicate that all errors were successfully resolved by the tool. The tool detected the Kubernetes event and executed the memory, cpu limits solution, increasing the available resources for the pod.
A total of 12,221 requests were processed, with 275 errors, resulting in a failure rate of 2.25%. This indicates that the tool reduced the error rate by 91.97%.
The response time statistics are presented in table A.47.

The figure A.17 illustrates the Kubernetes Invalid Image scenario with the tool deployed. The tool detected the Kubernetes event and executed the verify image solution, which checks the image tag and updates it to latest if necessary. This solution successfully resolved all errors in the scenario.
A total of 12,175 requests were processed, with 77 errors, resulting in a failure rate of 0.63%. This corresponds to a 97.56% reduction in the error rate.
The response time statistics are presented in table A.48.

The figure A.18 presents the Kubernetes Configuration scenario with the tool deployed. Similar to the SMS scenario, the tool detected the Kubernetes event and applied the

Kubernetes YAML formatting solution, which reformatted the YAML file to the correct structure. After the execution of this solution, the pod was successfully created and able to respond to all endpoints, including postOrders.
A total of 12,350 requests were executed, with 95 errors, yielding a failure rate of 0.77%. This indicates that the tool was able to reduce the error rate by 96.11%.
The response time statistics are presented in table A.49.

7.3 Discussion and Findings

System	Tool	Scenario	Total Requests	Request Errors	Failure Rate (%)	Error Reduction (%)	Solution Used
SMS	No	Spring Timeout	23,362	8,603	36.82	-	-
SMS	No	Spring Third-Party Service	21,899	4,857	22.18	-	-
SMS	No	Spring Request	45,617	8,650	18.96	-	-
SMS	No	Spring Down	12,947	4,662	36.01	-	-
SMS	No	Spring Configuration	15,891	6,560	41.28	-	-
SMS	No	Spring Database Connection	22,483	9,006	40.06	-	-
SMS	No	Kubernetes Pod Unhealthy	22,732	9,386	41.29	-	-
SMS	No	Service Down Kubernetes Error	22,691	9,340	41.16	-	-
SMS	No	Kubernetes Node Problem	27,718	19,236	69.40	-	-
SMS	No	Kubernetes Low CPU Memory	12,488	4,716	37.76	-	-
SMS	No	Kubernetes Invalid Image	12,352	4,581	37.09	-	-

System	Tool	Scenario	Total Requests	Request Errors	Failure Rate (%)	Error Reduction (%)	Solution Used
SMS	No	Kubernetes Configuration	12,575	4,659	37.05	-	-
SMS	Yes	Spring Timeout	24,019	7,596	31.62	14.12	Fallback Mechanism
SMS	Yes	Spring Third-Party Service	21,818	2,602	11.93	46.21	Fallback Mechanism
SMS	Yes	Spring Request	38,849	4,033	10.38	45.25	Fallback Mechanism
SMS	Yes	Spring Down	12,831	4,683	36.50	0.00	None
SMS	Yes	Spring Configuration	15,559	6,232	40.05	0.00	None
SMS	Yes	Spring Database Connection	22,459	8,992	40.04	0.00	None
SMS	Yes	Kubernetes Pod Unhealthy	23,226	228	0.98	97.62	CPU Memory Limits & Health
SMS	Yes	Service Down Kubernetes Error	23,223	229	0.99	97.59	CPU Memory Limits & Health
SMS	Yes	Kubernetes Node Problem	29,755	4,761	16.00	76.94	Free Disk Space, Scale Node CPU & Memory
SMS	Yes	Kubernetes Low CPU Memory	12,757	257	2.01	94.67	Memory, CPU Limits
SMS	Yes	Kubernetes Invalid Image	12,614	50	0.40	98.92	Verify Image

System	Tool	Scenario	Total Requests	Request Errors	Failure Rate (%)	Error Reduction (%)	Solution Used
SMS	Yes	Kubernetes Configuration	12,860	135	1.05	97.16	Kubernetes YAML Formatting
EOMS	No	Spring Timeout	20,153	4,778	23.71	-	-
EOMS	No	Spring Third-Party Service	20,627	2,488	12.06	-	-
EOMS	No	Spring Request	41,025	5,451	13.29	-	-
EOMS	No	Spring Down	11,949	2,491	20.85	-	-
EOMS	No	Spring Configuration	14,347	3,319	23.13	-	-
EOMS	No	Spring Database Connection	19,111	5,323	27.85	-	-
EOMS	No	Kubernetes Pod Unhealthy	19,891	6,251	31.43	-	-
EOMS	No	Service Down Kubernetes	20,022	6,378	31.85	-	-
EOMS	No	Kubernetes Node Problem	24,097	15,946	66.17	-	-
EOMS	No	Kubernetes Low CPU and Memory	10,926	3,061	28.02	-	-
EOMS	No	Kubernetes Invalid Image	10,623	2,752	25.91	-	-

System	Tool	Scenario	Total Requests	Request Errors	Failure Rate (%)	Error Reduction (%)	Solution Used
EOMS	No	Kubernetes Configuration	11,373	2,254	19.82	-	-
EOMS	Yes	Spring Timeout	21,735	3,567	16.41	30.78	Fallback Mechanism
EOMS	Yes	Spring Third-Party Service	20,627	1,460	7.08	41.29	Fallback Mechanism
EOMS	Yes	Spring Request	43,221	2,460	5.69	57.18	Fallback Mechanism
EOMS	Yes	Spring Down	12,316	2,710	22.00	0.00	None
EOMS	Yes	Spring Configuration	16,981	4,877	28.72	0.00	None
EOMS	Yes	Spring Database Connection	19,052	5,269	27.66	0.00	None
EOMS	Yes	Kubernetes Pod Unhealthy	22,343	341	1.53	95.13	CPU Memory Limits & Health
EOMS	Yes	Service Down Kubernetes	22,353	272	1.22	96.16	CPU Memory Limits & Health
EOMS	Yes	Kubernetes Node Problem	27,371	3,849	14.06	78.75	Free Disk Space, Scale Node CPU & Memory
EOMS	Yes	Kubernetes Low CPU and Memory	12,221	275	2.25	91.97	Memory, CPU Limits
EOMS	Yes	Kubernetes Invalid Image	12,175	77	0.63	97.56	Verify Image

System	Tool	Scenario	Total Requests	Request Errors	Failure Rate (%)	Error Reduction (%)	Solution Used
EOMS	Yes	Kubernetes Configuration	12,350	95	0.77	96.11	Kubernetes YAML Formatting

Table 7.2: Summary of all results with details of the system used, the tool, the scenario, the total requests, the errors, the error rate, the error reduction, and the solution used

The table 7.2 summarizes the results of the experiments conducted in the SMS and EOMS systems, both with and without the tool deployed. Regarding Spring failures, the tool had a limited impact. Specifically, the Spring Down, Spring Configuration, and Spring Database Connection failure scenarios exhibited negligible changes in error rates, indicating that these issues were beyond the tool's current mitigation capabilities. As previously discussed, the primary challenge was the classification algorithm's inability to correctly categorize these errors. However, the fallback mechanism demonstrated effectiveness in mitigating failures in the Spring Timeout, Third-Party Service, and Request scenarios, reducing error rates by 14% to 57%. Other mechanisms within the resilience library were not triggered, as the fallback mechanism was sufficient to resolve the failures in all cases. Furthermore, the additional mechanisms were not well-suited to addressing failures in these scenarios, as they require a service to exhibit intermittent availability rather than prolonged downtime or persistent error responses. This explains why the fallback mechanism proved to be the most effective solution in these cases. The tool exhibited significant effectiveness in mitigating Kubernetes-related failures, demonstrating an error rate reduction of over 95% in scenarios such as Pod Unhealthy, Service Down, Low CPU/Memory, Invalid Image, and Configuration issues. The most impactful solutions in these cases were CPU and memory limit adjustments, health check configurations, and Kubernetes YAML corrections. The effectiveness of these solutions can be attributed to Kubernetes' built-in API, which provides robust functions for handling such failures. The tool leveraged these existing capabilities to resolve the issues efficiently. In contrast, Spring-related failures relied more on the resilience library deployed alongside the service, which limited the tool's effectiveness in those scenarios. For the Kubernetes Node Problem scenario, the tool successfully reduced the error rate by approximately 77%. This result is particularly notable because the tool was not deployed on the same

node where the failure occurred. Instead, it was hosted on a separate node (in this case, locally on the laptop), allowing it to execute the necessary recovery actions remotely. Had the tool been deployed on the same failing node, it would have become inoperative once the node failed, preventing it from executing the required mitigation steps.

During the execution of the SMS scenario, a total of 252,755 requests were processed without the tool, compared to 249,970 requests with the tool deployed. For the EOMS scenarios, 206,006 requests were executed without the tool, while 242,745 requests were processed with the tool. Figure 7.31 illustrates the impact of the tool on error reduction across these requests.

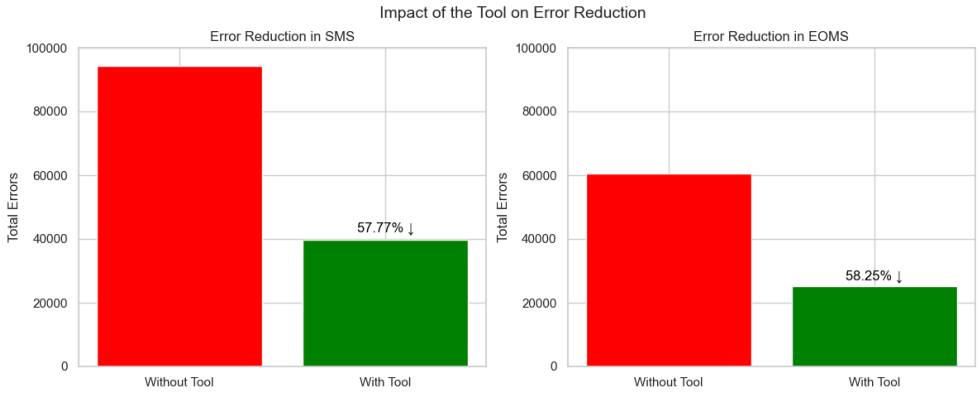

Figure 7.31: Impact of the tool on error reduction

For the SMS system, the tool achieved a total error rate reduction of 57.77% across all scenarios. In the EOMS system, the error rate reduction was comparable at 58.25%. Across all test cases, the tool achieved an overall error reduction of 57.96%, underscoring its effectiveness in mitigating failures. While the SMS system consistently exhibited a higher absolute number of errors, both systems experienced similar relative improvements when the tool was applied. This supports the hypothesis that the tool's effectiveness is independent of the system itself and is primarily influenced by the nature of the failure scenario. The higher error count in the SMS system can be attributed to the fact that the EOMS system has a greater number of endpoints, only a subset of which were targeted by the chaos engineering actions. Since both systems were subjected to identical request rates, the load was distributed across a larger number of endpoints in the EOMS system, many of which remained unaffected by the chaos engineering actions. The analysis highlights the tool's strong performance in mitigating failures related to Kubernetes resource

allocation and configuration management. While the tool also demonstrated effectiveness in addressing failures within the Spring framework, its success rate was lower in those cases.

The experimental results provide key insights into how the integration of the MAPE-K reference model, XGBoost-driven error prediction, and chaos engineering can facilitate the development of self-healing systems, ultimately enhancing the reliability and resilience of microservice architectures. These findings directly answer the research question of this thesis. The MAPE-K model forms the foundation of the tool's self-healing capabilities, enabling it to monitor failures, analyze patterns, and execute mitigation strategies through Kubernetes' recovery mechanisms and a resilience library. This approach proved highly effective, significantly reducing error rates and demonstrating the impact of automated adaptation. Accurate error classification is a crucial component of self-healing systems. The difficulty in mitigating Spring-related failures was primarily due to misclassification, raising the question of whether XGBoost could be enhanced by incorporating additional classification methods to improve accuracy. Nevertheless, the classification mechanism was successful in approximately 75% of scenarios, yielding strong overall results. Chaos engineering played a pivotal role in validating the tool's effectiveness by introducing controlled failure scenarios. The tool successfully mitigated both Kubernetes and Spring-related failures, confirming chaos engineering as a valuable framework for testing self-healing mechanisms. With an overall error reduction of 57.96%, the integration of MAPE-K, XGBoost-driven error prediction, and chaos engineering with load testing enhances microservice resilience. The tool's consistent performance across SMS and EOMS systems further demonstrates its adaptability. However, improving error classification and expanding the strategies within the Spring Boot resilience library will be essential in evolving the tool into a fully autonomous self-healing system.

8 Conclusion

This thesis presented the development and evaluation of a tool that integrates chaos engineering, the MAPE-K model, and machine learning to enhance the resilience of microservice systems. The tool was tested on two systems, SMS and EOMS, using chaos engineering and load testing to simulate failure scenarios. Its effectiveness was assessed by measuring error rates across twelve scenarios, both with and without the tool deployed.

The experimental results demonstrate the tool's strong impact on error reduction, particularly in Kubernetes-related failures, where it achieved an error rate reduction of over 95%. Effective mitigation strategies included fallback mechanisms and Kubernetes configuration adjustments. However, the tool showed limitations in handling Spring-related failures due to misclassification within the error classification model. While the fallback mechanism was effective in certain cases, other resilience strategies were not triggered or were less suited to persistent failures. Despite these limitations, the tool achieved an overall error rate reduction of 57.96%, indicating its potential as a self-healing solution for microservices.

By integrating the MAPE-K model for adaptive decision-making, XGBoost for predictive error classification, and chaos engineering for resilience validation, the tool enhances the reliability of microservice architectures. This combination enables automated failure mitigation, reducing system downtime and improving overall service resilience. The findings confirm the practicality of this approach and provide a foundation for future research aimed at refining classification accuracy and expanding resilience strategies to create a fully autonomous self-healing system.

8.1 Outlook

The findings of this thesis give rise to several questions for further investigation. While the experimental results demonstrate promising outcomes, certain scenarios remain in

which the tool does not perform as expected. Moreover, the current architecture of the tool is not yet optimized. As outlined in the Implementation chapter, the Executor component is responsible for both planning and execution within the MAPE-K loop. To more accurately adhere to the MAPE-K architecture, this component should be divided into separate planning and execution components. Future research should explore alternative decision algorithms, assess their scalability, and determine how additional classes can be seamlessly integrated into the tool. Additionally, there is potential to enhance the accuracy of the classification algorithm. Further investigations should explore the combination of the XGBoost classification algorithm with other techniques to improve classification precision. The Spring Boot library used in this work should also be further investigated to identify additional patterns and algorithms that could improve the function of the tool. The chaos engineering component of the tool should also be expanded to encompass a broader range of scenarios, creating a more realistic testing environment. More generally, this thesis represents an initial step in integrating chaos engineering, MAPE-K, and machine learning to develop a tool aimed at improving the resilience of microservices. While the implemented tool is currently in a prototypical stage, the results indicate its potential effectiveness. Therefore, further research is necessary to refine the tool and enhance its accuracy.

Bibliography

[1] Ade, M., and Sheriffdeen, K. Redundancy and failover mechanisms in microservices. ResearchGate, Sept. 2019. Retrieved from ResearchGate.

[2] Afzal, S., and Kavitha, G. Load balancing in cloud computing – A hierarchical taxonomical classification. *Journal of Cloud Computing 8*, 1 (Dec. 2019), 22.

[3] Ali, B. ali-bouali/microservices-full-code. https://github.com/ali-bouali/microservices-full-code, 2023. Accessed: 2025-02-27.

[4] Ali, B. ali-bouali/springboot-3-micro-service-demo. https://github.com/ali-bouali/springboot-3-micro-service-demo, 2023. Accessed: 2025-02-27.

[5] Amazon Web Services. Different ways to implement your test - AWS Graviton Performance Testing: Tips for Independent Software Vendors. https://docs.aws.amazon.com/whitepapers/latest/aws-graviton-performance-testing/different-ways-to-implement-your-test.html, 2024. Accessed: 2025-02-19.

[6] Amazon Web Services. PERF05-BP04 Load test your workload - Performance Efficiency Pillar. https://docs.aws.amazon.com/wellarchitected/latest/performance-efficiency-pillar/perf_process_culture_load_test.html, 2024. Accessed: 2025-02-19.

[7] Amazon Web Services. REL12-BP03 Test scalability and performance requirements - Reliability Pillar. https://docs.aws.amazon.com/wellarchitected/latest/reliability-pillar/rel_testing_resiliency_test_non_functional.html, 2024. Accessed: 2025-02-19.

[8] Amazon Web Services. Testing stages in continuous integration and continuous delivery - Practicing Continuous Integration and Continuous Delivery on AWS. https://docs.aws.amazon.com/whitepapers/latest/practicing-continuous-integration-continuous-delivery/testing-stages-i

n-continuous-integration-and-continuous-delivery.html, 2024. Accessed: 2025-02-19.

[9] Bassam, I. Building Resilient Distributed Systems: 8 Strategies for Success. https://www.axelerant.com/blog/how-to-build-resilient-distributed-systems, 2024. Accessed: 2025-02-09.

[10] Beyer, B., Jones, C., Petoff, J., and Murphy, N. R., Eds. *Site reliability engineering: how Google runs production systems*, first edition ed. O'Reilly, Beijing Boston Farnham Sebastopol Tokyo, 2016.

[11] Boyapati, S. R., and Szabo, C. Self-adaptation in Microservice Architectures: A Case Study. In *2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)* (Hiroshima, Japan, Mar. 2022), IEEE, pp. 42–51.

[12] Bucchiarone, A., Guidi, C., Lanese, I., Bencomo, N., and Spillner, J. A MAPE-K Approach to Autonomic Microservices. In *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)* (Honolulu, HI, USA, Mar. 2022), IEEE, pp. 100–103.

[13] Casalicchio, E. Container Orchestration: A Survey. In *Systems Modeling: Methodologies and Tools*, A. Puliafito and K. S. Trivedi, Eds. Springer International Publishing, Cham, 2019, pp. 221–235. Series Title: EAI/Springer Innovations in Communication and Computing.

[14] Chaos Community. PRINCIPLES OF CHAOS ENGINEERING - Principles of chaos engineering. https://principlesofchaos.org/, 2019. Accessed: 2025-02-19.

[15] Chen, T., and Guestrin, C. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Aug. 2016), pp. 785–794. arXiv:1603.02754 [cs].

[16] Computing, A., et al. An architectural blueprint for autonomic computing. *IBM White Paper 31*, 2006 (2006), 1–6.

[17] D'Orazio, N., and Reiners, J. Load testing applications - AWS Prescriptive Guidance. https://docs.aws.amazon.com/prescriptive-guidance/latest/load-testing/welcome.html, 2024. Accessed: 2025-02-19.

[18] Filho, M., Pimentel, E., Pereira, W., Maia, P. H. M., and Cortes, M. I. Self-Adaptive Microservice-based Systems - Landscape and Research Opportunities. In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (Madrid, Spain, May 2021), IEEE, pp. 167–178.

[19] Ganesh Raja, M., and Jeyalaksshmi, S. Self-Configuration and Self-Healing Framework Using Extreme Gradient Boosting (XGBoost) Classifier for IoT-WSN. *Journal of Interconnection Networks 24*, 03 (Sept. 2024), 2350022.

[20] Gorla, A., Pezze, M., Wuttke, J., Mariani, L., and Pastore, F. Achieving cost-effective software reliability through self-healing. *Computing and Informatics 29*, 1 (2010), 93–115.

[21] Grafana Labs. Scenarios | Grafana k6 documentation. https://grafana.com/docs/k6/latest/using-k6/scenarios/, 2024. Accessed: February 27, 2025.

[22] Gustafsson, A. S., and Wirehed, A. Data-Driven Fault Categorization of Multimodal Logs using Natural Language Processing Techniques. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, 2021.

[23] Kang, Y., Ozdogan, M., Zhu, X., Ye, Z., Hain, C., and Anderson, M. Comparative assessment of environmental variables and machine learning algorithms for maize yield prediction in the US Midwest. *Environmental Research Letters 15*, 6 (June 2020), 064005.

[24] Karol Santos Nunes, J. P., Nejati, S., Sabetzadeh, M., and Nakagawa, E. Y. Self-adaptive, Requirements-driven Autoscaling of Microservices. In *Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (Lisbon AA Portugal, Apr. 2024), ACM, pp. 168–174.

[25] Kephart, J., and Chess, D. The vision of autonomic computing. *Computer 36*, 1 (Jan. 2003), 41–50.

[26] Kubernetes Documentation Team. Kubernetes overview. https://kubernetes.io/docs/concepts/overview/, 2024. Accessed: 2025-02-19.

[27] Kwasny, S. C., and Faisal, K. A. Overcoming Limitations of Rule-Based Systems: An Example of a Hybrid Deterministic Parser. In *Konnektionismus in Artificial Intelligence und Kognitionsforschung*, W. Brauer and G. Dorffner, Eds.,

vol. 252. Springer Berlin Heidelberg, Berlin, Heidelberg, 1990, pp. 48–57. Series Title: Informatik-Fachberichte.

[28] Lewis, J., and Fowler, M. Microservices. https://martinfowler.com/articles/microservices.html, 2014. Accessed: 2025-02-09.

[29] Magableh, B., and Almiani, M. A Self Healing Microservices Architecture: A Case Study in Docker Swarm Cluster. In *Advanced Information Networking and Applications*, L. Barolli, M. Takizawa, F. Xhafa, and T. Enokido, Eds., vol. 926. Springer International Publishing, Cham, 2020, pp. 846–858. Series Title: Advances in Intelligent Systems and Computing.

[30] Malik, S., Naqvi, M. A., and Moonen, L. CHESS: A Framework for Evaluation of Self-Adaptive Systems Based on Chaos Engineering. In *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (Melbourne, Australia, May 2023), IEEE, pp. 195–201.

[31] Massé, M. H., and Massé, M. *REST API design rulebook: designing consistent RESTful Web Service Interfaces.* O'Reilly, Beijing Köln, 2012.

[32] Mendonca, N. C., Aderaldo, C. M., Camara, J., and Garlan, D. Model-Based Analysis of Microservice Resiliency Patterns. In *2020 IEEE International Conference on Software Architecture (ICSA)* (Salvador, Brazil, Mar. 2020), IEEE, pp. 114–124.

[33] Mozilla Contributors. 504 Gateway Timeout - HTTP | MDN. https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status/504, 2025. Accessed: 2025-02-09.

[34] Nassu, B. T., and Nanya, T. Interaction Faults Caused by Third-Party External Systems — A Case Study and Challenges. In *Service Availability*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, T. Nanya, F. Maruyama, A. Pataricza, and M. Malek, Eds., vol. 5017. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 59–74. Series Title: Lecture Notes in Computer Science.

[35] Newman, S. *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2021.

[36] Nguyen, Q. *Mastering concurrency in python: create faster programs using concurrency, asynchronous, multithreading, and parallel programming*, 1st ed ed. Packt Publishing, Birmingham, 2018.

[37] Rodemer, D. domzilla/Caffeine. https://github.com/domzilla/Caffeine, 2022. Accessed: February 27, 2025.

[38] Rutten, E., Marchand, N., and Simon, D. Feedback Control as MAPE-K Loop in Autonomic Computing. In *Software Engineering for Self-Adaptive Systems III. Assurances*, R. De Lemos, D. Garlan, C. Ghezzi, and H. Giese, Eds., vol. 9640. Springer International Publishing, Cham, 2017, pp. 349–373. Series Title: Lecture Notes in Computer Science.

[39] Shekhar, G. Microservices Design Patterns for Cloud Architecture. *International Journal of Computer Science and Engineering 11*, 9 (Sept. 2024), 1–7.

[40] Skoric, A. Classifier Development for Log Analysis in Spring Boot and Kubernetes, 2024.

[41] Ve, P., Sai, M., Vuppalapati, V. S. M., Modi, S., and Ponnusamy, S. Exponential backoff: A comprehensive approach to handling failures in distributedarchitectures. ResearchGate, 2024. Retrieved from ResearchGate.

[42] Wong, T., and Barahona, M. Deep incremental learning models for financial temporal tabular datasets with distribution shifts, Oct. 2023. arXiv:2303.07925 [cs].

[43] xChange Solutions. Container xChange | Containers on Demand in 2500+ Locations. https://www.container-xchange.com/, 2025. Accessed: 2025-02-24.

[44] XGBoost Developers. Categorical Data — xgboost 3.0.0 documentation. https://xgboost.readthedocs.io/en/stable/tutorials/categorical.html, 2022. Accessed: 2025-02-27.

[45] Xu, T., Zhang, J., Huang, P., Zheng, J., Sheng, T., Yuan, D., Zhou, Y., and Pasupathy, S. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton Pennsylvania, Nov. 2013), ACM, pp. 244–259.

[46] Yang, T., Lee, C., Shen, J., Su, Y., Feng, C., Yang, Y., and Lyu, M. R. MicroRes: Versatile Resilience Profiling in Microservices via Degradation Dissemination Indexing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna Austria, Sept. 2024), ACM, pp. 325–337.

[47] Yazdanparast, Z. A Survey on Self-healing Software System, Mar. 2024. arXiv:2403.00455 [cs].

[48] Özsu, M. T., and Valduriez, P. *Principles of Distributed Database Systems*. Springer International Publishing, Cham, 2020.

A Appendix

A.1 Tools Used

Table A.1 presents the tools and resources utilized throughout the development of this thesis. Not all tools are included in the list, as many have already been discussed in relevant sections of the thesis. The table specifically highlights tools that were used in the background and not explicitly mentioned elsewhere in the document.

Table A.1: Tools and Resources Used

Tool	Usage
LaTeX	Typesetting system used for formatting and structuring this document.
ChatGPT	AI assistant used for idea generation, literature exploration, and text refinement.
Claude	AI model utilized for code debugging and issue resolution.
DeepL	Translation tool employed for multilingual text conversion.
DeepL Write	AI-powered writing assistant used for grammar and style enhancement.
IntelliJ IDEA	Integrated development environment (IDE) used for software development and code management.

Scenario	Mean	Min	Max	Std
getSchools	1834.40	0.29	5036.91	2345.20
getSchoolsWithStudents	2364.42	0.04	5043.13	2483.83
getStudents	957.55	3.23	5041.91	1936.38
postSchools	1719.97	0.10	5035.45	2327.60
postStudents	1239.04	2.87	5029.37	2131.29

Table A.2: Response time statistics (milliseconds) of the Spring Timeout scenario

Scenario	Mean	Min	Max	Std
getSchools	21.99	5.78	1394.66	26.05
getSchoolsWithStudents	22.68	4.00	1206.25	29.21
getStudents	1115.33	3.45	5030.45	2056.49
postSchools	20.98	4.20	1667.62	34.93
postStudents	853.43	3.91	5026.44	1851.68

Table A.3: Response time statistics (milliseconds) of the Spring Third-Party Service scenario

Scenario	Mean	Min	Max	Std
getSchools	219.76	3.04	5025.24	960.27
getSchoolsWithStudents	298.65	2.91	5021.56	1141.75
getStudents	496.68	3.03	5047.98	1453.94
postSchools	250.53	3.32	5023.51	1047.54
postStudents	543.53	3.56	5047.02	1526.11

Table A.4: Response time statistics (milliseconds) of the Spring Request scenario

Scenario	Mean	Min	Max	Std
getSchools	19.84	5.65	286.54	9.92
getSchoolsWithStudents	21.19	5.44	130.85	9.99
getStudents	2418.84	3.78	5030.05	2446.75
postSchools	20.64	5.09	266.85	10.33
postStudents	1981.62	3.34	5011.64	2359.17

Table A.5: Response time statistics (milliseconds) of the Spring Configuration scenario

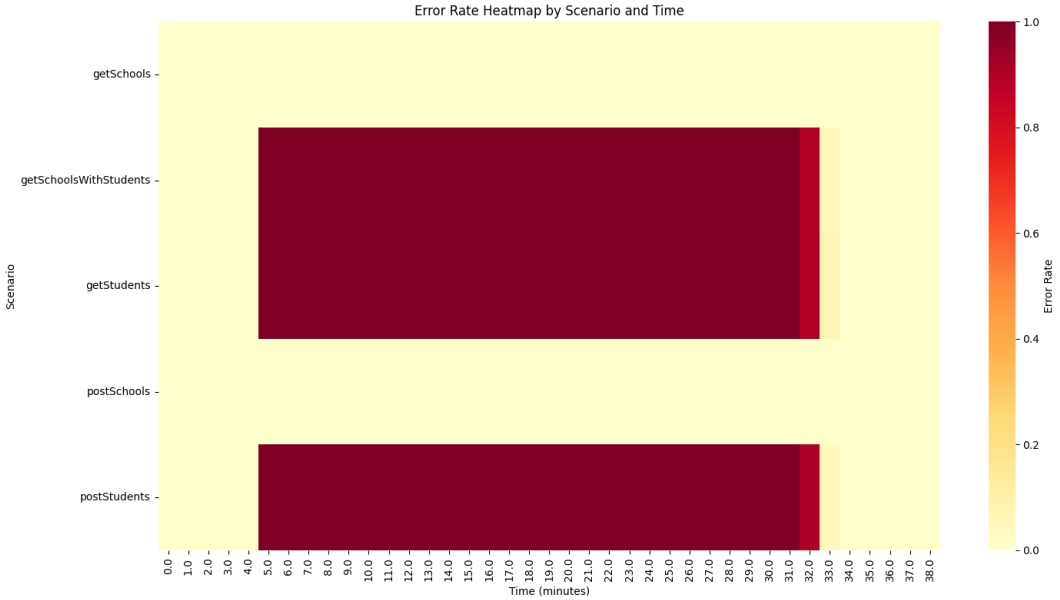

Figure A.1: Heatmap of the Spring Configuration scenario

Scenario	Mean	Min	Max	Std
getSchools	17.47	5.31	374.95	9.39
getSchoolsWithStudents	17.64	4.57	149.04	8.62
getStudents	2837.75	3.25	5019.27	2435.35
postSchools	17.52	4.72	374.30	10.59
postStudents	1871.28	2.97	5017.44	2352.32

Table A.6: Response time statistics (milliseconds) of the Spring Configuration scenario

Scenario	Mean	Min	Max	Std
getSchools	2824.11	3.17	5045.58	2453.52
getSchoolsWithStudents	3195.53	3.11	5041.15	2380.75
getStudents	21.06	5.26	334.72	11.10
postSchools	3037.16	3.66	5063.33	2422.13
postStudents	16.94	3.58	98.83	8.29

Table A.7: Response time statistics (milliseconds) of the Spring Database Connection scenario

Figure A.2: Heatmap of the Kubernetes Pod Unhealthy scenario

Scenario	Mean	Min	Max	Std
getSchools	2306.50	2.60	5055.96	2445.42
getSchoolsWithStudents	2501.58	1.91	5031.68	2457.09
getStudents	19.53	5.87	357.94	9.75
postSchools	2146.76	2.81	5031.06	2433.23
postStudents	15.05	3.10	96.69	8.16

Table A.8: Response time statistics (milliseconds) of the Kubernetes Pod Unhealthy scenario

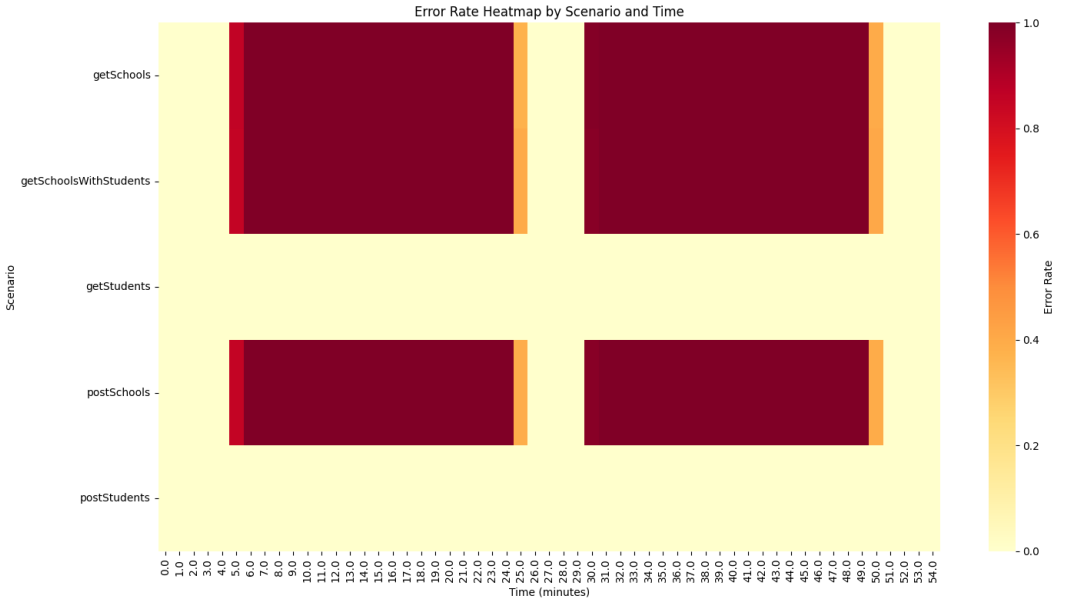

Figure A.3: Heatmap of the Service Down Kubernetes scenario

Scenario	Mean	Min	Max	Std
getSchools	2260.34	2.24	5058.51	2440.63
getSchoolsWithStudents	2593.63	2.12	5056.01	2452.95
getStudents	19.95	6.18	352.22	10.38
postSchools	2343.41	2.74	5057.49	2453.38
postStudents	14.69	3.10	100.90	8.48

Table A.9: Response time statistics (milliseconds) of the Service Down Kubernetes scenario

Scenario	Mean	Min	Max	Std
getSchools	3503.52	0.0	5015.23	2262.80
getSchoolsWithStudents	3610.16	0.0	5021.36	2211.29
getStudents	3504.59	0.0	5019.29	2262.75
postSchools	3422.78	0.0	5020.28	2294.56
postStudents	3427.67	0.0	5018.80	2292.16

Table A.10: Response time statistics (milliseconds) of the Kubernetes Node Problem scenario

Scenario	Mean	Min	Max	Std
getSchools	2176.70	3.23	5036.84	2431.60
getSchoolsWithStudents	2535.82	3.06	5035.22	2462.87
getStudents	21.05	5.99	361.98	12.31
postSchools	2201.35	3.92	5037.53	2443.89
postStudents	17.48	3.84	201.26	11.68

Table A.11: Response time statistics (milliseconds) of the Kubernetes Low CPU Memory scenario

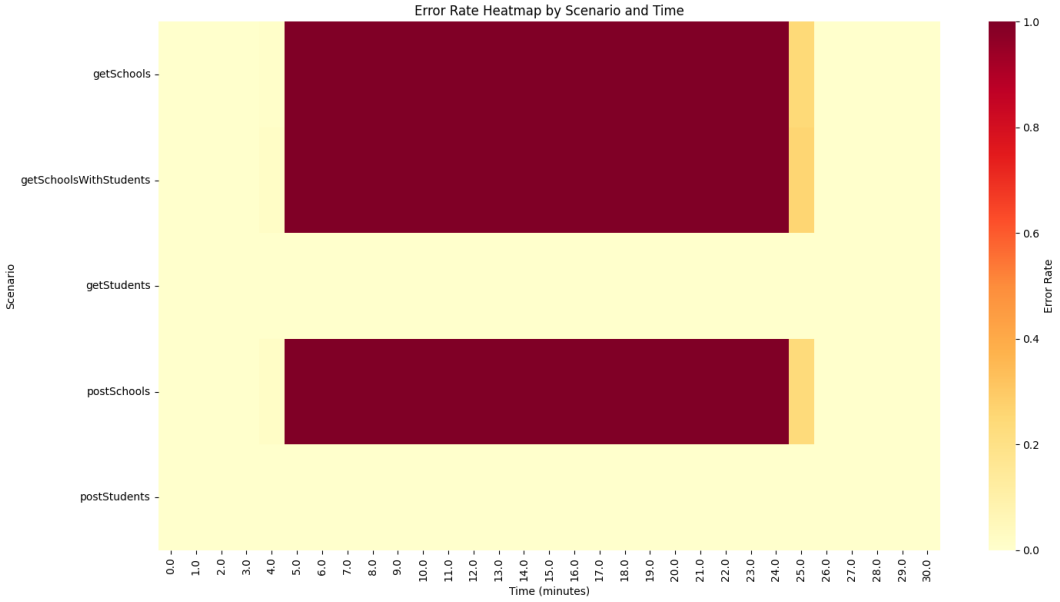

Figure A.4: Heatmap of the Kubernetes Invalid Image scenario

Scenario	Mean	Min	Max	Std
getSchools	2615.83	3.48	5037.62	2443.34
getSchoolsWithStudents	3121.40	3.72	5023.90	2396.09
getStudents	20.42	5.34	295.06	9.53
postSchools	2918.08	3.97	5025.18	2438.53
postStudents	17.72	4.13	116.58	8.68

Table A.12: Response time statistics (milliseconds) of the Kubernetes Invalid Image scenario

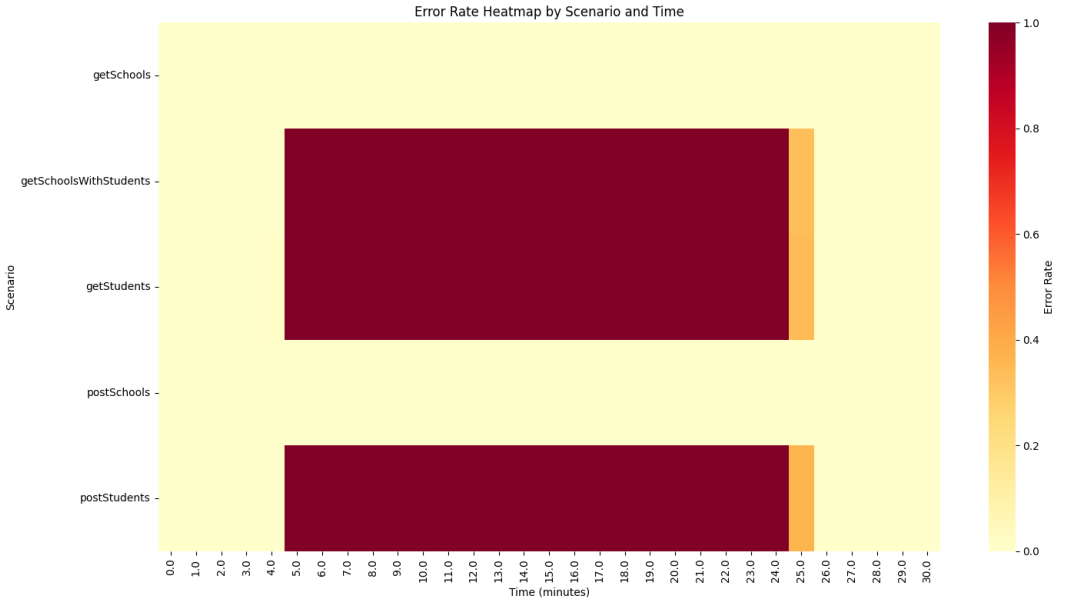

Figure A.5: Heatmap of the Kubernetes Configuration scenario

Scenario	Mean	Min	Max	Std
getSchools	18.17	5.49	361.01	10.69
getSchoolsWithStudents	17.65	5.21	149.85	9.74
getStudents	2523.63	3.81	5029.20	2468.97
postSchools	17.12	4.58	345.65	11.63
postStudents	2476.28	3.18	5024.81	2467.12

Table A.13: Response time statistics (milliseconds) of the Kubernetes Configuration scenario

Scenario	Mean	Min	Max	Std
deleteCustomer	20.25	5.45	144.00	15.08
getCustomers	18.68	4.94	150.65	7.73
getOrder	820.78	3.47	5025.13	1834.32
getOrderLines	818.81	2.94	5024.31	1831.89
getOrders	824.07	5.52	5010.43	1831.90
getProduct	1559.39	2.94	5014.75	2258.28
getProducts	1033.68	3.21	5016.73	1956.84
postCustomers	19.87	4.91	66.26	7.96
postOrders	1891.08	5.87	5011.06	2404.76
postProducts	2461.16	8.76	5012.60	2442.67
putCustomers	19.82	6.22	48.24	7.45
setup	8.61	6.27	14.08	2.31

Table A.14: Response time statistics (milliseconds) of the Spring Timeout scenario

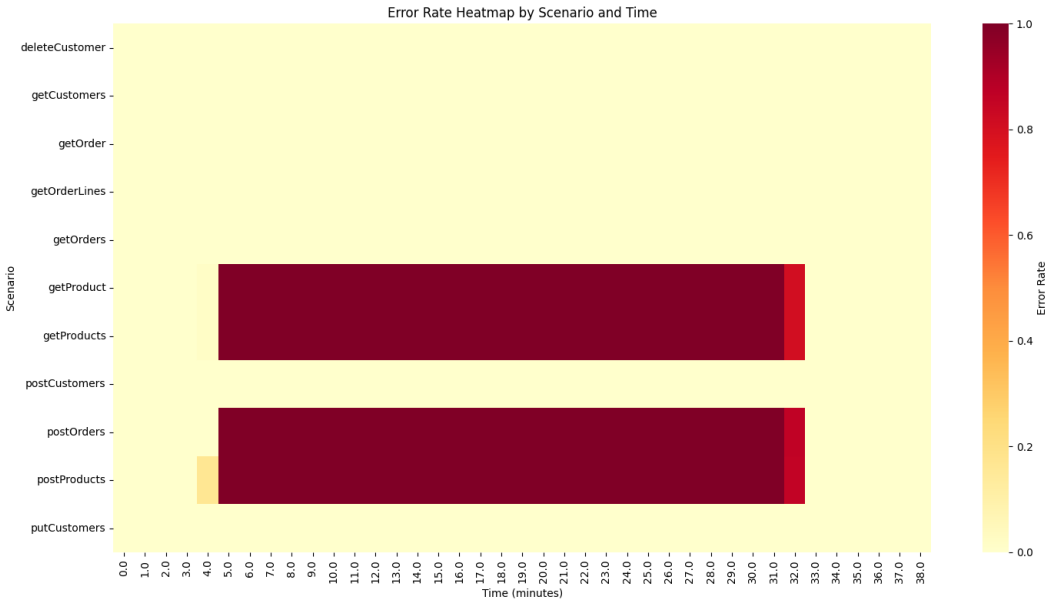

Figure A.6: Heatmap of the Spring Configuration scenario

Scenario	Mean	Min	Max	Std
deleteCustomer	17.55	5.50	45.68	8.95
getCustomers	14.43	3.88	108.12	6.82
getOrder	12.70	2.98	57.75	6.54
getOrderLines	13.06	2.98	57.65	6.68
getOrders	14.99	3.99	280.48	8.95
getProduct	770.61	3.21	5020.93	1730.92
getProducts	338.04	2.54	5006.17	1144.54
postCustomers	16.58	4.39	55.52	8.34
postOrders	26.36	5.84	559.23	19.96
postProducts	1271.73	6.15	5002.18	2099.03
putCustomers	16.30	6.95	32.12	5.76
setup	7.50	6.10	12.00	1.30

Table A.15: Response time statistics (milliseconds) of the Spring Third-Party Service scenario

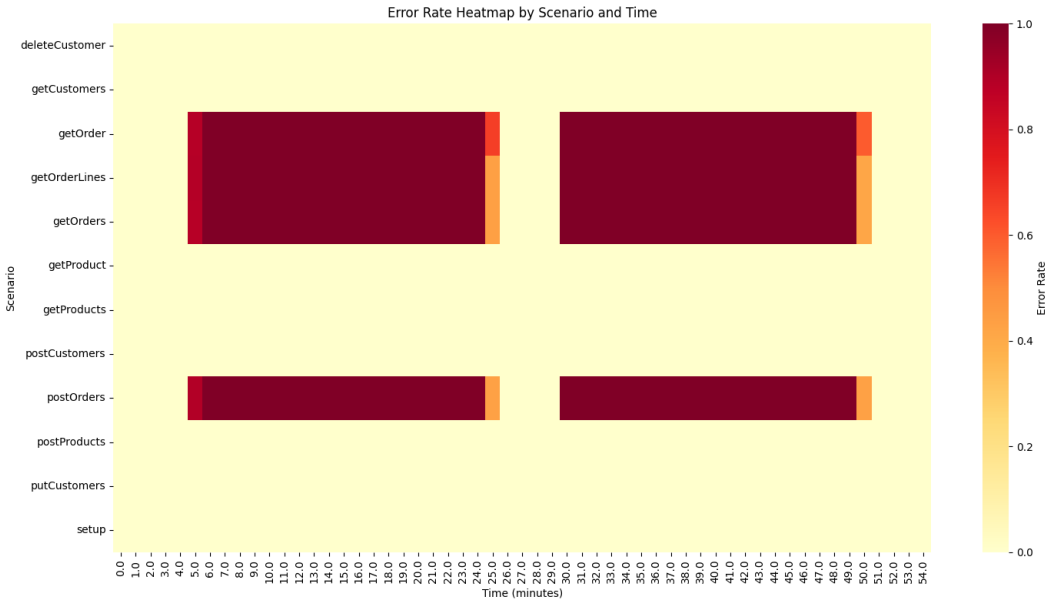

Figure A.7: Heatmap of the Kubernetes Pod Unhealthy scenario

Scenario	Mean	Min	Max	Std
deleteCustomer	19.42	6.55	72.28	8.76
getCustomers	17.54	4.57	253.07	9.20
getOrder	59.54	3.27	5002.71	469.11
getOrderLines	54.19	3.21	5002.51	434.09
getOrders	58.98	2.72	5002.78	452.56
getProduct	348.05	3.16	5028.78	1224.73
getProducts	121.97	2.96	5006.82	678.36
postCustomers	18.18	5.66	121.20	8.75
postOrders	77.00	4.39	5000.98	500.45
postProducts	706.85	8.37	5010.41	1692.35
putCustomers	11.71	5.10	40.56	6.10
setup	8.73	6.68	17.58	2.69

Table A.16: Response time statistics (milliseconds) of the Spring Request scenario

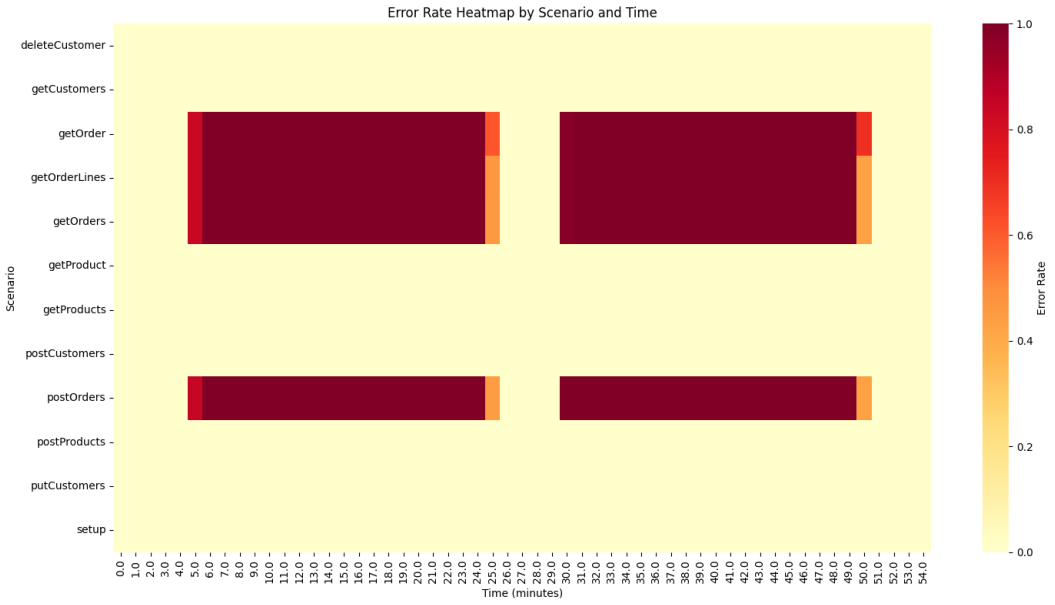

Figure A.8: Heatmap of the Service Down Kubernetes scenario

Scenario	Mean	Min	Max	Std
deleteCustomer	24.75	10.21	59.77	11.10
getCustomers	19.14	5.21	105.48	8.50
getOrder	16.48	3.85	217.91	9.12
getOrderLines	17.27	4.34	218.68	9.13
getOrders	18.31	4.42	288.70	11.23
getProduct	1043.34	3.55	5009.38	1952.45
getProducts	1503.65	3.86	5022.95	2210.65
postCustomers	22.07	7.87	70.75	8.82
postOrders	31.84	6.71	554.58	23.04
postProducts	2855.84	6.43	5004.58	2389.58
putCustomers	21.36	11.81	42.08	5.85
setup	8.89	6.22	19.03	3.37

Table A.17: Response time statistics (milliseconds) of the Spring Down scenario

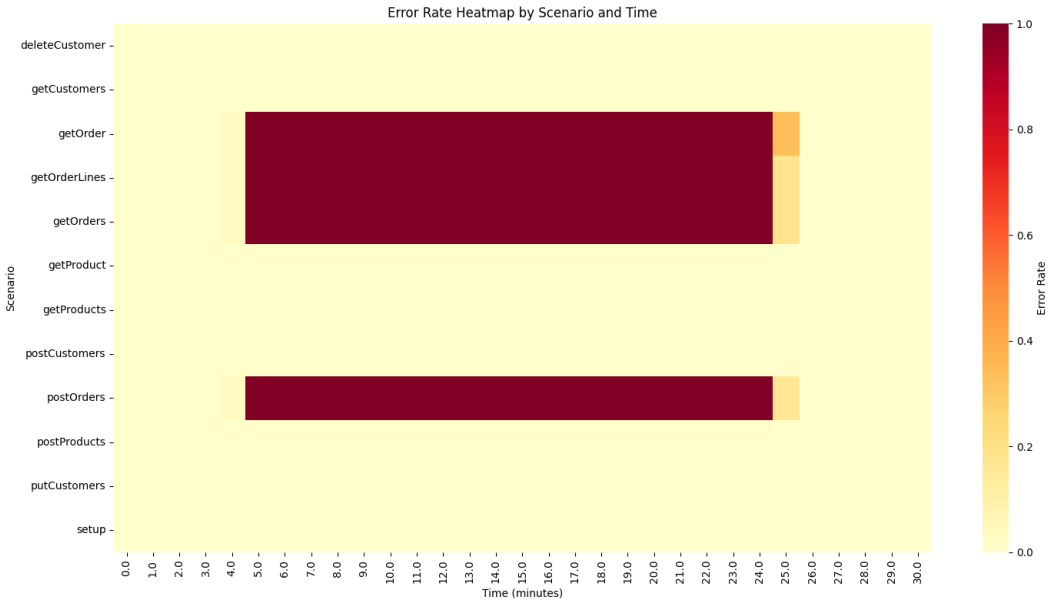

Figure A.9: Heatmap of the Kubernetes Invalid Image scenario

Scenario	Mean	Min	Max	Std
deleteCustomer	13.77	4.81	52.50	8.17
getCustomers	17.22	5.13	117.41	7.35
getOrder	14.36	3.43	119.10	7.29
getOrderLines	14.80	3.57	119.50	7.31
getOrders	15.85	3.90	233.34	8.73
getProduct	2174.42	3.04	5011.38	2310.51
getProducts	1394.31	2.94	5004.64	2069.57
postCustomers	13.50	5.01	39.43	6.12
postOrders	25.83	5.75	497.18	19.14
postProducts	2909.68	3.52	5004.85	2284.11
putCustomers	15.84	7.37	28.99	5.24
setup	8.57	6.40	14.22	2.38

Table A.18: Response time statistics (milliseconds) of the Spring Configuration scenario

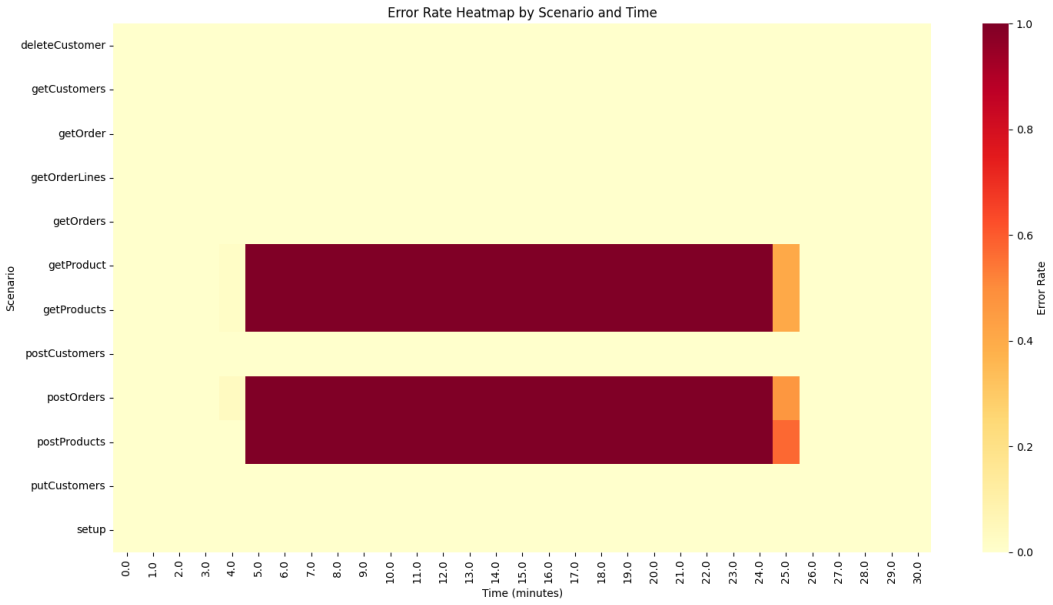

Figure A.10: Heatmap of the Kubernetes Configuration scenario

Scenario	Mean	Min	Max	Std
deleteCustomer	29.73	11.63	73.85	11.59
getCustomers	23.46	4.81	204.84	11.56
getOrder	2040.09	3.26	5030.81	2425.82
getOrderLines	2230.56	2.62	5028.11	2457.34
getOrders	1979.89	3.51	5030.91	2418.25
getProduct	16.57	3.50	162.88	9.50
getProducts	23.17	5.40	166.53	11.20
postCustomers	32.19	7.70	88.30	14.67
postOrders	2625.05	2.44	5027.52	2460.71
postProducts	34.67	10.76	87.42	14.36
putCustomers	27.76	8.99	70.37	10.24
setup	8.93	6.65	16.17	2.30

Table A.19: Response time statistics (milliseconds) of the Spring Database Connection scenario

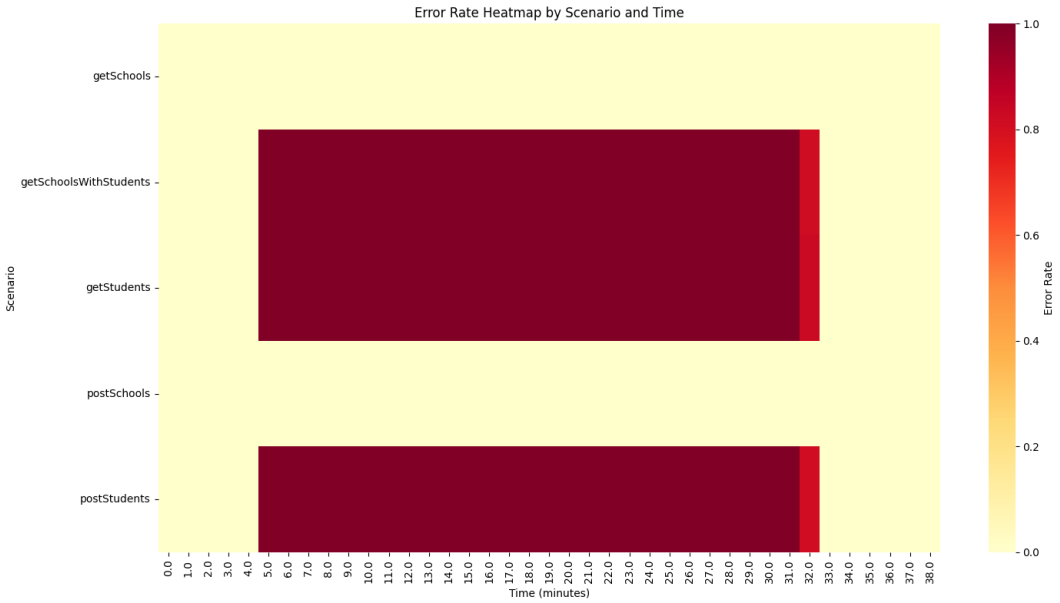

Figure A.11: Heatmap of the Spring Configuration scenario with tool deployed

Scenario	Mean	Min	Max	Std
deleteCustomer	29.18	6.51	71.62	14.38
getCustomers	21.10	5.89	153.43	10.82
getOrder	1615.45	2.87	5033.42	2288.67
getOrderLines	1688.11	2.85	5036.10	2311.16
getOrders	1313.50	2.98	5034.85	2150.47
getProduct	15.06	2.96	89.47	8.52
getProducts	20.25	4.76	123.19	10.33
postCustomers	27.14	5.08	92.30	13.81
postOrders	2138.00	2.29	5028.47	2403.93
postProducts	30.15	7.64	79.50	13.90
putCustomers	22.08	7.57	50.08	8.85
setup	11.97	7.18	20.26	3.68

Table A.20: Response time statistics (milliseconds) of the Kubernetes Pod Unhealthy scenario

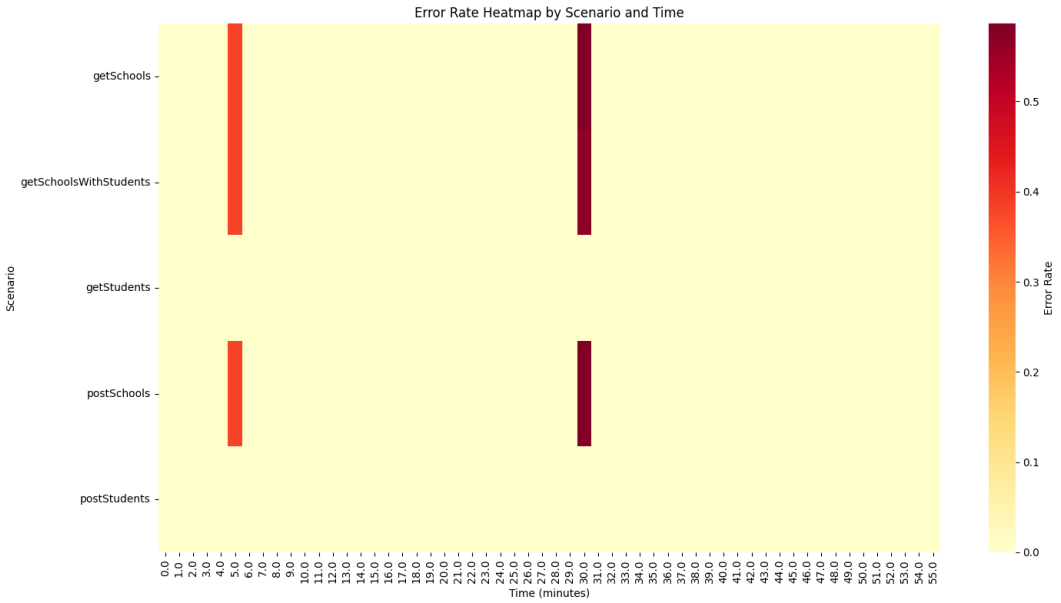

Figure A.12: Heatmap of the Service Down Kubernetes scenario with tool deployed

Scenario	Mean	Min	Max	Std
deleteCustomer	29.45	7.33	60.57	13.23
getCustomers	22.27	6.32	135.28	11.09
getOrder	1730.35	2.26	5048.36	2321.99
getOrderLines	1438.26	2.29	5032.90	2206.31
getOrders	1137.09	2.55	5050.30	2042.45
getProduct	16.00	3.41	95.34	9.18
getProducts	22.55	6.61	134.28	11.21
postCustomers	26.21	5.86	77.09	12.84
postOrders	2090.87	2.63	5044.09	2399.51
postProducts	30.11	8.28	80.39	13.37
putCustomers	26.82	8.01	58.75	10.52
setup	8.34	6.98	15.21	1.73

Table A.21: Response time statistics (milliseconds) of the Service Down Kubernetes scenario

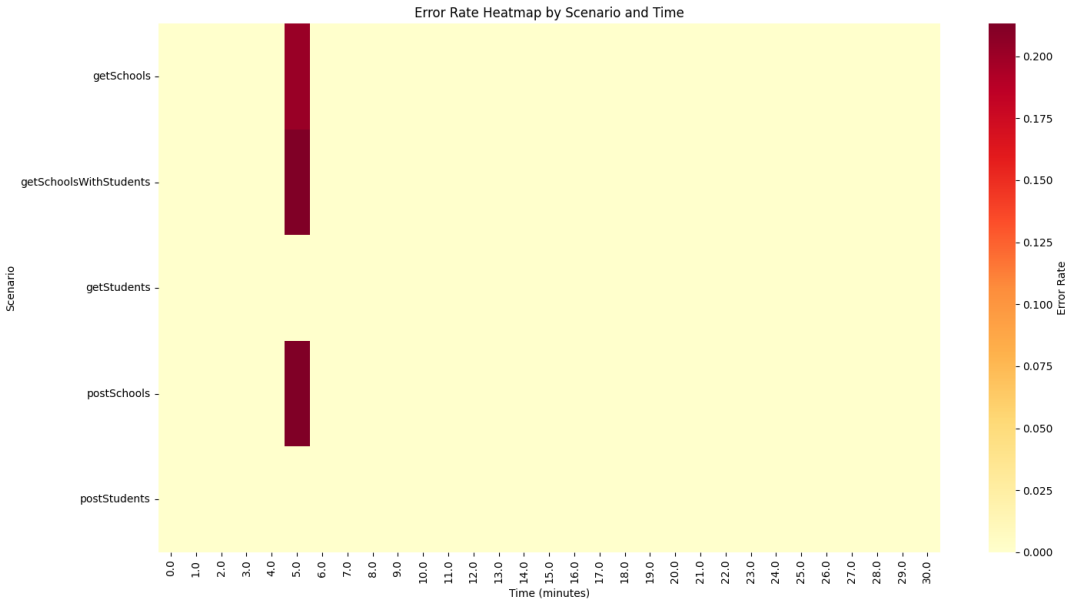

Figure A.13: Heatmap of the Kubernetes Invalid Image scenario with tool deployed

Scenario	Mean	Min	Max	Std
deleteCustomer	10884.46	0.00	60000.42	15724.97
getCustomers	1109.06	0.00	5008.80	2047.40
getOrder	1067.31	0.00	5016.92	2008.15
getOrderLines	1068.95	0.00	5016.70	2009.42
getOrders	1069.41	0.00	5017.42	2007.14
getProduct	1063.55	0.00	5017.07	2004.86
getProducts	1078.00	0.00	5007.29	2016.60
postCustomers	1753.48	0.00	5002.92	2349.25
postOrders	1380.22	0.00	5014.09	2194.89
postProducts	1710.27	0.00	5002.18	2341.73
putCustomers	1695.91	0.00	5002.22	2333.38
setup	8.55	6.63	12.34	1.51

Table A.22: Response time statistics (milliseconds) of the Kubernetes Node Problem scenario

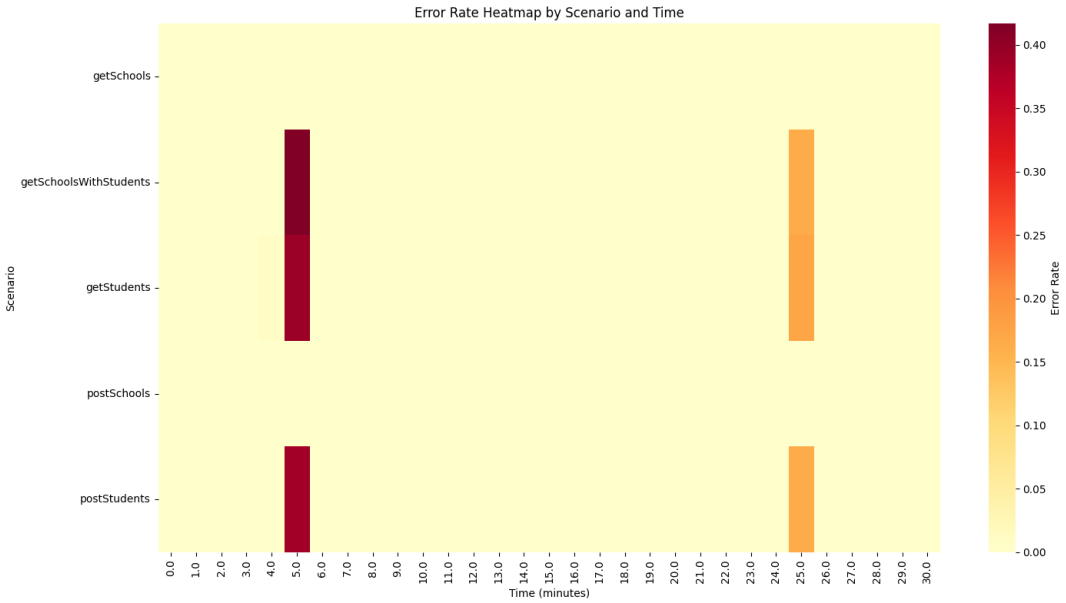

Figure A.14: Heatmap of the Kubernetes Configuration scenario with tool deployed

Scenario	Mean	Min	Max	Std
deleteCustomer	22.90	7.30	58.83	8.05
getCustomers	17.78	5.27	124.12	6.52
getOrder	1240.84	2.61	5002.60	2111.75
getOrderLines	1642.46	2.72	5023.50	2292.33
getOrders	1733.43	2.58	5005.21	2329.11
getProduct	15.15	3.62	119.95	6.51
getProducts	18.68	5.53	124.54	7.19
postCustomers	20.08	5.84	122.80	9.91
postOrders	2087.60	2.24	5021.39	2396.44
postProducts	23.04	8.18	124.23	9.60
putCustomers	21.06	7.45	34.93	5.64
setup	7.75	6.28	12.14	1.41

Table A.23: Response time statistics (milliseconds) of the Kubernetes Low CPU and Memory scenario

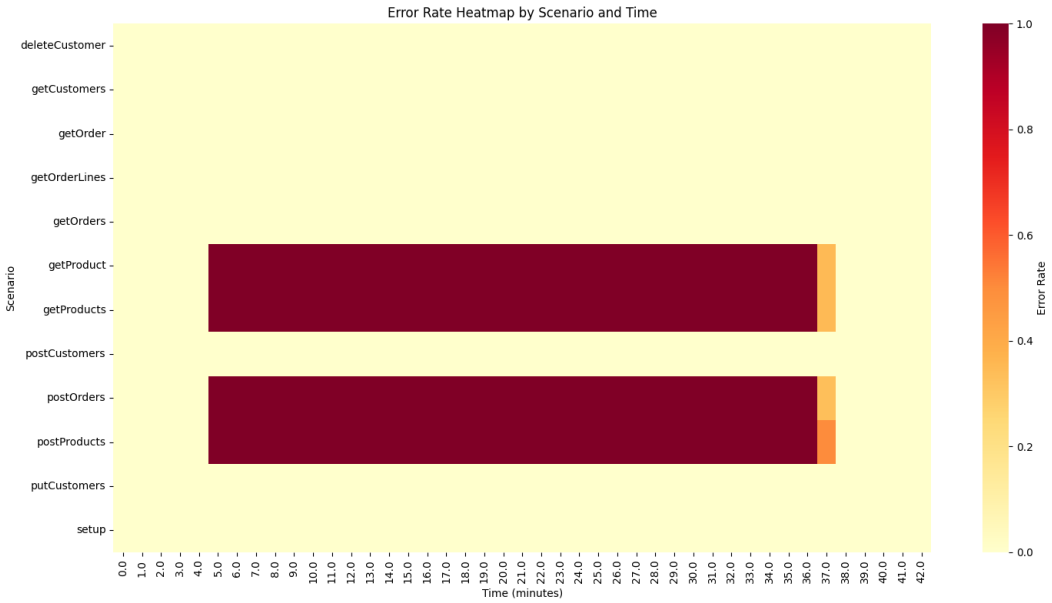

Figure A.15: Heatmap of the Spring Configuration scenario with tool deployed

Scenario	Mean	Min	Max	Std
deleteCustomer	15.72	4.33	49.33	8.09
getCustomers	15.60	4.77	83.06	6.82
getOrder	2373.64	2.61	5011.89	2459.80
getOrderLines	2392.52	2.96	5008.27	2448.18
getOrders	970.35	3.26	5005.31	1914.44
getProduct	12.48	3.70	78.49	6.39
getProducts	16.61	5.32	109.41	7.50
postCustomers	21.66	6.11	41.79	6.48
postOrders	2482.25	3.08	5020.33	2420.22
postProducts	25.17	9.13	52.70	6.88
putCustomers	14.20	5.69	33.65	5.83
setup	7.95	6.16	13.42	1.67

Table A.24: Response time statistics (milliseconds) of the Kubernetes Invalid Image scenario

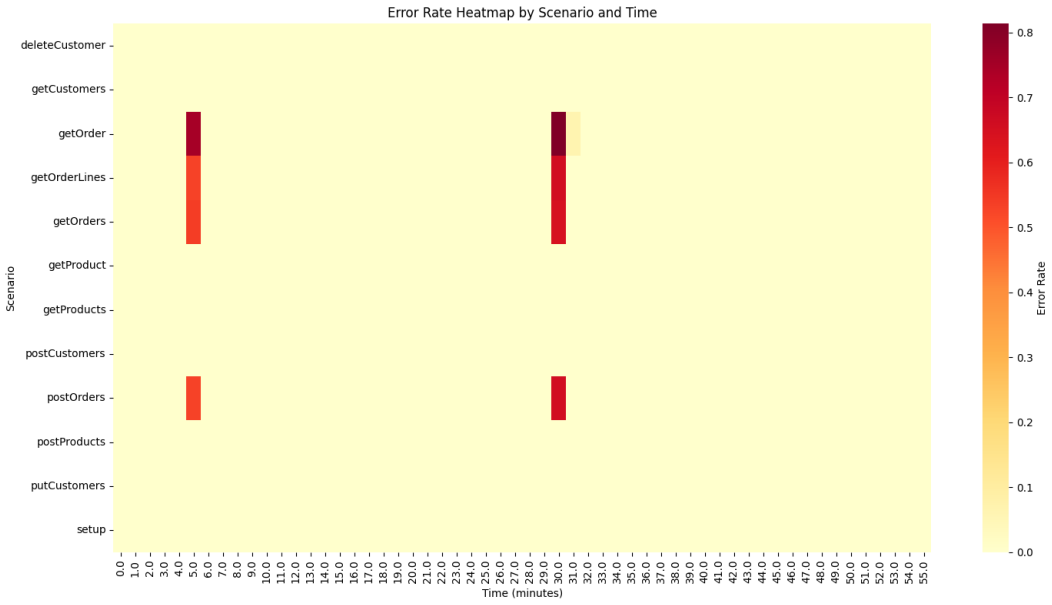

Figure A.16: Heatmap of the Service Down Kubernetes scenario with tool deployed

Scenario	Mean	Min	Max	Std
deleteCustomer	27.04	7.79	73.86	15.53
getCustomers	23.83	5.86	610.40	23.56
getOrder	19.71	4.32	429.42	19.10
getOrderLines	20.68	4.64	498.40	20.05
getOrders	22.17	4.65	514.67	22.76
getProduct	2054.40	5.84	13278.26	2212.85
getProducts	2092.61	7.63	13279.54	2230.37
postCustomers	24.12	5.77	79.01	13.26
postOrders	39.45	8.38	727.27	48.05
postProducts	2800.86	14.74	5021.46	2283.88
putCustomers	33.24	13.01	65.94	13.66
setup	11.38	7.65	24.91	4.75

Table A.25: Response time statistics (milliseconds) of the Kubernetes Configuration scenario

Scenario	Mean	Min	Max	Std
getSchools	1798.11	7.50	5070.44	2227.01
getSchoolsWithStudents	2395.61	4.58	5050.88	2447.63
getStudents	1124.27	5.71	5045.16	1982.55
postSchools	1693.64	5.88	5060.48	2201.94
postStudents	1213.80	4.42	5032.05	2061.62

Table A.26: Response time statistics (milliseconds) of the Spring Timeout scenario with tool deployed

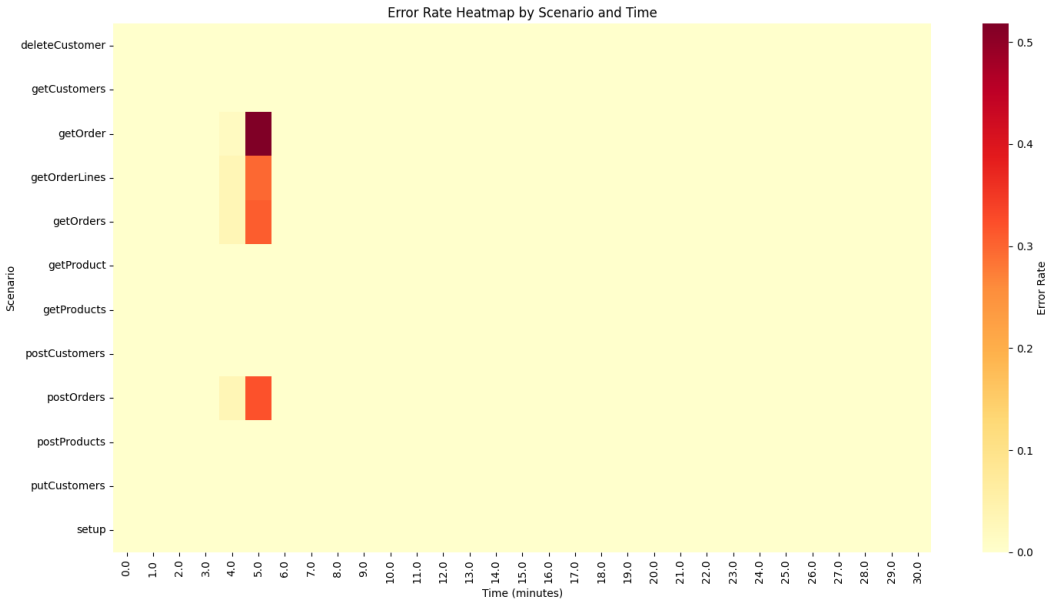

Figure A.17: Heatmap of the Kubernetes Invalid Image scenario with tool deployed

Scenario	Mean	Min	Max	Std
getSchools	28.34	5.42	5002.34	147.54
getSchoolsWithStudents	31.02	3.86	5000.80	164.32
getStudents	900.97	4.02	5034.50	1886.89
postSchools	23.65	4.88	5001.38	93.86
postStudents	1265.81	3.88	5012.58	2153.21

Table A.27: Response time statistics (milliseconds) of the Spring Third-Party Service scenario with tool deployed

Scenario	Mean	Min	Max	Std
getSchools	1714.66	4.67	902634.48	35454.41
getSchoolsWithStudents	1748.14	4.15	902632.37	35307.40
getStudents	642.47	4.41	5020.19	1618.82
postSchools	1068.71	4.67	902596.63	26627.79
postStudents	572.41	3.95	5012.60	1543.37

Table A.28: Response time statistics (milliseconds) of the Spring Request Service scenario with tool deployed

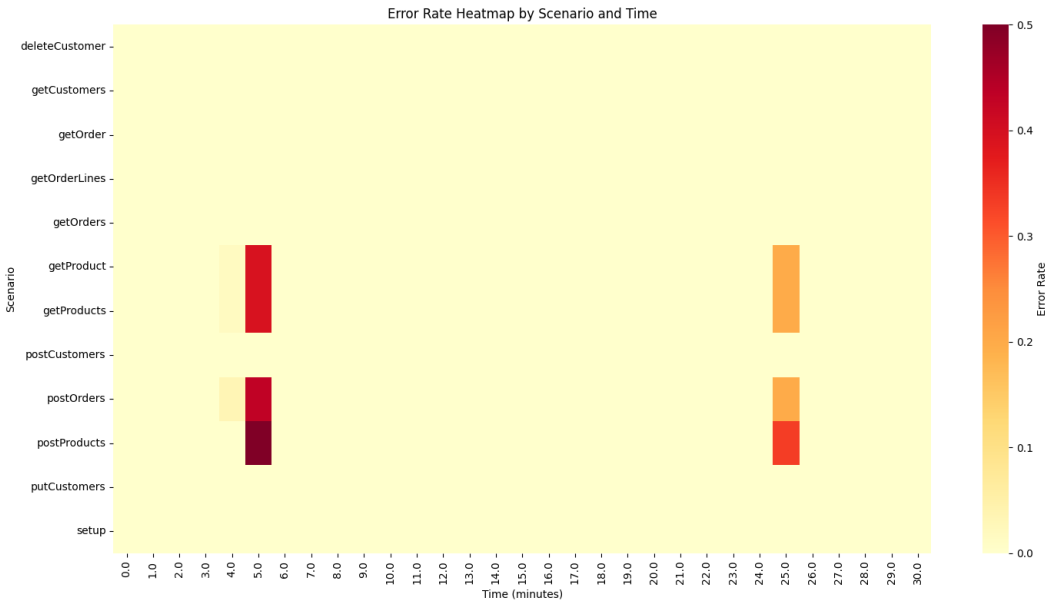

Figure A.18: Heatmap of the Kubernetes Configuration scenario with tool deployed

Scenario	Mean	Min	Max	Std
getSchools	19.87	6.12	415.37	11.41
getSchoolsWithStudents	20.97	4.52	1037.72	25.95
getStudents	2500.64	4.43	5014.78	2460.74
postSchools	19.48	5.00	392.99	12.72
postStudents	2641.41	4.07	5017.87	2476.23

Table A.29: Response time statistics (milliseconds) of the Spring Down scenario with tool deployed

Scenario	Mean	Min	Max	Std
getSchools	18.32	5.87	394.11	10.04
getSchoolsWithStudents	16.41	3.17	356.56	11.68
getStudents	3020.81	3.37	5010.78	2383.52
postSchools	15.80	4.06	373.84	12.40
postStudents	3217.60	3.72	5012.03	2295.36

Table A.30: Response time statistics (milliseconds) of the Spring Configuration scenario with tool deployed

Scenario	Mean	Min	Max	Std
getSchools	2959.32	3.99	5089.29	2410.86
getSchoolsWithStudents	3199.06	3.96	5058.52	2351.52
getStudents	39.37	7.86	1442.34	54.52
postSchools	2969.08	3.86	5059.01	2416.37
postStudents	35.24	4.91	1329.49	54.75

Table A.31: Response time statistics (milliseconds) of the Spring Database Connection scenario with tool deployed

Scenario	Mean	Min	Max	Std
getSchools	19.90	4.06	1466.75	30.25
getSchoolsWithStudents	23.16	3.87	1367.80	31.14
getStudents	21.54	6.52	476.11	12.52
postSchools	20.03	4.02	1370.93	31.09
postStudents	15.81	4.01	465.85	11.75

Table A.32: Response time statistics (milliseconds) of the Kubernetes Pod Unhealthy scenario with tool deployed

Scenario	Mean	Min	Max	Std
getSchools	39.91	4.48	4554.20	137.91
getSchoolsWithStudents	52.76	4.37	4556.33	164.35
getStudents	34.11	7.24	3298.76	82.66
postSchools	42.92	5.32	4557.18	145.65
postStudents	27.67	5.06	2727.90	76.55

Table A.33: Response time statistics (milliseconds) of the Service Down Kubernetes scenario with tool deployed

Scenario	Mean	Min	Max	Std
getSchools	44.75	0.0	5021.78	250.47
getSchoolsWithStudents	56.93	0.0	5005.82	273.19
getStudents	45.36	0.0	5022.99	248.55
postSchools	47.10	0.0	5015.40	237.85
postStudents	46.10	0.0	5013.96	237.37

Table A.34: Response time statistics (milliseconds) of the Kubernetes Node Problem scenario with tool deployed

Scenario	Mean	Min	Max	Std
getSchools	39.13	4.28	5005.43	272.70
getSchoolsWithStudents	61.53	4.83	5004.38	377.58
getStudents	20.01	6.12	625.54	18.23
postSchools	48.40	4.85	5002.80	318.39
postStudents	18.28	4.44	623.80	18.29

Table A.35: Response time statistics (milliseconds) of the Kubernetes Low CPU Memory scenario with tool deployed

Scenario	Mean	Min	Max	Std
getSchools	22.70	5.80	4133.35	123.19
getSchoolsWithStudents	26.28	5.84	4133.31	133.18
getStudents	18.94	5.99	928.51	22.80
postSchools	20.51	5.36	2074.19	61.76
postStudents	16.98	4.21	384.23	13.67

Table A.36: Response time statistics (milliseconds) of the Kubernetes Invalid Image scenario with tool deployed

Scenario	Mean	Min	Max	Std
getSchools	19.35	6.08	725.52	14.71
getSchoolsWithStudents	21.40	6.18	169.01	10.11
getStudents	45.72	4.93	5006.21	335.59
postSchools	18.22	4.23	692.49	17.81
postStudents	42.59	4.65	5001.15	332.72

Table A.37: Response time statistics (milliseconds) of the Kubernetes Configuration scenario with tool deployed

Scenario	Mean	Min	Max	Std
deleteCustomer	84.91	10.15	641.14	104.32
getCustomers	67.30	7.56	1863.09	112.97
getOrder	58.07	4.99	4679.65	179.98
getOrderLines	61.92	5.01	4682.08	184.07
getOrders	73.06	7.22	4678.94	188.97
getProduct	1728.55	6.32	5066.18	2238.26
getProducts	1443.45	4.67	5057.85	2126.14
postCustomers	81.15	9.60	1001.22	109.01
postOrders	1981.03	5.85	5053.95	2329.19
postProducts	2435.83	6.25	5018.22	2345.09
putCustomers	102.17	10.76	902.64	143.01
setup	8.40	6.67	14.60	1.75

Table A.38: Response time statistics (milliseconds) of the Spring Timeout scenario with tool deployed

Scenario	Mean	Min	Max	Std
deleteCustomer	35.36	12.65	165.96	22.29
getCustomers	27.15	7.34	375.57	16.74
getOrder	25.31	4.33	5000.30	153.64
getOrderLines	30.86	4.82	5001.27	193.83
getOrders	39.68	8.83	5002.99	202.69
getProduct	566.78	4.48	5014.02	1437.01
getProducts	686.80	5.64	5013.91	1582.36
postCustomers	36.27	10.34	167.44	17.61
postOrders	62.17	8.25	5000.25	153.65
postProducts	1160.78	15.80	5016.56	1973.60
putCustomers	28.05	11.25	60.71	11.40
setup	11.34	7.14	22.45	4.07

Table A.39: Response time statistics (milliseconds) of the Spring Third-Party Service scenario with tool deployed

Scenario	Mean	Min	Max	Std
deleteCustomer	40.64	10.94	352.68	35.12
getCustomers	34.96	7.89	3621.75	64.50
getOrder	118.60	4.50	5014.49	666.61
getOrderLines	161.38	5.05	5013.87	798.98
getOrders	147.47	5.00	5011.00	723.70
getProduct	275.06	4.29	5014.54	1054.73
getProducts	308.02	5.19	5020.18	1108.56
postCustomers	43.43	7.98	472.46	33.41
postOrders	260.86	11.44	5009.72	946.12
postProducts	611.70	12.16	5011.67	1536.21
putCustomers	46.86	11.86	331.07	44.61
setup	11.28	7.51	23.55	4.19

Table A.40: Response time statistics (milliseconds) of the Spring Request Service scenario with tool deployed

Scenario	Mean	Min	Max	Std
deleteCustomer	67.71	9.20	426.26	72.87
getCustomers	49.84	6.89	1043.69	85.10
getOrder	39.01	5.50	995.83	65.59
getOrderLines	41.34	6.39	1283.53	71.35
getOrders	43.38	5.92	1313.98	71.38
getProduct	1409.04	5.74	5040.97	1999.20
getProducts	1530.89	7.60	5027.44	2049.79
postCustomers	65.89	8.12	862.91	95.15
postOrders	89.26	10.05	4338.41	189.18
postProducts	1801.75	13.24	5016.25	2193.79
putCustomers	68.43	13.74	285.89	71.06
setup	9.30	6.84	18.44	2.54

Table A.41: Response time statistics (milliseconds) of the Spring Down scenario with tool deployed

Scenario	Mean	Min	Max	Std
deleteCustomer	42.12	15.48	168.54	26.17
getCustomers	24.57	7.83	236.77	12.47
getOrder	19.44	5.56	163.32	10.93
getOrderLines	20.36	5.84	163.95	11.23
getOrders	20.95	4.86	382.90	13.65
getProduct	171.40	3.84	5008.73	854.00
getProducts	286.02	4.12	5007.27	1115.89
postCustomers	42.21	13.94	160.85	17.40
postOrders	38.13	8.18	802.80	32.38
postProducts	649.90	4.96	5002.35	1636.04
putCustomers	34.74	10.58	71.09	13.47
setup	10.93	7.08	20.84	3.26

Table A.42: Response time statistics (milliseconds) of the Spring Configuration scenario with tool deployed

Scenario	Mean	Min	Max	Std
deleteCustomer	41.25	14.25	121.07	18.03
getCustomers	28.03	7.70	308.37	16.21
getOrder	2242.09	4.55	5015.30	2432.57
getOrderLines	2142.42	5.10	5013.09	2419.51
getOrders	2083.66	4.02	5014.99	2410.87
getProduct	19.84	3.77	301.14	13.94
getProducts	26.96	7.16	302.96	15.16
postCustomers	44.32	12.79	202.07	17.84
postOrders	2592.14	4.86	5012.79	2421.05
postProducts	47.87	12.89	201.09	18.97
putCustomers	31.69	12.40	102.44	14.77
setup	9.37	7.50	14.90	1.72

Table A.43: Response time statistics (milliseconds) of the Spring Database Connection scenario with tool deployed

Scenario	Mean	Min	Max	Std
deleteCustomer	83.34	10.43	1406.49	204.26
getCustomers	43.31	8.43	1904.45	83.32
getOrder	51.73	5.14	2552.73	117.53
getOrderLines	57.02	5.17	2841.52	126.05
getOrders	61.82	5.20	2840.74	124.39
getProduct	33.04	4.68	1883.87	70.96
getProducts	41.40	7.91	1924.38	82.91
postCustomers	67.96	12.51	1645.66	123.80
postOrders	159.25	7.49	4552.37	246.60
postProducts	67.50	11.87	1656.96	130.78
putCustomers	47.63	11.49	267.40	42.53
setup	13.82	9.40	30.04	5.07

Table A.44: Response time statistics (milliseconds) of the Kubernetes Pod Unhealthy scenario with tool deployed

Scenario	Mean	Min	Max	Std
deleteCustomer	38.83	14.06	127.80	20.28
getCustomers	26.19	6.40	351.75	15.79
getOrder	25.99	5.44	1878.62	47.81
getOrderLines	28.66	4.63	2063.34	53.34
getOrders	33.52	5.25	1976.55	53.63
getProduct	19.25	3.74	325.83	13.28
getProducts	25.38	6.79	343.66	14.48
postCustomers	39.60	11.90	144.69	17.94
postOrders	81.67	5.92	2473.51	90.69
postProducts	37.72	14.72	154.87	18.38
putCustomers	36.67	14.03	99.76	15.81
setup	12.68	8.71	20.96	3.51

Table A.45: Response time statistics (milliseconds) of the Service Down Kubernetes scenario with tool deployed

Scenario	Mean	Min	Max	Std
deleteCustomer	3759.73	0.00	60000.50	12837.98
getCustomers	341.44	0.00	5021.39	1121.47
getOrder	317.12	0.00	5016.72	1110.35
getOrderLines	321.42	0.00	5031.26	1110.10
getOrders	323.35	0.00	5032.92	1105.88
getProduct	321.95	0.00	5049.63	1110.85
getProducts	332.43	0.00	5028.40	1111.67
postCustomers	591.61	0.00	5030.34	1490.25
postOrders	516.83	0.00	5019.36	1266.47
postProducts	559.20	0.00	5029.38	1461.86
putCustomers	575.45	0.00	5005.25	1451.54
setup	17.71	8.57	63.78	12.89

Table A.46: Response time statistics (milliseconds) of the Kubernetes Node Problem scenario with tool deployed

Scenario	Mean	Min	Max	Std
deleteCustomer	125.24	19.24	3505.65	510.71
getCustomers	41.27	6.91	5029.98	163.14
getOrder	67.98	4.80	5000.96	285.60
getOrderLines	83.83	5.14	5000.57	373.29
getOrders	88.03	6.70	5003.19	385.86
getProduct	34.85	4.43	5003.21	188.17
getProducts	41.20	8.18	5028.74	191.68
postCustomers	68.15	13.83	3229.53	241.67
postOrders	185.61	6.82	5002.40	513.02
postProducts	64.87	10.95	3222.58	240.56
putCustomers	38.50	10.85	234.23	36.28
setup	10.38	7.45	18.62	2.60

Table A.47: Response time statistics (milliseconds) of the Kubernetes Low CPU Memory scenario with tool deployed

Scenario	Mean	Min	Max	Std
deleteCustomer	48.43	12.14	176.31	25.02
getCustomers	30.35	8.20	859.46	25.22
getOrder	39.94	5.15	5010.94	256.47
getOrderLines	39.76	5.35	5009.75	238.74
getOrders	48.61	7.98	5007.75	267.69
getProduct	24.40	5.81	864.19	24.61
getProducts	30.40	9.01	867.95	25.53
postCustomers	53.00	12.94	290.15	29.85
postOrders	92.21	7.04	5005.96	308.23
postProducts	48.76	15.74	196.38	21.03
putCustomers	39.84	11.47	84.94	15.82
setup	13.23	8.39	21.92	4.04

Table A.48: Response time statistics (milliseconds) of the Kubernetes Invalid Image scenario with tool deployed

Scenario	Mean	Min	Max	Std
deleteCustomer	42.49	16.03	78.15	13.89
getCustomers	28.15	7.28	280.59	18.91
getOrder	22.77	4.09	199.47	16.11
getOrderLines	24.58	5.69	208.46	17.64
getOrders	30.02	7.32	269.64	21.03
getProduct	41.14	4.58	5000.52	242.70
getProducts	48.51	6.71	5000.16	266.92
postCustomers	42.17	10.56	91.38	15.52
postOrders	68.38	13.19	654.68	50.29
postProducts	46.92	13.47	1022.51	74.53
putCustomers	40.71	19.53	121.93	17.78
setup	10.40	8.02	17.44	2.21

Table A.49: Response time statistics (milliseconds) of the Kubernetes Configuration scenario with tool deployed

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original