Multi-Versioning and Microservices:
A Strategy for Developing Reliable Software Systems

Nazanin Akhtarian

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES IN
PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE
OF MASTER OF SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE

YORK UNIVERSITY
TORONTO, ONTARIO

October 2023

Abstract

In the dynamic realm of software engineering, adaptability is key to sustaining system performance and reliability. Software iterations often bring about challenges such as unexpected bugs and performance issues, necessitating a nuanced approach to maintain system integrity. In this work, we propose employing software multi-versioning to enhance system reliability. We embark on an in-depth exploration of the reliability of microservices within chaotic environments. Using Chaos Mesh, we simulate a series of disruptions in a microservices-based application, i.e., the Online Boutique. Through real experimentation, we systematically introduce various chaos disruptions, such as Pod failures, response delay, and memory stress, to investigate their impact on the system's reliability.

We define a reliability metric that quantifies the robustness and efficiency of each software version under adverse conditions. Leveraging this metric, we introduce a dynamic controller that adjusts the population of each version, ensuring optimal resource distribution, reliability and system performance. Additionally, our research evaluates how the system adapts to varying workloads. We investigate how well the system can adjust its scalability—specifically, the number of replicas—in response to changes in CPU usage as the user load fluctuates. Our findings demonstrates the system's capability to scale dynamically based on work-

load demands, ensuring robustness and efficiency.

In conclusion, our study provides a detailed framework for employing software multi-versioning as a means to enhance system reliability. By devising a reliability metric and implementing a dynamic scaling system that responds to both reliability assessments and workload variations, we offer a comprehensive strategy to fortify systems against the unpredictable nature of software evolution, ensuring they remain resilient and make efficient use of resources.

Preface

The research presented in this thesis is the original work of Nazanin Akhtarian, and it has been conducted in collaboration between the Performant and Available Computing Systems (PACS) Lab led by Dr. Hamzeh Khazaei and the Centre for Research in Adaptive Software (CERAS) Lab led by Dr. Marin Litoiu.

Acknowledgement

First and foremost, I want to extend my profound thanks to my supervisors, Dr. Hamzeh Khazaei and Dr. Marin Litoiu, for their guidance and assistance during my postgraduate journey. Their constant presence and invaluable advice have been instrumental to my research. Without their help, achieving my goals would have been impossible.

I'm also grateful to my peers and associates in the Performant and Available Computing Systems (PACS) Lab and Centre for Research in Adaptive Software (CERAS) Lab, whose assistance has been essential throughout my academic journey.

Lastly, a big thank you goes out to my parents, whose continual support and motivation have been the backbone of my life's journey.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ARIMA	AutoRegressive Integrated Moving Average
Bi-LSTM	Bidirectional Long Short-Term Memory
CPU	Central Processing Unit
DF	Diversity Factor
DNS	Domain Name System
HPA	Horizontal Pod Auto-scaler
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
LSTM	Long Short-Term Memory
MTTF	Mean Time To Failure

OS	Operating System
RAM	Random Access Memory
UI	User Interface
VM	Virtual Machine

Chapter 1

Introduction

In today's world, software systems are an integral part of our daily lives, powering many applications. The continuous evolution of technology and the growing demands for high performance, availability, and security require advanced software development and deployment strategies. One such strategy that has emerged is software multi-versioning, which could enhance system adaptability, reliability, and performance. In the following sections, we will explain this thesis's core concepts and terms, namely, software multi-versioning, microservices, auto-scaling, load balancing, and utility functions. Finally, we discuss our contributions and provide an outline of the thesis.

1.1 Software Multi-Versioning

Software multi-versioning is a development and deployment strategy that involves maintaining multiple versions of a software. It allows the system to select and execute the most appropriate version of a software based on the current condition. This approach can improve performance, fault tolerance, reliability, security, and availability.

- **Performance optimization:** In situations where system workloads or hardware constraints vary, having multiple software component versions can help balance performance and resource utilization. A lightweight version can be employed during periods of high demand or on resource-constrained devices. Conversely, a resource-intensive version can provide more accurate results during periods of low demand or on more capable devices.

- **Fault tolerance and reliability:** Multi-versioning can increase fault tolerance and reliability in critical systems such as flight control. By maintaining multiple software component versions, developers can provide redundancy, ensuring continuous operation by preventing common bug failure.

- **Security:** Software multi-versioning can enhance the security of a system by offering alternative versions with varying security features. If a vulner-

ability is discovered in one version, another version can mitigate the risk until the issue is resolved.

- **Availability and adaptability:** Multi-versioning can improve the availability and adaptability of a software system by providing different versions suited for diverse user needs or environments. For instance, a feature-rich version can be available for users with high-end hardware. In contrast, users with lower-end hardware or limited connectivity can be offered a simplified version.

1.2 Microservices

Microservices have become an essential paradigm in software design and architecture. Microservices architecture breaks down a complex system into smaller and independent services. Each microservice has its own functionality and can be developed, deployed, and scaled independently, allowing for better resource utilization and cost efficiency [1]. This approach allows for greater flexibility, scalability, and maintainability compared to traditional monolithic architectures. In addition, microservices can be developed using different technologies and programming languages, allowing teams to choose the most suitable tools for each

service. Given this structure, software multi-versioning can be more selectively applied to individual components rather than the system as a whole. So, in this work, we choose our subject system an application with microservice architecture.

1.3 Auto-Scaling

As we delve deeper into the paradigms of software multi-versioning and microservices, it becomes essential to understand the scaling approaches that stand as the backbone in ensuring optimal performance and resource utilization. Elasticity is the key component of cloud computing that lets application owners to control unpredictable workloads that are inherent in internet-based services [2]. This can be done by increasing or reducing resources based on the workload to enhance service performance while lowering costs [3]. In environments where workloads fluctuate, being able to scale the resources dynamically, both at the microservices level and across different software versions, is vital. In this context, our focus is on effectively scaling Pod replicas to address variations in workload and maintain service stability.

The importance of auto-scaling algorithms in cloud computing is increasing. Properly provisioning and de-provisioning these replicas is essential, as under-

provisioning can lead to performance degradation and service unavailability, resulting in potential revenue loss for providers [4]. Conversely, over-provisioning results in resource wastage and becomes cost-inefficient due to pay-per-use pricing. Therefore, auto-scaling is essential in optimizing Pod replica resources and avoiding service level objective (SLO) violations.

Auto-scaling refers to a dynamic resource acquisition and release procedure that may be divided into two categories: proactive and reactive [5]. **Reactive techniques** analyze the system's current status and decide about the scaling based on predefined rules or thresholds. **Proactive techniques** examine the historical data, predict the future, and perform scaling decisions in advance. The disadvantage of the reactive strategy is that it will respond to a change in workload after it has already occurred. As a result, the system must take some time to reconfigure itself to handle the increased workload. On the other hand, in the proactive strategies, statistical methods such as AutoRegressive (AR), Moving Average (MA), AutoRegressive Moving Average (ARMA), and AutoRegressive Integrated Moving Average (ARIMA) are utilized to predict future workloads and prepare scaling actions ahead of time. However, one of the main drawbacks of proactive auto-scaling is its reliance on the accuracy of these predictions. Mispredictions can lead to resource wastage or potential performance degradation. Many existing

reactive auto-scaling solutions use rules with thresholds that are based on Central Processing Unit (CPU) utilization and memory [6–8]. Implementing reactive techniques is simple, but deciding on the correct threshold value is challenging because of the workload fluctuation. For proactive auto-scalers, only a few works have been done by employing machine learning algorithms [9–11], compared to the numerous works done by using time series data analysis [12–16]. In the scope of this work, we primarily employ a reactive auto-scaling approach. The choice is influenced by the desire to maintain a balance between efficiency, simplicity, and robustness. A comprehensive discussion on the existing works related to this topic will be elaborated in section 2.5.

1.4 Load Balancing

Load balancing methods refer to techniques used to distribute incoming network traffic across multiple servers to ensure that no single server is overwhelmed with too much traffic. This helps maintain the availability and reliability of applications and services. Several common load balancing methods include Round Robin, Least Connections, Least Response Time, IP Hash, and URL Hash [17].

In the real world, it is more common to have uneven servers in load balanc-

ing. In this case, utilizing adaptive load balancing or weighted algorithms such as Weighted Round Robin or Weighted Least Connections would be more helpful [18]. Adaptive load balancing is a dynamic and intelligent approach for distributing network traffic across multiple servers. Unlike traditional load balancing methods that rely on predetermined algorithms or fixed weights, adaptive load balancing based on factors like server performance, fluctuations in workload, or other contextual factors adjusts the distribution of incoming requests to optimize resource usage and maintain optimal system performance.

In this work, we adopt the approach of adaptive load balancing, specifically employing the method of Weighted Round Robin to distribute incoming requests among servers. Utilizing Nginx as the load balancer, we establish the weights following our system's configuration to ensure a harmonized workload distribution. Section 3.2.1 presents a detailed discussion of this implementation.

1.5 Utility Function

A utility function is a mathematical representation to quantify the desirability or value of different system states. It maps system states to a numerical value, indicating the level of utility or "usefulness" each state offers. This function is

beneficial for decision-making, as it helps determine the optimal choices or paths by maximizing (or in some cases, minimizing) the utility value. In brief, the utility function provides a structured way to make informed decisions based on the perceived value, ensuring that the system operates most efficiently [19].

The selection of an appropriate utility function is a crucial step in software engineering processes, as it directly impacts how system performance is evaluated and optimized. Our objective is to devise a function that maps lower performance metric values to higher utilities, thereby favouring systems with better performance characteristics. We focus on three critical metrics: response time, memory consumption, and the number of restarts. Each metric requires a specific approach to accurately capture the desired behaviour and ensure the utility function effectively guides decision-making towards optimal system configurations.

1. **Response Time:** Response time is a crucial measure of system performance. Lower response times generally indicate a more efficient system, so we are interested in a utility function that favours lower response times. Variability in response time is also significant, as consistent response times are often preferred. We calculate the standard deviation for response time values; versions with higher standard deviation are assigned a lower normalized utility value.

2. **Memory Consumption:** Monitoring memory consumption provides insights into the efficiency and stability of a software version. While different software versions may have varied memory requirements, sharp increases in memory consumption could signal problems, such as memory leaks.

   By calculating the standard deviation of memory consumption over time (e.g., every 30 seconds) we can detect such anomalies. Versions with higher standard deviation in memory usage are assigned a lower utility, indicating potential issues.

3. **Number of Restarts:** The number of restarts is a direct indicator of system stability. A higher number of restarts usually signals underlying issues. Consequently, we desire a utility function that rewards systems with fewer restarts.

The choice of a utility function can be diverse. The specific structure of the utility function U is determined by the preferences of the individual making the decision. There is no universally standard form that can be considered the best or most accurate representation for the preferences of all decision-makers [20].

In our work, some of the utility functions that can be chosen include linear, logarithmic, and exponential functions, among others. We chose a linear utility

function in our system for its simplicity and ease of understanding and implementation. However, different utility functions might suit other preferences. Here are some of the utility functions that can be utilized with their pros and cons:

Linear Utility Function

This utility function demonstrates a straightforward and proportional correlation between the metric x and the resultant utility value.

$$\text{utility}(x) = \frac{\max(X) - x}{\max(X) - \min(X)} \tag{1.1}$$

Where:

- X symbolizes the entire range of potential values for the metric.
- x indicates a specific value within this range.

Intuition: Consider a simple slider control. As you move the slider from one end (min) to the other end (max), the change in position directly and proportionally affects the value it represents. The relationship is consistent throughout.

The linear function is straightforward and easy to use. Yet, if a metric requires emphasizing lower values more, it might not be the best fit.

Exponential Utility Function

This function uses an exponential decay, giving a more noticeable drop in utility as the metric increases.

Intuition: Think of a car that loses its value faster in the initial years of purchase and slows down as it gets older. The decay in value (or utility) is more pronounced at the start.

$$\text{utility}(x) = e^{-\lambda x} \tag{1.2}$$

Where:

- λ is a positive constant determining the decay rate.

The exponential structure is adept at emphasizing a marked preference for lower metric values. Although its application offers adaptability via the λ parameter, it may demand more nuanced calibration and understanding.

Logarithmic Utility Function

This function provides a more gradual decrease in utility as the metric's value grows.

Intuition: Imagine charging a smartphone battery. At first, when the battery

is very low, it charges up quite rapidly. You might see it jump from 5% to 30% relatively quickly. But as the battery level approaches 100%, the charging rate slows down, especially in the last few percentages. It seems to take an especially long time to move from 95% to 100% compared to the initial rapid increase.

$$\text{utility}(x) = \frac{\log(1 + \max(X) - x)}{\log(1 + \max(X) - \min(X))} \tag{1.3}$$

Where:

- X represents the entire set of possible metric values.
- x is a specific metric value from this set.

The logarithmic form is suitable for scenarios that prefer a gentler decrease in utility. In terms of prioritizing lower metric values, it may not be as assertive as the exponential counterpart.

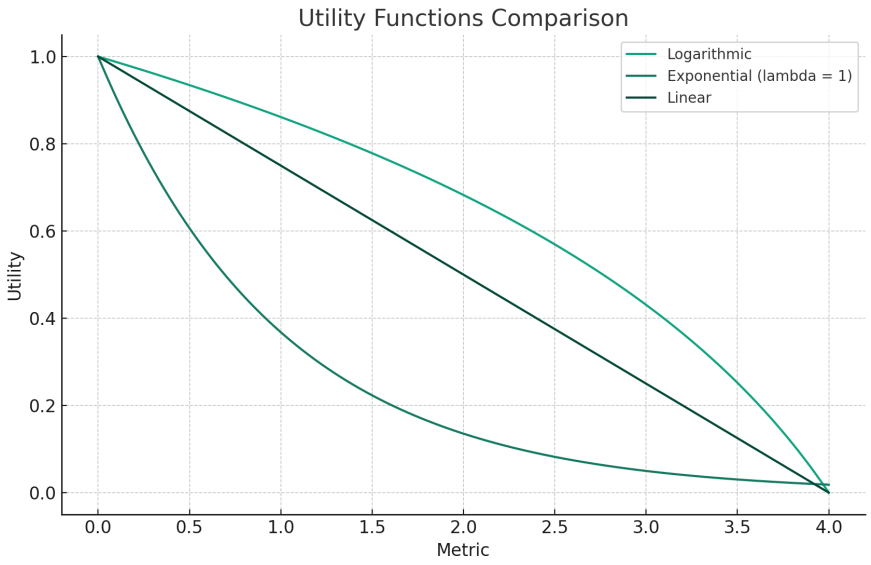

Figure 1.1: Comparison of utility functions: Logarithmic, exponential, and linear.

In summary, the selection of the utility function is not one-size-fits-all but must be carefully tailored to the specific needs and characteristics of the system. As we compute reliability scores for different versions repeatedly, we would like to choose a utility function whose computation is fast. In addition, as we want to choose a function that is easy to understand, the linear function is chosen. Nonetheless, depending on the user's priorities and the context of the system, other functions like exponential or logarithmic could also be suitable. A comparison between utility functions is illustrated in Figure 1.1.

1.6 Objectives and Contributions

This work addresses the challenges of maintaining system reliability in dynamic, cloud-native environments, specifically those orchestrated using Kubernetes. By proposing a solution grounded in the principles of software multi-versioning, we seek to provide a resilient approach to handling variable service demands without compromising system performance. Our key contributions are:

1. **Conceptualization of Multi-version Containers:** A unique approach that employs multi-versioning transparently at the container level, ensuring that users interact with services without awareness of underlying versions.

2. **Evolutionary Scaling Approach:** An innovative framework inspired by natural selection principles to enhance Kubernetes Pod scaling, ensuring that more reliable deployments are allocated additional resources.

3. **Reliability Scoring Mechanism:** Development of a dynamic scoring system that employs real-time metrics to evaluate the reliability of Kubernetes deployments. This mechanism informs system decisions, from load distribution to scaling.

4. **Adaptive Scaling:** An approach to ensure optimal system performance

amidst fluctuating demands. This strategy combines real-time monitoring, threshold-based decisions, and a historical analysis-based algorithm.

5. **Diversity Emphasis:** Introduction of the Diversity Factor (DF), which quantifies software version distribution, advocating for a balanced deployment approach over a solely reliability-centric one.

6. **Configurable Deployment:** Providing a customizable deployment configuration to fit specific user requirements.

Our contributions offer a comprehensive solution for system reliability in Kubernetes environments, laying the foundation for future cloud-native application advancements.

1.7 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 presents the related work. In Chapter 3, we explain about our methodology. In Chapter 4, we discuss the evaluation of our proposed methods by way of real implementation on the cloud. Chapter 5 lists our contributions, explains the threats to the validity and elaborates on the possible future directions for this research. Finally, a conclusion summarizing the benefits of the proposed approach is provided in Chapter 6.

Chapter 2

Related Work

In this section, we discuss prior work that is related to this thesis. In particular, we discuss related work on software multi-versioning, software multi-versioning for containerized systems, microservices in action, reliability for microservices, and scaling techniques.

2.1 Software Multi-Versioning

Software multi-versioning is a technique that involves creating multiple versions of a software program that are functionally equivalent but differ in implementation and/or design. Researchers have investigated the potential benefits of software multi-versioning for a variety of purposes, including enhancing a system's

security, safety, reliability, and availability. Larsen et al. [21] examined the impact of automated software redundancy on system security, while Franz et al. [22] utilized software multi-versioning as a defence mechanism. They suggested that having multiple versions of a system could make it more difficult for attackers to target a specific version, thereby improving system security.

Persaud et al. [23] used genetic algorithms to enhance security by employing software redundancy. Cigsar et al. [24] investigated the use of software multi-versioning to improve the reliability of repairable systems, and Gracie et al. [25] noted that redundancy has been employed for safety purposes. Gorbenko et al. [26] applied software multi-versioning to a web service to improve its functionality and features, such as availability and reliability.

Overall, software multi-versioning has been utilized for various purposes and has been shown to have potential benefits in improving the security, reliability, and availability of software systems. Borck et al. [27] propose FEVIS, a program diversification approach for detecting new cyber attacks. FEVIS generates program variants and runs them in parallel to detect attacks through behavior divergence. The authors provide a proof of concept and describe the application of FEVIS to an open-source web server, demonstrating its ability to detect multiple classes of attacks.

2.2 Software Multi-Versioning for Containerized Applications

Software multi-versioning for containerized systems refers to the practice of managing and deploying multiple versions of software within container environments. Prior research has primarily utilized software multi-versioning in containerized systems to enhance fault tolerance [28, 29], security, and reliability [30].

Wang et al. [28] proposed using multi-versioning on critical components of cloud-based software to improve fault tolerance, reducing cost and complexity by applying it selectively. Meanwhile, Zheng et al. [29, 30] demonstrated that multi-versioning can improve reliability and fault tolerance in service-oriented systems. However, multi-versioning can affect the system's quality of service, and Zheng et al. proposed an optimization problem to maintain the system's quality of service while improving reliability.

Gholami et al. [31] explored a cost-effective approach to meet the performance requirements of containerized software systems. They introduced DockerMV (Docker with Multi-Versioning), an open-source extension of the Docker framework that supports multi-versioning of containerized software systems. They used different versions of a microservice (light-weight or heavy-weight), primarily for

the purpose of scaling the application.

In the context of microservices and Kubernetes, Mohamed and El-Gayar [32] evaluated the end-to-end latency prediction of microservices workflows. They compare different machine learning models and resource metrics to predict the performance of containerized applications. Pinto et al. [33] evaluated the user acceptance testing for multi-tenant cloud applications and investigate the detection of faults in different variants of the application.

2.3 Microservices in Action

Microservices architecture breaks down a complex system into smaller and independent services. Each microservice has its own functionality and can be developed, deployed, and scaled independently. There are many new research studies exploring the use of the microservices for building different applications.

Timur et al. [34] proposed a scalable solution for deploying deep learning facial recognition systems using distributed microservices. By using Docker for containerization, they distributed data processing tasks across a scalable cluster. Lu et al. [35] presented microservice-based platform for Space Situational Awareness (SSA) data analytics. The platform's efficacy was tested through two

experimental applications, which were built to handle complex computations and provide insights into satellite coverage and space object conjunction assessment. Asaithambi et al. [36] presented a Microservice-Oriented Big Data Architecture (MOBDA). The aim was to create a scalable system that can handle large-scale transportation data effectively, thereby enhancing the management and optimization of the public transport system. The authors carried out the evaluation part by focusing on Singapore's public transport system, using data from the Singapore Land Transport Authority (LTA) DataMall. Ali et al. proposed a microservices-oriented strategy to offer effective data analytics solutions in the form of modular microservices [37]. The researchers divided the data analytics procedure into key steps and established them as independent microservices. The use case scenario was related to monitoring health conditions using Internet of Things (IoT) in real-time, which can help in predicting health situations and offering personalized services.

In [38], the authors proposed a new federated DL model (Fed-TH) for detecting and classifying cyber threats in industrial IoT. They utilized a container-based edge computing framework and explored microservice placement methods to address latency issues. The model's effectiveness was validated using two datasets, ToN_IoT [39] and LITNET-2020 [40], demonstrating high accuracy and perfor-

mance in detecting cyber threats. Trilles et al. proposed an IoT architecture that brings together diverse IoT devices and tools into one ecosystem [41]. Their proposed architecture uses microservices and serverless technologies, aiming to guarantee scalability, stability, and reusability. The authors applied their proposed IoT architecture in viticulture. They developed a system called "SEnviro for Agriculture" which is designed to monitor environmental conditions, particularly vineyards, aiming to enhance the quality of the production. Xu et al. [42] proposed BlendSM-DDM, a decentralized security services architecture for decentralized data marketplaces, combining containerized microservices and blockchain technology. They aimed to provide a decentralized, scalable, and auditable data exchange platform.

2.4 Reliability for Microservices

Many research have been done in recent years to improve the availability and reliability of microservices. Vincenzo and Dragi [43] introduced a new technique for task unloading using Pareto optimization, focusing on three goals: response time, reliability, and cost.

Clab et al. [44] suggested a delay adaptive strategy for replica synchroniza-

tion, as well as a recovery method relying on load balancing, to address the issues of replica synchronization and failed node recovery in cloud applications. Metrics such as Mean Time To Failure (MTTF), Rate of Failure Occurrence (RoCoF), hazard rate, and Probability of Failure on Demand (PoFoD) have been formulated to gauge software reliability. Among these, MTTF and hazard rate are most applicable for system failure that follows statistical regularity over time.

In Ref. [45], an overall theoretical estimate of MTTF was presented to align with QoS needs by adjusting redundancy levels. Chen et al. [46] made advancements in cloud service by reducing uncertainty in the scheduling of workflow applications. In Ref. [47], an algorithm that uses Repetitions in Reserve (RIR) to minimize redundancy computes workflow reliability without the need to evaluate the reliability of individual tasks.

Liu et al. in [48] tried maximizing the entire cloud application's reliability, focusing more on the microservice with higher criticality. They modelled the reliability of cloud applications by the fluctuation frequency and amplitude of reliability in a cycle. In their next work [49], the authors addressed the challenge of improving the reliability of microservice-based cloud applications. The authors measured the reliability of a microservice using a two-part strategy, involving redundancy and the implementation of a circuit breaker. Redundancy ensures

backup instances are available to replace any failure, while the circuit breaker isolates failures to prevent them from affecting other dependent microservices.

In [50], the authors present a model that uses Petri nets to predict reliability. For a single service, there's a proposal to predict reliability from four factors: availability of the network environment, hermit equipment, discovery reliability, and critical reliability. Zang et al. [51] suggested a method for creating a service dependency graph to enhance a service fault tree with a reliability model. In the area of stable, cloud-based software operation, there's considerable interest in fault-tolerant technology to improve reliability in uncertain environments.

Gao et al. [52] suggested a dynamic method for mobile e-commerce service workflow. Others, like Sharif et al. [53] and Yao et al. [54], proposed techniques to ensure reliability through quality improvement or node normalization. Ray et al. [55] proposed an innovative scheduling algorithm combining resubmission and replication, while other strategies, like [56], focus on preempting faults based on CPU temperature. Shi et al. [57] and Wang et al. [58] offered strategies to ensure real-time applications and to measure reliability sensitivity in cloud services, respectively.

Pietrantuono et al. [59] introduced an in vivo testing technique targeted at assessing the reliability of microservice architecture during its operational phase.

Hasselbring and Steinacker [60] developed designs for microservice architectures with a focus on scalability and reliability, enabling substantial enhancements in load performance and scalability through the detailed breakdown and decentralized handling of data.

Wang et al. [61] explored an automated fault diagnosis approach using statistics in microservice applications, estimating anomalies in workflow and employing tree edit distance to identify and locate the specific microservice. An in-depth analysis of software reliability as a crucial metric was conducted by Wang and others [62], who also presented a reliability testing procedure as part of the acceptance testing phase. Camilli et al. introduced MIPaRT, a methodology to conduct intergrated performance and reliability testing for microservices [63].

Various works have been done for estimating reliability of IoT systems. The authors of [64] proposed a reliability model focusing on QoS (Quality of Service) parameters violations. Their approach incorporates a failure and recovery model to calculate reliability, emphasizing network aspects. Behera et al. proposed a method that models reliability by considering system availability, including the availability of programs, subsystems, and required input data [65].

In [66], authors introduced three redundancy models whose reliability is computed in two ways, through Reliability Block Diagram (RBD) and Reliability

Graph (RG). In paper [67] reliability is estimated using a Markov Decision Model (MDM) in a probabilistic manner. Despite its potential, the complexity of MDMs for large IoT systems can be computationally demanding. The model in [68] views an IoT system as a set of interacting instances or "black boxes." It builds reliability evaluation on transaction success ratios and it's heavily dependent on historical data, limiting its real-time applicability over extended periods.

The author in [69] introduced the Enhanced Real-Time CORE (ERT-CORE), a real-time reliability model for Multi-Agent IoT Systems (MAIS), characterized by linear time complexity and efficiency. In this approach, authors computed reliability based on three parameters for their system's components. The parameters were workload, average request processing time, and availability. Their specified components were CPU, RAM, network interface, and storage.

Sabino et al. [70] conducted a systematic literature review on energy consumption in microservices architectures, exploring approaches to improve system reliability and manage power consumption. Chen and Xiao [71] designed a multi-objective and parallel particle swarm optimization algorithm for container-based microservice scheduling, aiming to improve service reliability and efficiency. Liu et al [72] proposed a reliability modelling and optimization method for microservice-based cloud applications using a multi-agent system, aiming to

maximize reliability and minimize delay within budget constraints.

Despite the variety of studies on reliability, few current research works consider the reliability of microservice-based applications. Specifically, we focus on enhancing the reliability of containerized systems through multi-versioning.

2.5 Auto-Scaling Approaches

In the rapidly evolving landscape of cloud computing, the strategies for scaling resources have seen a wide variety of approaches. In this section, we explain some of the notable strategies in scaling, emphasizing their methodologies and the outcomes derived from their implementation.

Al-Dhuraibi et al. [7] proposed the ELASTICDOCKER architecture, which is based on the reactive scaling strategy. ELASTICDOCKER vertically scales both memory and virtual CPU cores resulting from the workload. The disadvantage of vertical scaling is its limited resources for hosting the machine capacity. When the hosting machine does not have enough resources, ELASTICDOCKER executes a live migration of the container to solve this issue. Their experimental results showed that this approach helps to reduce expenses for customers, make better resource utilization, and improve Quality of Experience (QoE).

The Horizontal Pod Auto-scaler (HPA) [8] is a control loop in Kubernetes (K8S) that increases the number of pods based on CPU utilization in the reactive approach, regardless of how well the workload or application is working. So, by changing the number of pods, it'll keeps the overall average CPU usage at the preferred level.

A cloud workload prediction module utilizing ARIMA was proposed by Rodrigo et al. [13], which may dynamically provide virtual machines to handle the anticipated requests. They used actual HTTP web server request traces from the Wikimedia Foundation for the performance assessment. This model has 91% accuracy for seasonal data, but it is unsuitable for workload that is not seasonal. Additionally, the authors didn't compare this model with any other strategies.

A prediction model employing genetic algorithms (GAs) to combine numerous statistical techniques was proposed by Messias et al. in [16]. To test their model, they used three logs that were taken from actual web servers. The outcomes demonstrated that the suggested methodology produces the best outcomes. The ARIMA model, however, produces the highest result with the best forecast accuracy when using the NASA web server records [73]. The auto.ari-ma() function of the R package, suggested in [74], was used to automatically select the value for p, d, and q, resulting in the construction of the ARIMA model.

The authors of [75] used Artificial Neural Networks (ANN) to estimate the task duration and resource utilization. A crawler was set up to collect the job length, the number of files, and the size of the GitHub repositories in order to create the ANN dataset. The ANN model was then offline trained. Comparing the suggested model to a simple linear prediction model, the prediction error was less than 20%. However, this model is unsuitable for real world applications because it was trained offline.

In order to forecast workload in the future, Prachimutita et al. [9] presented a new autoscaling framework using Artificial Neural Network (ANN) and Recurrent Neural Network (RNN) models. They then changed to the required RAM and CPU core based on the anticipated workload to maintain services in accordance with the Service Level Agreement (SLA). Utilizing multiple-step ahead forecasting, the performance was assessed using access data from the 1998 World Cup Web site [76]. The outcome shown that when more steps were forecasted ahead, the ARIMA model's accuracy decreased. Additionally, the Long Short-Term Memory (LSTM) model outperformed the MLP model in terms of accuracy. An autoscaling architecture based on machine learning was also proposed by Imdoukh et al. in [10]. The researchers used the 1998 World Cup website dataset and the LSTM model. They compared their results with those produced by both

the ANN model and the ARIMA model. In one-step forecasting, the suggested LSTM model's prediction error was found to be somewhat greater than that of the ARIMA model, while its prediction speed was found to be 530–600 times faster.

Tang et al. [11] proposed a container load prediction model by using the Bidirectional Long Short-Term Memory (Bi-LSTM) approach, which uses the container's past CPU utilization load to predict the future load. Comparing the accuracy of the proposed model to the ARIMA and LSTM models, it showed the lowest prediction error. The authors did not specify how to set up the parameters of the suggested model, though. Additionally, the research solely focuses on forecasting future load; it does not use it to address auto-scaling issues. A hybrid elastic scaling technique for Kubernetes was proposed by Ming Yan et al. [77] by combining reactive and proactive methods. In order to predict future workload, the proactive strategy uses the Bi-LSTM model to learn the history of physical host and pod resource usage (CPU and Memory usage). To produce elastic scaling judgements, the Bi-LSTM prediction model is then integrated with the online reinforcement learning using reactive model. The trials demonstrate that it can support the system in edge computing environments in meeting the microservice SLA. In comparison to the ARIMA, LSTM, and RNN models, the Bi-LSTM model also exhibited the lowest prediction error for the root mean square error

(RMSE) metric. However, no approach has been taken to mitigate oscillations. Laszlo Toka et al. [78] proposed using an AI-based forecast method for proactive scaling policy. The Hierarchical Temporal Memory (HTM), LSTM, and AR techniques are included in their the AI-based forecasting technique. The rate of incoming web requests is learned and predicted from each model. Additionally, they suggested the use of a backtesting plugin called HPA+ to seamlessly swap between the AI-based model and the HPA. The HPA+ will change to HPA and vice versa if the AI-based model's performance declines. The findings demonstrate that the HPA+ can dramatically reduce the amount of requests that are refused at the expense of slightly higher resource usage.

Dang-Quang and Yoo proposed a system architecture based on the Kubernetes orchestration system with a proactive custom autoscaler using a deep neural network model to calculate and provide resources ahead of time [5]. The proposed autoscaler focuses on the analysis and planning phases of the MAPE loop. In the analysis phase, they used a prediction model based on Bi-LSTM, which learns from the historical time-series request rate data to predict the future workload. The proposed model was evaluated through a simulation and the real workload trace from NASA [73] and FIFA [76]. The planning phase implements cooling down time (CDT) to prevent the oscillation mitigation problem, and when the

workload decreases, the scaler uses the resource removal strategy (RRS) to simply remove an amount of surplus pods to maintain the system stability while handling it faster if a burst of workload occurs in the next interval. The results of an experiment conducted with different datasets indicate that the proposed prediction model achieves better accuracy not only than the LSTM model but also than the state-of-the-art statistical ARIMA model in terms of short- and long-term forecasting. The prediction speed is 530 to 600 times faster than that of the ARIMA model and almost equal compared to the LSTM model when examined with different workloads. Compared to the LSTM model, the Bi-LSTM model performs better in terms of autoscaler metrics for resource provision accuracy and elastic speedup. Moreover, when the workload decreases, the architecture was found to remove a part of pod replicas, thus making it efficient when handling burst of workload in the near future. Finally, the proposed system design shows better performance in accuracy and speed than the default HPA of the Kubernetes when provisioning and de-provisioning resources.

Dogani et al. [79] proposed proactive auto-scaling method in Kubernetes using an attention-based gated recurrent unit (GRU) encoder-decoder (K-AGRUED). This technique extracts temporal patterns, in contrast to many existing methods, and predicts the resource usage of several future steps based on cool-down time

(CDT). The results showed that, in comparison to state-of-the-art approaches, the proposed strategy reduces prediction error by 2-25%. In comparison to two earlier experiments and Kubernetes' conventional HPA, their method greatly decreased scaling operations and under-provisioning. They claimed that K-AGRUED increases the scaling speedup by a factor of up to five in a real environment. For the evaluation, they used the FIFA [76] and the NASA [73] web server log set.

Dang-Quang and Yoo proposed multivariate deep learning prediction model to predict future resource workload for cloud computing environment [80]. Their prediction model uses Bi-LSTM. They did their experiments on real world workload dataset of GWA-T-12 Bitbrains and they concluded that the proposed multivariate Bi-LSTM model outperforms the univariate Bi-LSTM model in prediction accuracy. They chose multivariate time series data analysis because that allows researchers to find relationships between variables, in contrast to univariate time series data. The authors, in their next work [81], added feature selection step in their solution. They used the Pearson correlation method to select the best features as inputs for the multivariate model. They claimed that this new multivariate Bi-LSTM model outperforms the previous multivariate LSTM model in prediction accuracy.

Balla et al. [82] presented an auto-scaling mechanism called Libra for applica-

tions running on top of Kubernetes. Libra combines both vertical and horizontal scaling techniques to optimize resource allocation and scaling in cloud computing environments. Ju et al. introduced a Proactive Pod Autoscaler (PPA) for edge computing applications that operate within a Kubernetes environment [83]. The main advantage of PPA over the default Kubernetes autoscaler (HPA) is its ability to forecast incoming workloads of applications using time series prediction methods. This proactive approach helped in scaling the applications ahead of time, based on the predicted load.

Chapter 3

Methodology

In the context of modern cloud-native environments, assessing the reliability and performance of applications deployed on Kubernetes[1] can be a challenging task due to the dynamic and complex nature of such systems. In this work, we develop an evolutionary approach to Kubernetes Pod scaling. The idea of an evolutionary approach is inspired by the process of natural selection, where individuals with the most characteristics suited to their environment have a better chance of survival. Similarly, in our system, the deployments with the highest reliability scores have more Pods. In this work, we propose a self-adaptive framework that is based on the MAPE (monitor, analyze, plan, and execute) concept [84]. Our system

[1] https://kubernetes.io

continuously monitors the environment, analyzes data, plans actions, and executes these actions, thereby exhibiting the MAPE loop's properties.

- Monitor: We use Prometheus for continuous monitoring of the system. It collects metrics such as the number of Pod restarts and memory consumption. As the average response time metric should be calculated from the client side, we employ Nginx logs to collect this data. Also, beyond the system metrics, our monitoring approach extends to continually observing the system's workload for scaling decisions.

- Analyze: This is the core of our evolutionary scaling process. Here, we analyze data obtained during the monitoring phase and calculate the reliability score for each Kubernetes deployment based on a weighted average of three key metrics. Additionally, we examine the workload to see whether scaling actions are needed.

- Plan: During this phase, strategies are formulated for upcoming actions. This includes decisions like adjusting the number of Pod replicas for each software version, scaling in/out the overall replica count of our software, or altering the traffic distribution among different software versions. The foundation for these strategies is the reliability scores determined in the

analysis phase.

- Execute: The execution phase in our system involves changing the replicas and adjusting the load balancer configuration to control the traffic flow.

3.1 System Architecture

This section presents an overview of the system architecture, consisting of two essential microservices: Load Balancer and Scaling Engine. Figure 3.1 illustrates the system's architecture. Our system distributes the workload dynamically based on the reliability score of microservice versions and manages their replicas according to these scores. The load generator component generates and directs the load to the load balancer. The load balancer, powered by Nginx[2], evenly distributes incoming requests from the load generator across multiple backend servers representing multi-version microservice. Our scaling engine calculates the reliability score for each microservice version and adjusts replicas based on this score. It also continually assesses and changes the overall Pod replica count based on workload monitoring, enhancing system reliability to variable demands and optimizing the allocation of resources. Additionally, it provides configura-

[2]https://www.nginx.com

tions to Nginx for weight distribution. In the following sections, we will explain these critical microservices in more detail.

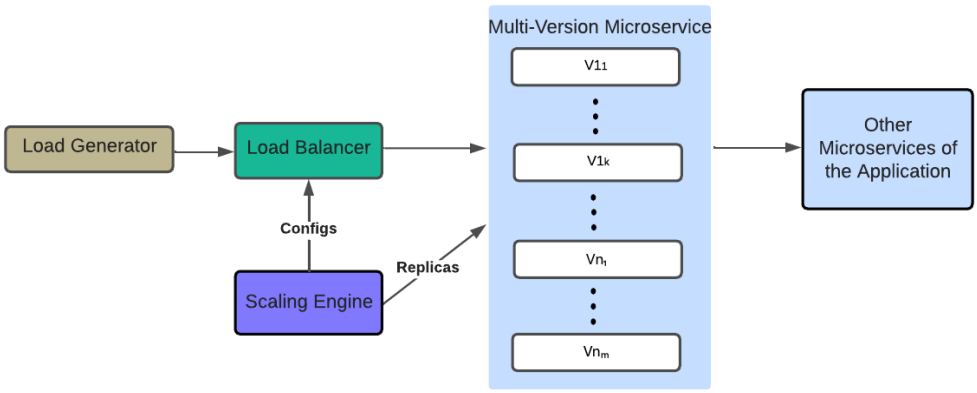

Figure 3.1: System architecture diagram. This diagram illustrates the overall structure of the proposed system, detailing component interactions and data flow paths.

3.2 The Scaling Engine

Our scaling engine has two main components: an automated load balancer configuration updater and a reliability scoring system. In the following subsections, we discuss these parts.

3.2.1 Automated Load Balancer Configuration Updater

The first component of the system involves the automatic updating of the Nginx load balancer configuration in response to changes in the reliability scores of Kubernetes deployments. Nginx is a popular open-source tool that can be used as a reverse proxy, HTTP cache, and load balancer. In this case, Nginx serves as a tool to manage and distribute incoming application traffic among Kubernetes Pods, enhancing the application's ability to handle large traffic volumes. This method eliminates the risk of any Pod becoming overloaded by equally sharing the workload across all Pods. Dynamically updating the Nginx configuration based on the reliability scores of the deployments guarantees that more traffic goes toward the most reliable deployments. In other words, deployments with more Pods receive a larger share of the total workload. As a result, our system's overall performance and reliability are significantly improved.

3.2.2 Reliability Scoring System

This module is designed to calculate the reliability scores of different deployments running on Kubernetes, interfacing with Prometheus to fetch key real-time metrics. The reliability score for each Kubernetes deployment is computed as a weighted average of three metrics. The system translates monitored values into

an overall reliability assessment by employing a utility function that linearly combines these metrics. To bring more depth to this assessment, we leveraged formal utility theory to construct a utility function following the methodology proposed by [85]. Here, we describe the structure of the utility function, detail the metrics under consideration, and explain the methodology adopted to compute the reliability scores.

Utility Function Definition

We define the reliability utility function $U_{\text{reliability}}$ based on the observed metrics at a given time t as:

$$U_{\text{reliability}}\left(\theta(t) \mid \phi\right) = \sum_{i=1}^{N} w_i \cdot u_i\left(\theta_i(t) \mid \phi\right)$$

Where:

- $\theta(t)$ represents the metrics vector at time t.
- ϕ denotes a set of additional parameters influencing the utility function.
- w_i are the weights assigned to each metric, ensuring $\sum_{i=1}^{N} w_i = 1$.
- $u_i(\theta_i(t)|\phi)$ are individual utility functions for each metric, further elaborated in the scoring methodology subsection 3.2.2.

In our experiments, we consider $N = 3$ metrics which are detailed in the next section.

Monitored Metrics

We consider three principal monitoring metrics:

- **Number of Restarts:** This measures how often the deployment has had to restart, with a higher number indicating potential instability.
- **Response Time Variability:** While the average response time sheds light on the deployment's responsiveness, its variability is equally crucial. Significant fluctuations in response times can indicate unpredictable behaviour or potential bottlenecks.
- **Memory Consumption:** By monitoring memory usage, we can detect potential memory leaks or understand how efficiently the deployment utilizes resources.

The utility functions u_i corresponding to each metric are defined using linear functions, which are detailed in section 1.5.

Scoring Methodology

The scoring system is designed to monitor these metrics continuously, calculate the reliability scores, and adjust the number of Pod replicas accordingly. This dynamic adjustment means that our system can quickly respond to deployment performance and reliability changes. The score calculation system involves the following steps:

1. **Metric Retrieval**: Fetch raw metrics from Prometheus using specific queries.

2. **Metric Normalization**: Normalize metrics linearly between 0 and 1. This uses the utility function u defined for each metric m in the metric set I, for each deployment d in the deployment set D. The function is:

$$\forall d \in D, \forall m \in I : u_m(d) = 1 - \frac{\text{metric}_m(d) - \min(\text{metric}_m(D))}{\max(\text{metric}_m(D)) - \min(\text{metric}_m(D))} \quad (3.1)$$

   In cases where $\max(\text{metric}_m(D)) = \min(\text{metric}_m(D))$, we set $u_m(d) = 1$.

   With this approach, a lower metric value results in a higher normalized value, thus giving deployments with better performance (lower frequency of restarts, less variability in response time, and lower memory usage) a higher reliability score.

In addition, the normalization process allows us to meaningfully combine different metrics, which may initially operate on different scales.

3. **Reliability Score Calculation**: Compute a weighted average of metric scores for each deployment's overall reliability score. Metric weights can vary based on importance. We assign the following weights to the metrics, but they can be modified according to the specific requirements.

   For each $d \in D$,

$$\begin{aligned} \text{Reliability\_Score}(d) = &\text{responseTimeWeight} \times u_{\text{responseTime}}(d) \\ &+ \text{restartWeight} \times u_{\text{restarts}}(d) \\ &+ \text{memoryWeight} \times u_{\text{memoryUsage}}(d) \end{aligned} \tag{3.2}$$

   With current metric weights being:

   - restartWeight: 0.5
   - memoryWeight: 0.3
   - responseTimeWeight: 0.2

The functioning of the reliability scoring and the replica calculation systems can be understood by referring to Algorithm 1 and Algorithm 2, respectively.

Algorithm 1 Monitoring without scaling

```
Require: Global variable: MONITORING_TIME, ACTION_TIME
 1: while True do
 2:     if time elapsed is MONITORING_TIME then
 3:         for deploymentVersion in deploymentVersions do
 4:             getPrometheusData(deploymentVersion)
 5:         end for
 6:     else if time elapsed is ACTION_TIME then
 7:         reliability_scores ← empty array
 8:         for deploymentVersion in deploymentVersions do
 9:             Compute metrics and reliabilityScore for deploymentVersion
10:             reliability_scores.add(reliabilityScore)
11:         end for
12:         ADJUSTREPLICADISTRIBUTION(deploymentVersions)
13:     end if
14: end while
```

3.2.3 Replica Adjustment

In our system, we carry out replica adjustment to optimally distribute the total number of replicas across multiple software versions. This distribution is guided by the respective reliability scores of each version. Our approach ensures that every software version maintains a minimum of one replica. This is important as reliability can fluctuate over time, and maintaining at least one replica for each version safeguards against potential failures in other versions.

First, the proportional number of replicas for each software version is calculated based on its reliability score. If this calculation results in a fraction, the system considers the fractional part in the subsequent step.

Algorithm 2 Adjust replicas distribution based on reliability scores

Require: Global variable: $TOTAL\_REPLICAS$

function ADJUSTREPLICADISTRIBUTION(deploymentVersions)
    $total\_score \leftarrow 0.0$
    for score in reliability_scores **do**
        $total\_score \leftarrow total\_score + score$
    end for
    newReplicas $\leftarrow$ array of zeros, one for each deployment version
    fractionalParts $\leftarrow$ array of zeros, one for each deployment version
    $allReplicas \leftarrow 0$
    for index, score in reliability_scores **do**
        $proportionalReplica \leftarrow \frac{TOTAL\_REPLICAS \times score}{total\_score}$
        newReplicas[index] $\leftarrow$ integer part of $proportionalReplica$
        fractionalParts[index] $\leftarrow proportionalReplica -$ newReplicas[index]
        if newReplicas[index] = 0 **then**
            newReplicas[index] $\leftarrow 1$
        end if
        $allReplicas \leftarrow allReplicas +$ newReplicas[index]
    end for
    Sort indices of fractionalParts in descending order
    $difference \leftarrow allReplicas - TOTAL\_REPLICAS$
    if $difference < 0$ **then**
        for $i = 0$ to $-difference$ **do**
            newReplicas[indices[i]] =+ 1
        end for
    else if $difference > 0$ **then**
        for $i =$ len(indices) $- 1$ to len(indices) $- difference$ **do**
            if newReplicas[indices[i]] > 1 **then**
                newReplicas[indices[i]] =- 1
            end if
        end for
    end if
end function

Once the proportional replica counts are computed, the system checks if the sum of these counts (step 8 in Algorithm 2) either exceeds or falls short of the total replicas. If it exceeds, the system removes excess replicas from versions with lower fractional parts. If it falls short, the system assigns additional replicas to versions with higher fractional parts.

Before actual scaling operations are triggered, the system evaluates whether any significant change in the replica count for any software version has occurred. Specifically, the system checks if the change exceeds a predetermined threshold (e.g., 5% of the total replicas). If this condition is met, the adjustment process is initiated as described in Algorithm 3. This keeps our system efficient and avoids unnecessary and temporary adjustments.

During the replica adjustment process, the system updates the replica count for each software version based on the newly calculated figures. Furthermore, adjustments are made to the Nginx configuration file to guarantee that traffic is proportionally distributed across the different software versions.

3.2.4 Adaptive Scaling for Dynamic Workloads

While the previous section focused on adjusting replicas based on the reliability scores of individual deployments, our system also incorporates a mechanism for

Algorithm 3 Check for adjusting condition and apply changes

```
1: THRESHOLD_PERCENTAGE ← 5.0%
2: adjust_replicas ← False
3: for each deploymentVersion in deploymentVersions do
4:     Get currentReplica and newReplica
5:     absoluteDifference ← abs(newReplica − currentReplica)
6:     percentageDifference ← 100 ∗ (absoluteDifference/TOTAL_REPLICAS)
7:     if percentageDifference ≥ THRESHOLD_PERCENTAGE then
8:         adjust_replicas ← True
9:         break
10:    end if
11: end for
12: if adjust_replicas = True then
13:    for each deploymentVersion in deploymentVersions do
14:        Update deploymentVersion with newReplica
15:    end for
16: end if
```

dynamic scaling in response to overall workload changes. This dynamic scaling capability, detailed in Algorithm 4, is vital to ensure optimal system performance during periods of variable demand.

Monitoring CPU Utilization

Our system's active monitoring of CPU utilization, as depicted in step 3 of Algorithm 4, allows us to comprehend the demand placed on our deployments effectively. By using the Prometheus query, we can continuously gauge the current CPU usage across all Pods in the multi-version microservice, ensuring our system's performance aligns with the prevailing demand.

Algorithm 4 Monitoring with scaling

Require: Global variables:

- $MONITORING\_TIME$, $ACTION\_TIME$
- $MAX\_REPLICAS$, $MIN\_REPLICAS$
- $TOTAL\_REPLICAS$

```
1:  while True do
2:      if elapsed = MONITORING_TIME then
3:          currentCPUValue ← getCPU()
4:          currentScalingState ← determineScalingAction(currentCPUValue)
5:          scalingHistory.add(currentScalingState)
6:          for deploymentVersion in deploymentVersions do
7:              getPrometheusData(deploymentVersion)
8:          end for
9:      else if elapsed = ACTION_TIME then
10:         scalingAction ← decideScalingBasedOnHistory(scalingHistory)
11:         if scalingAction = Increase & TOTAL_REPLICAS < MAX_REPLICAS then
12:             TOTAL_REPLICAS =+ 1
13:         else if scalingAction = Decrease & TOTAL_REPLICAS > MIN_REPLICAS
    then
14:             TOTAL_REPLICAS =- 1
15:         end if
16:         reliability_scores ← empty array
17:         for deploymentVersion in deploymentVersions do
18:             Compute metrics and reliabilityScore for deploymentVersion
19:             reliability_scores.add(reliabilityScore)
20:         end for
21:         ADJUSTREPLICADISTRIBUTION(deploymentVersions)
22:     end if
23: end while
```

Algorithm 5 Scaling action decision-making

```
Require: Global variables:
MAX_CPU ← 60%
MIN_CPU ← 20%
function DETERMINESCALINGACTION(cpu)
    if cpu > MAX_CPU then
        return Increase
    else if cpu < MIN_CPU then
        return Decrease
    else
        return NoChange
    end if
end function
function DECIDESCALINGBASEDONHISTORY(history)
    increaseCount ← 0
    decreaseCount ← 0
    for action in history do
        if action = Increase then
            increaseCount =+ 1
        else if action = Decrease then
            decreaseCount =+ 1
        end if
    end for
    if decreaseCount > 2 then
        return Decrease
    else if increaseCount > 1 then
        return Increase
    else
        return NoChange
    end if
end function
```

Threshold-Based Approach

Our system employs a threshold-based approach, configured with user-defined upper and lower CPU utilization limits, to guide scaling decisions. The function `determineScalingAction` (steps 3-11 in Algorithm 5) evaluates the current CPU usage against predefined thresholds. Specifically:

- If CPU usage exceeds 60%, indicating heightened demand, the system initiates a scale-out operation, augmenting the total replicas by one.
- If CPU usage falls below 20%, reflecting reduced demand, a scale-in operation is activated, diminishing the total replicas by one.

The strength of this approach is in its flexibility. As will be detailed in Listing 3.1, these thresholds are user-configurable and can be adjusted based on system's requirements.

Historical Analysis for Scaling Decisions

Our method of making scaling decisions involves an historical analysis approach that takes into account past decisions to avoid making hasty scaling decisions that could lead to system instability or resource wastage.

The function `determineScalingAction` (steps 12-29 in Algorithm 5) performs an analytical review of recent scaling activities. Based on this examination:

- An increase in replicas is triggered if there are more than one recent indications to scale out.
- Conversely, a decrease is initiated if there are over two recent signals to scale in.
- In the absence of any strong inclination towards scaling out or in, the system once again opts for the 'NoChange' state.

3.3 Diversity Factor: Quantifying Version Variation

As we've explored the system's ability to scale dynamically based on workload, particularly focusing on CPU utilization, a natural question arises: "Why not simply increase the replicas of the perceived most reliable version when scaling out?"

The answer lies in the inherent unpredictability of software behaviour. Diversifying software versions in a deployment does more than merely distribute workload; it acts as a strategic defence. By ensuring a mix of different software versions, we are effectively hedging against unforeseen bugs, vulnerabilities, or performance issues specific to a single version. Even if a version is perceived as

"good" today, software behaviour can be context-sensitive, and what works optimally under one set of conditions might falter under another. Ensuring diversity ensures that the system remains resilient across broader scenarios, mainly when unforeseen issues arise. Consequently, the introduction of the *Diversity Factor (DF)* is crucial. It quantitatively captures how evenly distributed the software versions are within our deployment.

3.3.1 Definition

The idea behind the DF is to measure how equally distributed the software versions are within the replicas. A high DF indicates that the replicas are nearly equally distributed across different versions, while a low DF signifies an imbalance with some versions being used more than others.

Mathematically, we define the DF as:

$$DF = \frac{1}{\sigma(R)}$$

where $\sigma(R)$ represents the standard deviation of the replicas distribution across different software versions. For simplicity, if there are three software versions

$V1$, $V2$, and $V3$, with replica counts $R1$, $R2$, and $R3$ respectively, then

$$\sigma(R) = \sqrt{\frac{(R1-\bar{R})^2 + (R2-\bar{R})^2 + (R3-\bar{R})^2}{3}}$$

where $\bar{R}$ is the average replica count, $\bar{R} = \frac{R1+R2+R3}{3}$.

For maximum diversity, which is when each software version has equal replicas (e.g., 5 replicas each out of 15), the standard deviation $\sigma(R)$ is 0. This makes $DF \to \infty$. In practice, this can be capped at a large value or represented as the maximum possible value. Conversely, for minimum diversity, when one version is used more than the other two, DF will be at its lowest but always greater than zero.

3.3.2 Implications and Usage

The DF serves as a compass guiding the dynamic adjustment of replicas. Instead of mindlessly adding more replicas of a reliable version when scaling out, we aim for a diverse distribution, ensuring reliability, robust feedback, and proactive response to potential software pitfalls. This approach champions not only immediate system reliability but also the longevity and future adaptability of the system in the face of evolving challenges.

3.4 The Load Generator

The Online Boutique application has a load generator microservice which continually sends requests, simulating realistic user shopping patterns directed at the frontend. We customize this microservice for our specific needs. We use Locust[3] which is an open-source, scriptable and scalable performance testing tool that allows customized test cases using Python. With the integration of Locust, we introduce enhanced browser-based accessibility, as illustrated in Figure 3.2. This allows us to easily configure parameters such as the count of concurrent users and their spawn rates before initiating the workload. Additionally, by adapting the user wait time within our load generator, we aim to place a more concentrated load on the frontend microservice, enabling us to better observe and analyze the system's overall behaviour under modified conditions.

[3]https://locust.io

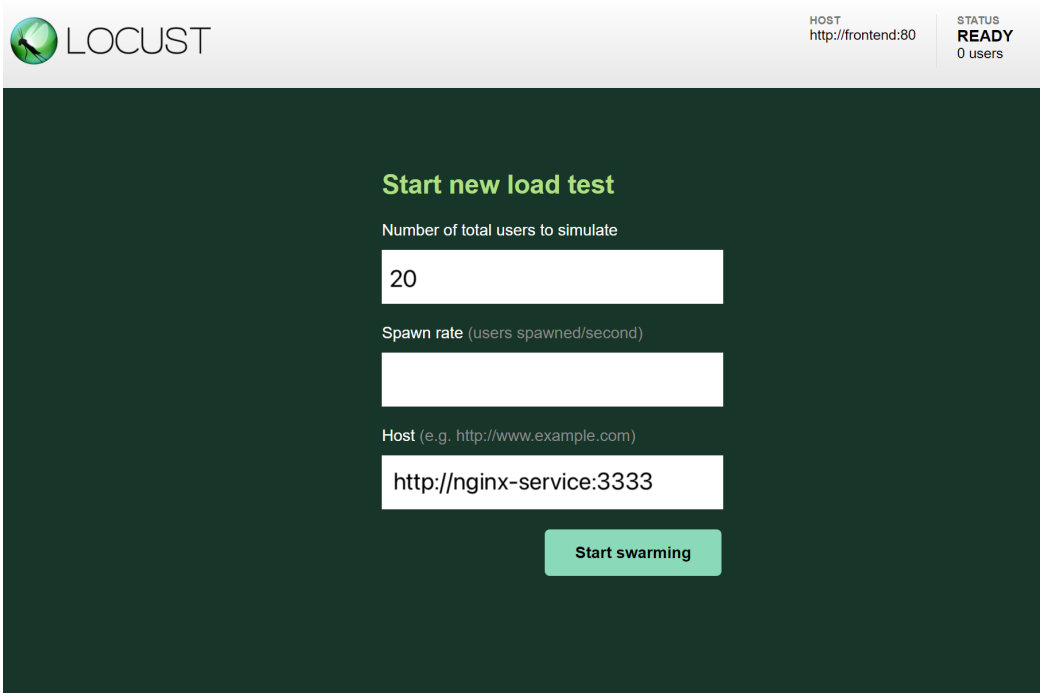

Figure 3.2: A snapshot of the Locust load testing tool's User Interface (UI), demonstrating the settings and parameters available for simulating user traffic.

3.5 The System's Parameters Configuration

In our proposed application, a flexible configuration mechanism has been implemented. This allows users to customize the behaviour of the system based on their specific needs. This is primarily achieved through the definition and manipulation of key parameters that control the functionality of our system.

Firstly, the user should create a deployment using our application's Docker

image by applying the corresponding Kubernetes configuration file. Listing 3.1 shows the corresponding command in the Kubernetes system.

Listing 3.1: Applying the deployment file

```
kubectl apply -f AppDeploymentFile.yaml
```

The user should then set the environmental variables for the created deployment. For example, if “ReplicaBalancer” is the deployment name defined in “AppDeploymentFile.yaml”, the command would be as follows:

Listing 3.2: Setting environmental variables

```
kubectl set env deployment/ReplicaBalancer \
DEPLOYMENT_IMAGES_REPLICAS="imageName1*replica1,imageName2*replica2,..."
TOTAL_REPLICAS=9
MONITORING_TIME=30s
ACTION_TIME=2m
MAX_REPLICAS=24
MIN_REPLICAS=3
MAX_CPU=60
MIN_CPU=20
SCALING=true
```

In this configuration, environmental variables play crucial roles, each detailed as follows:

- `DEPLOYMENT_IMAGES_REPLICAS`:
  - **Purpose**: Assigns the initial number of replicas for different Docker images. This parameter is essential and the system requires it for operation.
  - **Pattern**: The value for this parameter should follow the pattern:

    ```
    image_owner/image-name:version*number-of-replicas
    ```

    Where:
    - **image_owner/image-name:version** - Specifies the Docker image
    - **number-of-replicas** - Indicates the desired number of replicas.
  - **Example**: In `nazaninakhtarian/buggy-app:latest*5`, the Docker image `nazaninakhtarian/buggy-app` with the tag `latest` is allocated 5 replicas.
  - **Automatic Distribution**: If the number of replicas for an image is not provided, the system evenly distributes the `TOTAL_REPLICAS` among

the specified images. If an even distribution is not possible, the system will distribute the remainder sequentially until all replicas are assigned.

- `MONITORING_TIME`:
  - **Purpose**: Defines the frequency of system monitoring intervals.
  - **Default**: If this variable is not set, it defaults to 30 seconds.
- `ACTION_TIME`:
  - **Purpose**: Indicates the interval after which the system takes scaling or adjustment actions based on the collected monitoring data.
  - **Default**: By default, it is set to four times the `MONITORING_TIME`, unless otherwise specified.
- `TOTAL_REPLICAS`:
  - **Purpose**: Indicates the total number of replicas distributed among all versions of the microservice at startup. The system adjusts this number dynamically based on workload fluctuations.
  - **Default**: If unspecified, a default value (e.g., 9) is used as the starting point but may be altered by the system based on workload.

- `MAX_REPLICAS`:
  - **Purpose**: Sets an upper limit on the total replicas that can be provisioned across all pods of the multi-version microservice.
  - **Default**: If not specified, the default value is set to 24.
- `MIN_REPLICAS`:
  - **Purpose**: Sets a lower limit on the total replicas that the system must maintain across all pods of the multi-version microservice.
  - **Default**: If not specified, the default value is set to 3.
- `MAX_CPU`:
  - **Purpose**: Specifies the upper CPU utilization threshold for initiating scaling actions.
  - **Default**: If not specified, the default value is set to 60%.
- `MIN_CPU`:
  - **Purpose**: Sets the lower CPU utilization threshold below which the system considers scaling down.
  - **Default**: If not specified, the default value is set to 20%.

- `SCALING`:
  - **Purpose**: Enables or disables automatic scaling based on the observed workload.
  - **Default**: If not indicated, the system assumes `SCALING=false`, which turns off auto-scaling.

Chapter 4

Experimental Evaluation

4.1 Experimental Setup

To evaluate our method, we deployed a cluster including two Virtual Machines (VMs) within the Compute Canada cloud infrastructure[1]. We set one VM as a master node and the other as a worker node in our Kubernetes cluster. The VMs were set up with the following specifications: The Operating System (OS) used was Ubuntu 22.04.2 Jammy (x64). This configuration provided 15 GB of RAM, powered by 4 VCPUs. In terms of storage, the VMs were equipped with a primary disk storage of 20 GB and an additional ephemeral disk storage of 83 GB.

[1] https://arbutus.cloud.computecanada.ca

4.2 Subject System

In our experiments, we analyze the Online Boutique application[2], a cloud-native microservices demo application. Online Boutique is composed of 11 microservices, with each service playing a specific role within the application. Table 4.1 provides descriptions of these microservices.

The Online Boutique application simulates an e-commerce website where users can browse products, add them to their shopping cart, and proceed to checkout. The architecture of the application is illustrated in Figure 4.1. Furthermore, visual previews of the application's home page and checkout page can be found in Figures 4.2 and 4.3, respectively.

4.2.1 Critical Microservice Identification

Determining which microservices are critical to a system is a significant task. The failure of a critical microservice can have a profound impact on the overall system's functionality. As an application may comprise numerous microservices, provisioning various versions for all of them can be resource-intensive and costly. Hence, identifying a select number of critical microservices is vital to minimize expenses while creating highly reliable applications within budget constraints.

[2]https://github.com/GoogleCloudPlatform/microservices-demo

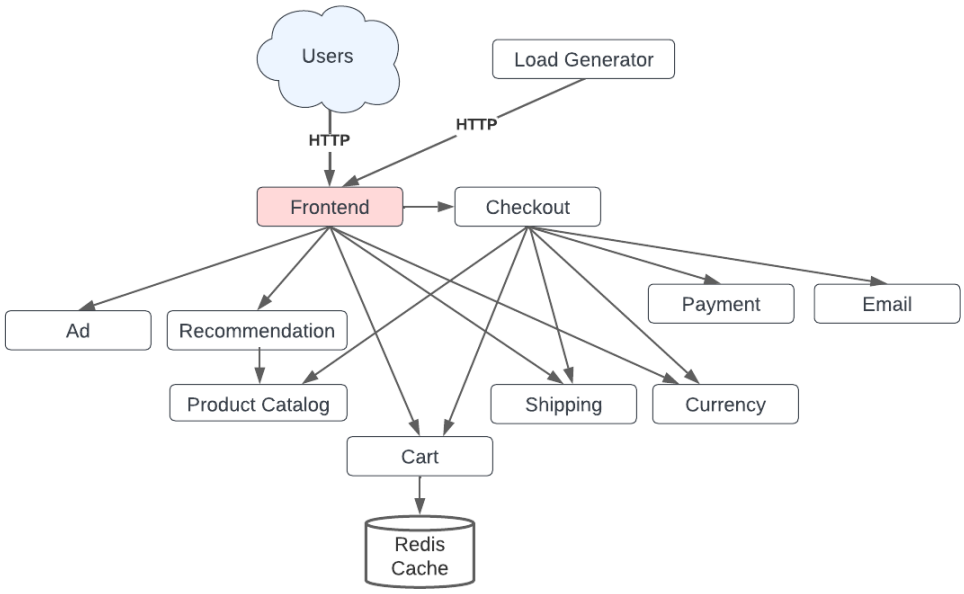

Figure 4.1: Online Boutique microservices layout. This visualization showcases the layout and interconnections of various microservices in the Online Boutique application. Each microservice plays a distinct role in ensuring the seamless functioning of the entire application.

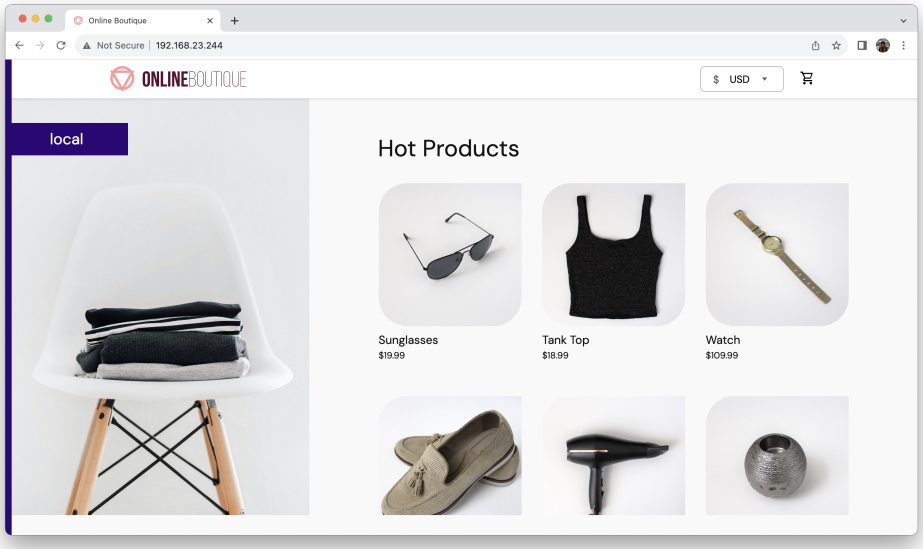

Figure 4.2: Online Boutique application home page.

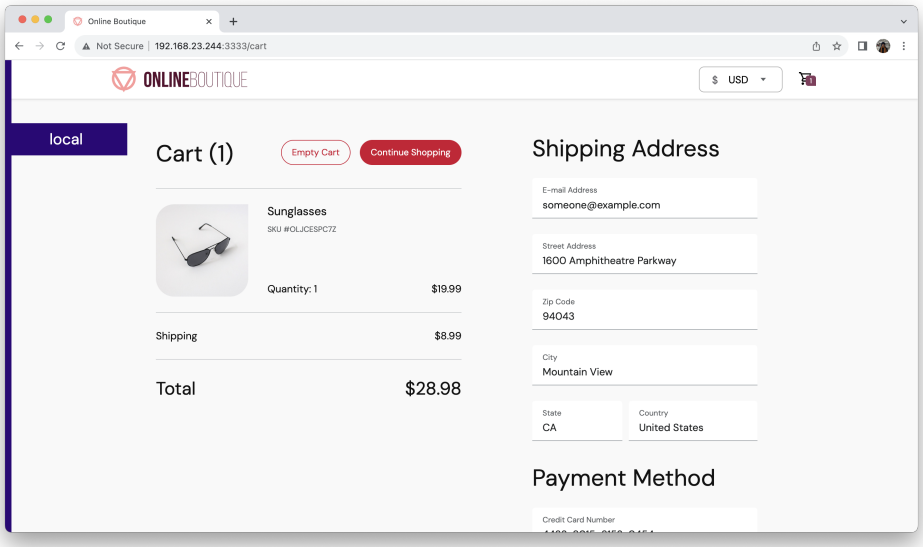

Figure 4.3: Online Boutique application checkout page.

Table 4.1: Description of the Online Boutique application microservices

Microservice Name	Image Description
shipping service	Gives shipping cost estimates based on the shopping cart. Ships items to the given address (mock).
checkout service	Retrieves user cart, prepares order and orchestrates the payment, shipping, and the email notification.
product catalog service	Provides the list of products from a JSON file and ability to search products and get individual products.
redis-cart	redis:alpine
load generator	Continuously sends requests imitating realistic user shopping flows to the frontend.
recommendation service	Recommends other products based on what's given in the cart.
email service	Sends users an order confirmation email (mock).
payment service	Charges the given credit card info (mock) with the given amount and returns a transaction ID.
currency service	Converts one money amount to another currency. Uses real values fetched from the European Central Bank.
cart service	Stores the items in the user's shopping cart in Redis and retrieves it.
ad service	Provides text ads based on given context words.
frontend	Exposes an HTTP server to serve the website. Does not require signup/login; generates session IDs for all users automatically.

In this work, we have identified the frontend microservice as the most critical, and have thus applied software multiversioning to it. Past studies [48, 86, 87] have utilized the PageRank algorithm to assess the importance of microservices within a system. While an in-depth exploration of this methodology falls outside the purview of our current research, interested readers are encouraged to refer to these cited works for further insight. Our designation of the frontend service as the critical microservice is predicated on the following considerations:

- **Initial User Interaction:** The frontend service represents the initial inter-

face for users. The entire user experience is disrupted if it fails, regardless of whether other services function perfectly.

- **Aggregator of Services:** The frontend often acts as aggregators, pulling data from various services to present to the user. Its failure can render these services effectively inaccessible.

As outlined in Section 3.1, our system consists of two pivotal microservices: Load Balancer and Scaling Engine. These are instrumental to the operation of the Online Boutique application, which we use as a case study. Table 4.2 provides a detailed description of the microservices in our system.

4.3 Scaling Engine Configuration

Within our system, a component for our scaling engine named 'ReplicaBalancer' is created using the command described in Listing 3.1. Following its instantiation, we configure the environment variable `DEPLOYMENT_IMAGES_REPLICAS` as shown in the command outlined in Listing 4.1. Upon configuring the environment variable, it is crucial to verify that all Pods have successfully transitioned to a 'running' state. This verification serves as a prerequisite before commencing the experimental procedures. Once all Pods are confirmed to be running, we proceed

with initiating the experiment.

Listing 4.1: Setting environmental variables

```
kubectl set env deployment/test \
DEPLOYMENT_IMAGES_REPLICAS = \
"gcr.io/google-samples/microservices-demo/frontend:v0.8.0, \
gcr.io/google-samples/microservices-demo/frontend:v0.8.0, \
gcr.io/google-samples/microservices-demo/frontend:v0.8.0"
```

4.4 Workload

For the load generation, we used Locust running a specific set of workloads. In this workload, there is a predefined list for products. A user can browse random products, set currency preferences from four choices (EUR, USD, JPY, CAD), view their cart, add products to the cart, and proceed to checkout. These tasks simulate typical online shopping patterns. When checking out, the user is assumed to add a random product to their cart and provide fixed checkout details. The user's behaviour is defined in a task set with weighted probabilities for each task, ensuring that browsing products is the most frequent action. Each simulated user will begin their session by visiting the homepage and then proceed with actions

Table 4.2: Comprehensive list of microservices used in the study, with **bold** indicating custom-developed services pivotal to the research's innovative approach.

Microservice Name	Image Name	Description
shipping service	shippingservice:v0.8.0	Standard Online Boutique microservice
checkout service	checkoutservice:v0.8.0	Standard Online Boutique microservice
product catalog service	productcatalogservice:v0.8.0	Standard Online Boutique microservice
redis-cart	redis:alpine	Standard online boutique microservice
load generator	nazaninakhtarian/locust-loadtest:latest	Our deployed microservice
recommendation service	recommendationservice:v0.8.0	standard Online Boutique microservice
email service	emailservice:v0.8.0	Standard Online Boutique microservice
payment service	paymentservice:v0.8.0	Standard Online Boutique microservice
currency service	currencyservice:v0.8.0	Standard Online Boutique microservice
cart service	cartservice:v0.8.0	Standard Online Boutique microservice
ad service	adservice:v0.8.0	Standard Online Boutique microservice
frontend-memory-leak-deployment	frontend:v0.8.0	Standard Online Boutique microservice
frontend-inconsistent-response-deployment	frontend:v0.8.0	Standard Online Boutique microservice
frontend-faulty-deployment	frontend:v0.8.0	Standard Online Boutique microservice
scaling engine	nazaninakhtarian/rsapp:testing	Our deployed microservice
nginx-deployment	nginx:latest	Standard NGINX microservice
prometheus-grafana	quay.io/kiwigrid/k8s-sidecar:1.25.1	Dashboard interface for Grafana
chaos-dashboard	ghcr.io/chaos-mesh/chaos-coredns:v0.2.6	Dashboard interface for Chaos Mesh

based on the weighted tasks. In the original configuration of the online boutique load generator, users pause between tasks for an interval ranging from 1 to 10 seconds. In our study, we adapted this behaviour to a shorter, more dynamic range of 100 to 1000 milliseconds. This alteration was made with the intent of placing a more concentrated load on the frontend microservice, allowing us to better observe and understand the system's overall response and behaviour under such conditions.

4.5 Chaos Mesh

Chaos Mesh[3] is an open-source Chaos Engineering platform designed to orchestrate chaos within Kubernetes environments. Chaos Engineering is a practice focused on identifying vulnerabilities in systems by deliberately introducing failures and observing the system's reaction. Chaos Mesh also offers a user-friendly UI, enabling the creation, management, and monitoring of chaos experiments. The platform supports a range of chaos experiments to test various aspects of Kubernetes deployments, including:

1. **Network Chaos:** Simulates network delays, packet loss, and other network

[3]https://chaos-mesh.org

issues between Pods to evaluate application behaviour under adverse network conditions.

2. **Kernel Chaos:** Triggers kernel panics, hangs, and faults in system calls, providing insights into application resilience at the operating system level.

3. **Pod Chaos:** Allows for the killing, restarting, or pausing of specific Pods, enabling the simulation of failure scenarios involving Pod disruptions.

4. **HTTP Chaos:** Alters HTTP requests and responses, including status codes, headers, and body content, to assess the impact on Pod communication.

5. **DNS Chaos:** Introduces DNS errors or alters DNS resolution to examine application responses to DNS issues.

6. **Stress Chaos:** Applies stress to CPU and memory resources within containers, helping identify memory leaks and system behaviour under resource strain.

7. **Workflow Chaos:** Facilitates the creation of complex chaos scenarios by arranging experiments in sequences or in parallel.

8. **Time Chaos:** Simulates time offsets to challenge synchronization and timing-dependent functionalities.

9. **IO Chaos:** Emulates file system faults, aiding in understanding system responses to I/O errors.

For our experiments, we employed Pod Chaos, HTTP Chaos, and Stress Chaos to investigate the resilience of our system. The subsequent section will delve into the specifics of integrating these chaos types into our experiments.

4.6 Chaos Injection

In this project, we introduced chaos into three distinct versions of the frontend microservice, each labeled as 'Faulty,' 'Inconsistent Response,' and 'Memory Leak'. These labels clearly indicate the type of chaos associated with each version:

1. **Faulty Version:** This version experiences crashes resulting in Pod restarts due to a deliberate bug.

2. **Inconsistent Response Version:** This version displays unpredictable response times.

3. **Memory Leak Version:** This version is affected by a memory leak, testing the system's handling of memory-related issues.

4.6.1 Chaos Types

As previously mentioned, we introduce three types of chaos into our system:

- **Pod Chaos:** Implemented to trigger every 3 minutes, this chaos specifically affects the 'frontend-faulty-deployment' application, causing all associated Pods to terminate for 30 seconds. This simulates the effect of sudden Pod crashes.

- **HTTP Chaos:** Scheduled to activate every 4 minutes on the 'frontend-inconsistent-response-deployment' application, this chaos introduces a 2-second delay to all HTTP responses, lasting for 2 minutes, to mimic network latency.

- **Stress Chaos:** This is applied to the 'frontend-memory-leak-deployment' application at 4-minute intervals, simulating a memory leak by rapidly consuming memory resources. Two worker processes each use 20 MB of memory without release, replicating a memory leak scenario for 2 minutes.

4.7 Monitoring Metrics with Prometheus

We leveraged the Prometheus Query Language (PromQL) for querying metrics. Prometheus[4] is an open-source system for monitoring and alerting, renowned for its robustness and scalability. For targeted insights, we tailored the 'Pod' regex within each query to correspond to the specific frontend deployment under test, allowing us to isolate and monitor the effects of chaos on each version. Below we present the PromQL queries utilized in our study:

Listing 4.2: Query to get the total restart count for the 'frontend-faulty' deployment.

```
sum(kube_pod_container_status_restarts_total{namespace="default",
    pod=~"frontend-faulty.*"} * on(pod) group_left
    kube_pod_status_phase{namespace="default", phase="Running"})
```

Listing 4.3: Query to get the average memory usage (MB) of 'frontend-faulty' Pods.

```
sum(container_memory_usage_bytes{namespace='default',
    pod=~'frontend-faulty.*'} * on (pod, namespace)
    group_left(phase) kube_pod_status_phase{phase='Running'}) /
    count(kube_pod_status_phase{namespace='default',
    pod=~'frontend-faulty.*', phase='Running'}) / 1024 / 1024)
```

[4]https://prometheus.io/

4.8 Experimental Discussion

To comprehend the system's behaviour in depth, we conducted two experiments. The first aimed to observe the change in replica distribution based on the reliability of individual components. By selectively introducing chaos, we gauged the direct influence of reliability metrics on replica count. The second experiment focused on the system's response to varying workloads, particularly its adaptability in replica scaling with changes in CPU utilization.

4.8.1 Experiment 1: Evolution under Constant Workload

Our first experiment investigated the system's behaviour under constant workload conditions. We conducted tests for 2 hours with 20 concurrent users. For clarity and to observe the direct influence of each metric on reliability, we first injected only one type of chaos into each software version. In this way, we isolated the effects of each chaos type, making it easier to understand the specific impact of each metric on the system's overall reliability. After that, we injected all three types of chaos into the system and observed its behaviour again.

Initially, we allowed the system to run undisturbed to observe the replica distribution across different versions. As shown in Figure 4.6, each version started

with 5 replicas, given a total of 15 replicas distributed equally. This uniform distribution can be attributed to the identical reliability scores of the versions, as no chaos had been introduced at this point.

Subsequently, we introduced Pod chaos, detailed in Section 4.6. Following this, a noticeable restart increase for the frontend-faulty-deployment was evident, as depicted in Figure 4.4. Additionally, as seen in Figure 4.5, memory usage patterns for this version fluctuated. Simultaneously, Figure 4.6 captures the system's adaptive response in terms of replica distribution. The faulty deployment's replica count was adjusted to 3. This change underscores the system's recognition of the diminished reliability of the faulty deployment due to the Pod chaos, which was an expected outcome. It's crucial to note that, at this juncture, only the Pod chaos was in play.

Stopping the chaos which replicating a scenario where a developer fixes a bug, the system gradually returned to an even distribution of 5 replicas per version.

Next, we applied HTTP chaos targeting the frontend-inconsistent-response-deployment. This resulted in noticeable system latency, as captured in Figure 4.7. Interestingly, a replica was reallocated from frontend-inconsistent-response to frontend-faulty-deployment. Upon halting this chaos, the system returned to its balanced state of 5 replicas for each version.

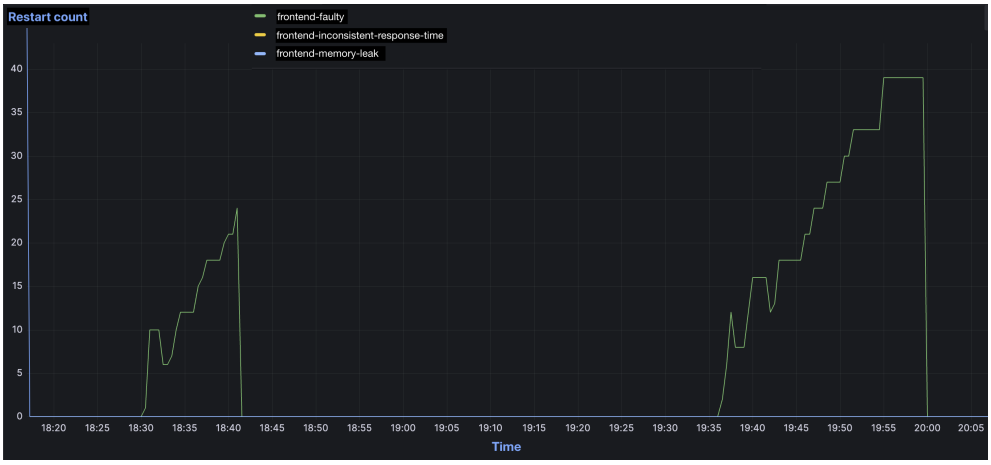

Figure 4.4: Restart count of frontend microservice versions. This chart illustrates the frequency and patterns of system restarts over a specific period.

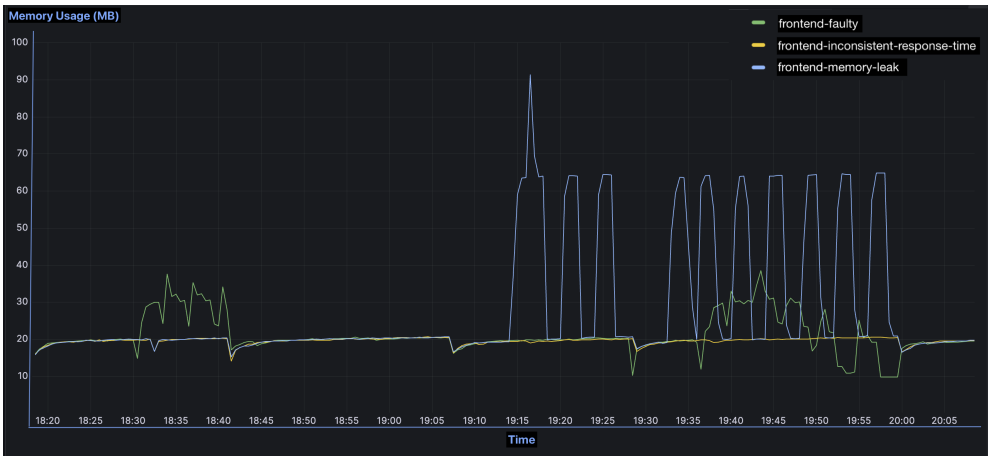

Figure 4.5: Memory usage over time (measured in MB). This graph provides a comprehensive look at the memory consumption patterns for different frontend deployments.

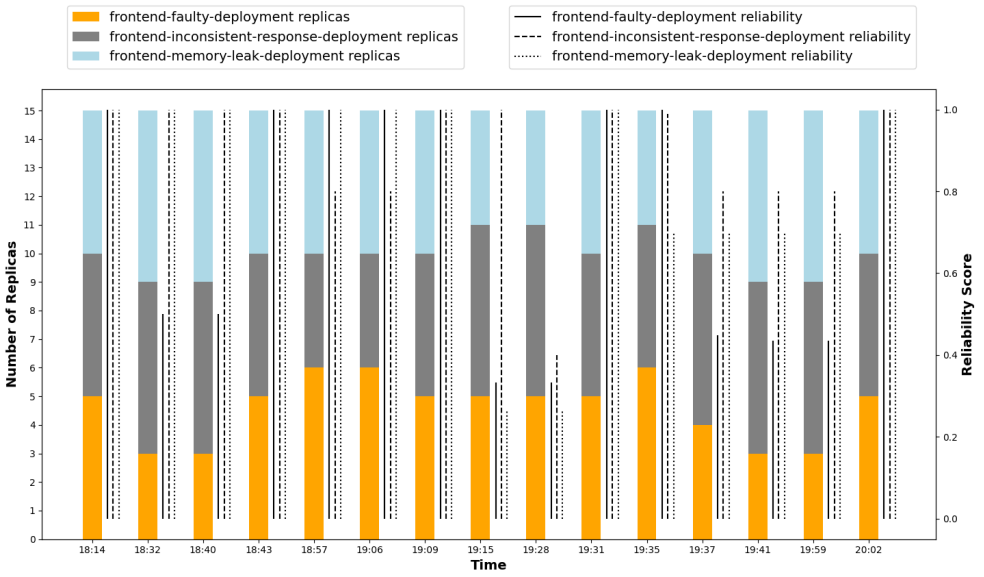

Figure 4.6: Replica and reliability over time. The bar chart illustrates the number of replicas for different frontend microservice versions over time. Adjacent vertical lines, differentiated by line style, represent the reliability scores for each version.

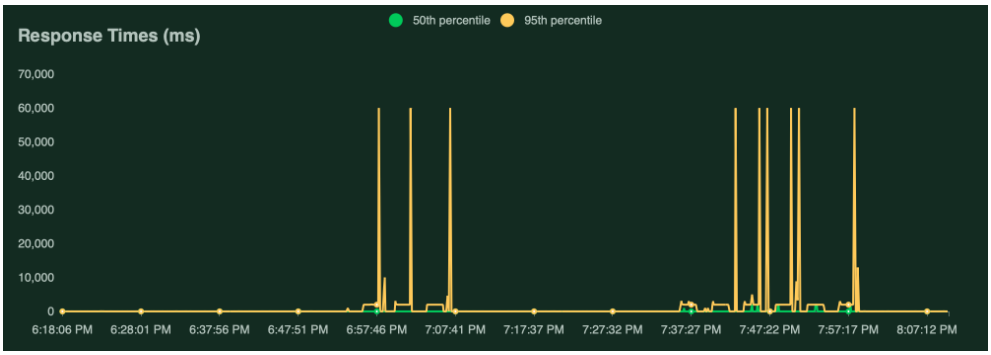

Figure 4.7: Application's response time during the first experiment.

Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/	1295	7	273	1	60072	10281	0.2	0.0
GET	/cart	3744	17	187	1	60072	14411	0.6	0.0
POST	/cart	3661	18	245	1	60077	18148	0.5	0.0
POST	/cart/checkout	1246	13	19	1	1043	6520	0.2	0.0
GET	/product/0PUK6V6EV0	1718	4	123	1	9159	7741	0.3	0.0
GET	/product/1YMWWN1N4O	1761	12	212	1	60012	7730	0.3	0.0
GET	/product/2ZYFJ3GM2N	1706	13	148	1	23381	7717	0.3	0.0
GET	/product/66VCHSJNUP	1745	8	111	1	5044	7713	0.3	0.0
GET	/product/6E92ZMYYFZ	1818	6	154	1	60009	7709	0.3	0.0
GET	/product/9SIQT8TOJO	1776	16	228	1	60013	7715	0.3	0.0
GET	/product/L9ECAV7KIM	1796	6	180	1	60010	7728	0.3	0.0
GET	/product/LS4PSXUNUM	1770	8	163	1	60013	7721	0.3	0.0
GET	/product/OLJCESPC7Z	1775	12	283	1	60013	7737	0.3	0.0
POST	/setCurrency	2496	19	205	1	60019	10259	0.4	0.0
	Aggregated	**28307**	**159**	**188**	**1**	**60077**	**10244**	**4.2**	**0.0**

Figure 4.8: Request statistics overview. This chart presents a detailed breakdown of request metrics, including successful requests, failures, and other pertinent statistics over a specified period.

Figure 4.9: Locust request analysis. This chart showcases the load test results using Locust, detailing request success rates and failures.

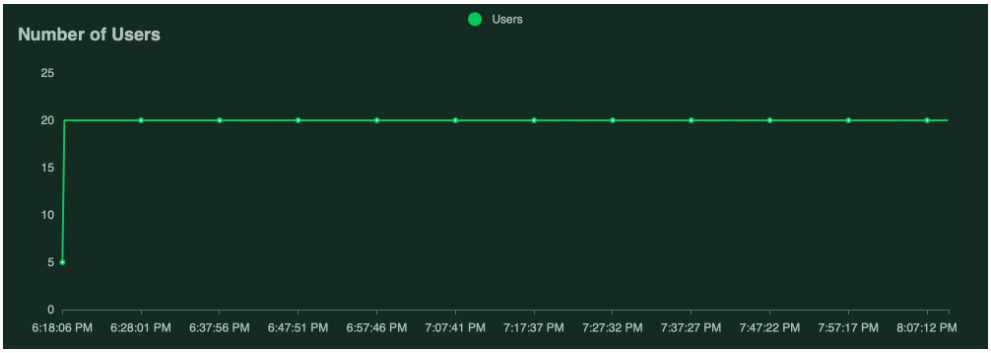

Figure 4.10: Number of users. This chart presents the number of active users accessing the system over a specific duration for the first experiment.

In a subsequent test, we introduced stress chaos to the frontend-memory-leak-deployment. The Grafana[5] dashboard, as seen in Figure 4.5, captured a rise in memory consumption. The result on replica distribution is illustrated in Figure 4.6, which involves the subtraction of a replica from frontend-memory-leak and its addition to frontend-inconsistent-response. Following 10 minutes after stopping this chaos, the system restored its equilibrium of 5 replicas per version.

In our final testing phase, we combined all three chaos types to understand their compounded effect. The replica distribution settled at 3, 6, and 6, influenced by the metric weights detailed in Section 3.2.2. After stopping all chaos injections, the system eventually returned to its initial balanced state.

Figure 4.8 shows the request statistics of our workload. Figure 4.9 shows the sent request pattern to the system by the load generator microservice. Lastly,

[5]https://grafana.com

Figure 4.10, shows the number of users that was 20 in the whole experiment.

4.8.2 Experiment 2: Dynamic Scaling based on Workload

Our second experiment was designed to understand the system's dynamic scaling capabilities in relation to variable workloads, primarily focusing on CPU utilization as an indicator.

As the system was subjected to different user loads (Figure 4.11), we closely monitored the frontend microservice pods' CPU usage as illustrated in Figure 4.13. Over time, as we changed the number of users, a direct correlation was observed between the workload and CPU usage. As the workload intensified, there was a consequent increase in CPU usage. In line with the user-configurable thresholds discussed in Section 3.5, our configuration set an upper CPU limit at 60% and a lower limit at 20%. Should the CPU usage exceed 60%, the system would trigger an upscale in the number of pod replicas. Initially, with a total of 9 replicas and CPU usage below 20%, the system demonstrated its efficiency by reducing the replica count, as evident in the initial segment of the Figure 4.14. However, as the experiment progressed and the number of users increased, a notable increase in the total number of replicas was recorded, demonstrating the system's ability in dynamic scaling to meet the demands of a fluctuating workload.

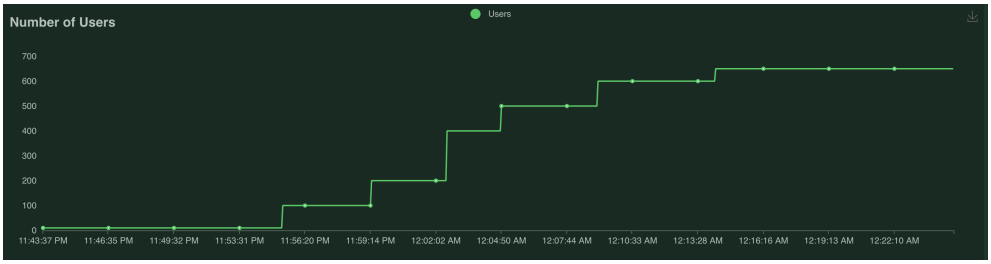

Figure 4.11: Number of users. This chart presents the number of active users accessing the system over a specific duration for the second experiment.

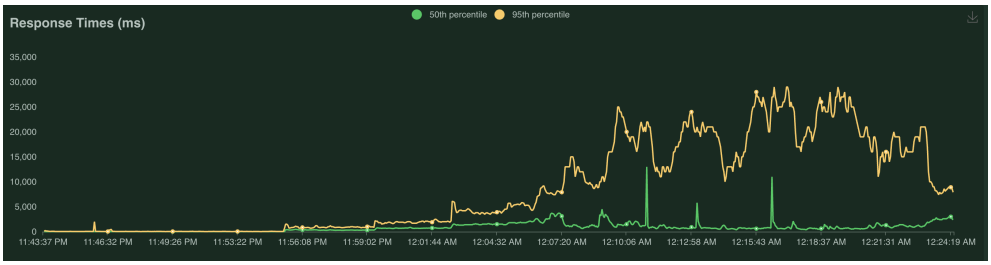

Figure 4.12: Application's response time during the second experiment.

Figure 4.13: Average CPU utilization of frontend microservice Pods over time.

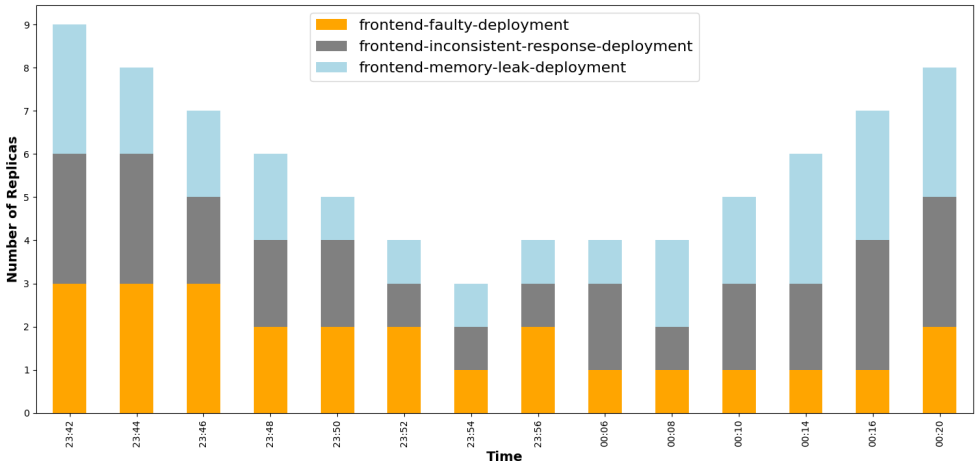

Figure 4.14: Dynamic scaling of frontend microservice Pods. This chart visualizes the system's dynamic scaling capabilities in response to varying workloads, highlighting the changes in the number of replicas over time. The system's adaptability to workload fluctuations is evident from the shifts in replica counts.

Chapter 5

Discussions and Future Works

5.1 Threats to Validity

In this section, we discuss the potential threats to the validity of our experiment involving Chaos Mesh to test the reliability and robustness of our subject system.

5.1.1 External Validity

Choice of Subject System: Our experiments were conducted on the Online Boutique application within a specific system configuration. Future research should explore the generalizability of our findings across diverse setups and varying cluster sizes.

Chaos Types Selection: The chaos experiments were limited to certain types provided by Chaos Mesh. Real-world systems may encounter a broader and more complex range of disruptions not encompassed by our study.

5.1.2 Internal Validity

Chaos Injection Timing: The frequency of chaos injections in our tests may not mirror actual operational conditions, where failures could be more erratic or frequent.

Replica Distribution: We initiated our experiments with uniform replica distribution across software versions, which may not accurately reflect the varied distributions present in live environments.

5.1.3 Construct Validity

Metric Selection: We chose specific metrics to represent system reliability. While informative, these metrics may not translate universally to all systems, which could have different reliability benchmarks or operational criteria.

Metric Weighting: The weights given to each metric in Section 3.2.2 are context-dependent. In different scenarios, the prioritization of these metrics could vary significantly.

5.2 Future Work

Extension to More Microservices: Given the scale and complexity of software systems, future work could expand to include a wider range of microservices.

Refinement of Metrics and Validation: While the current study relies on specific metrics like restart count, response time, and memory usage, there's room to investigate other metrics that might offer more comprehensive insights. This might also include a validation process for metrics selection across varied systems.

Analysis of Real-world Traffic Patterns: Incorporating actual user traffic patterns could provide a more accurate assessment of system behaviour under typical operational conditions.

Advanced Chaos Experimentation: Enriching chaos testing scenarios with more diverse real-world failures could enhance the thoroughness of system evaluations.

Dynamic Scaling Techniques: There is an opportunity to improve upon the threshold-based scaling approach by integrating predictive models that enable anticipatory scaling actions.

Multi-modal Metric Integration: Future work could also include a mix of performance metrics, such as network bandwidth, disk I/O, and tailored application metrics for a comprehensive performance and scalability analysis.

Chapter 6

Conclusions

In the realm of software engineering, multi-versioning has predominantly been employed to enhance system robustness, especially for critical applications. However, due to the substantial costs associated with implementing multiple versions of an application, its use has often been restricted to the application's critical components and microservices architecture presents a straightforward way to leverage software multi-versioning. In this work, we dynamically adjusted the replica count for each software version in response to the system's real-time reliability metrics. By leveraging Chaos Mesh, we simulated a series of disruptions and gained insights into how different metrics influence overall system reliability. With the rectification of bugs and the improvement of versions, we noted an

increase in their respective population. By the end of the experiment, we anticipated a similar number of replicas for each version if all of them were bug-free and efficient, as was illustrated in Figure 4.6. Our results contribute to ongoing research and highlight the potential of dynamic adaptation as an instrumental tactic for developing reliable software systems in the future.

Bibliography

[1] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: migration to a cloud-native architecture," *IEEE Software*, vol. 33, pp. 42–52, 2016.

[2] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling web applications in heterogeneous cloud infrastructures," in *2014 IEEE International Conference on Cloud Engineering*, 2014, pp. 195–204.

[3] N. C. Coulson, S. Sotiriadis, and N. Bessis, "Adaptive microservice scaling for elastic applications," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4195–4202, 2020.

[4] D.-D. Vu, M.-N. Tran, and Y. Kim, "Predictive hybrid autoscaling for containerized applications," *IEEE Access*, vol. 10, pp. 109 768–109 778, 2022.

[5] N.-M. Dang-Quang and M. Yoo, "Deep learning-based autoscaling using bidirectional long short-term memory for kubernetes," *Applied Sciences*, vol. 11, no. 9, p. 3835, 2021.

[6] S. Taherizadeh, V. Stankovski, and J.-H. Cho, "Dynamic multi-level autoscaling rules for containerized applications," *The Computer Journal*, vol. 62, no. 2, pp. 174–197, 2019.

[7] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with elasticdocker," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, 2017, pp. 472–479.

[8] "Horizontal pod autoscaler," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale, accessed: 2020-08-01.

[9] I. Prachitmutita, W. Aittinonmongkol, N. Pojjanasuksakul, M. Supattatham, and P. Padungweang, "Auto-scaling microservices on iaas under sla with cost-effective framework," in *2018 Tenth International Conference on Advanced Computational Intelligence (ICACI)*, 2018, pp. 583–588.

[10] M. Imdoukh, I. Ahmad, and M. G. Alfailakawi, "Machine learning-based auto-scaling for containerized applications," *Neural Computing and Applications*, vol. 32, pp. 9745–9760, 2020.

[11] X. Tang, Q. Liu, Y. Dong, J. Han, and Z. Zhang, "Fisher: An efficient container load prediction model with deep neural network in clouds," in *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*. IEEE, 2018, pp. 199–206.

[12] H. Zhang, G. Jiang, K. Yoshihira, H. Chen, and A. Saxena, "Intelligent workload factoring for a hybrid cloud computing model," in *2009 Congress on Services - I*, 2009, pp. 701–708.

[13] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload prediction using arima model and its impact on cloud applications' qos," *IEEE transactions on cloud computing*, vol. 3, no. 4, pp. 449–458, 2014.

[14] W. Fang, Z. Lu, J. Wu, and Z. Cao, "Rpps: A novel resource prediction and provisioning scheme in cloud data center," in *2012 IEEE Ninth International Conference on Services Computing*. IEEE, 2012, pp. 609–616.

[15] H. T. Ciptaningtyas, B. J. Santoso, and M. F. Razi, "Resource elasticity controller for docker-based web applications," in *2017 11th International Conference on Information Communication Technology and System (ICTS)*, 2017, pp. 193–196.

[16] V. Messias, J. Estrella, R. Ehlers, M. Santana, R. Santana, and S. Reiff-Marganiec, "Combining time series prediction models using genetic algorithm to autoscaling web applications hosted in the cloud infrastructure," *Neural Computing and Applications*, vol. 27, 11 2016.

[17] M. Elgili, "Load balancing algorithms round-robin (rr), least-connection and least loaded algorithm," *ResearchGate*, 2020. [Online]. Available: https://www.researchgate.net/publication/link_to_paper

[18] T. W. Harjanti, H. Setiyani, and J. Trianto, "Load balancing analysis using round-robin and least-connection algorithms for server service response time," *Applied Technology and Computing Science Journal*, vol. 5, no. 2, pp. 40–49, 2022.

[19] R. L. Keeney and H. Raiffa, *Decisions with multiple objectives: preferences and value trade-offs*. Cambridge university press, 1993.

[20] S. Rasmussen, *Utility Function*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 5–17. [Online]. Available: https://doi.org/10.1007/978-3-642-21686-2_3

[21] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 276–291.

[22] M. Franz, "E unibus pluram: Massive-scale software diversity as a defense mechanism," in *Proceedings of the 2010 New Security Paradigms Workshop*, ser. NSPW '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 7–16. [Online]. Available: https://doi.org/10.1145/1900546.1900550

[23] B. Persaud, B. Obada, N. Mansourzadeh, A. Moni, and A. Somayaji, "Frankenssl: Recombining cryptographic libraries for software diversity," 06 2016.

[24] C. Çiğşar and Y. Lim, "Modeling and analysis of cluster of failures in redundant systems," in *2017 2nd International Conference on System Reliability and Safety (ICSRS)*, 2017, pp. 119–124.

[25] E. Gracic, A. Hayek, and J. Börcsök, "Evaluation of fpga design tools for safety systems with on-chip redundancy referring to the standard iec 61508," in *2017 2nd International Conference on System Reliability and Safety (ICSRS)*, 2017, pp. 386–390.

[26] A. Gorbenko, V. Kharchenko, and A. Romanovsky, *Using Inherent Service Redundancy and Diversity to Ensure Web Services Dependability*, 03 2009, vol. 5454, pp. 324–341.

[27] H. Borck, M. Boddy, I. J. De Silva, S. Harp, K. Hoyme, S. Johnston, A. Schwerdfeger, and M. Southern, "Frankencode: Creating diverse programs using code clones," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 604–608.

[28] L. Wang, "Architecture-based reliability-sensitive criticality measure for fault-tolerance cloud applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 11, pp. 2408–2421, 2019.

[29] Z. Zheng, M. Lyu, and H. Wang, "Service fault tolerance for highly reliable service-oriented systems: an overview," *Science China Information Sciences*, vol. 58, pp. 1–12, 05 2015.

[30] Z. Zheng and M. R. Lyu, "Selecting an optimal fault tolerance strategy for reliable service-oriented systems with local and global constraints," *IEEE Transactions on Computers*, vol. 64, no. 1, pp. 219–232, 2015.

[31] S. Gholami, A. Goli, C.-P. Bezemer, and H. Khazaei, "A framework for satisfying the performance requirements of containerized software systems through multi-versioning," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 150–160. [Online]. Available: https://doi.org/10.1145/3358960.3379125

[32] H. Mohamed and O. F. El-Gayar, "End-to-end latency prediction of microservices workflow on kubernetes: a comparative evaluation of machine learning models and resource metrics," *Proceedings of the Annual Hawaii International Conference on System Sciences*, 2021.

[33] V. H. S. C. Pinto, R. R. Oliveira, R. F. Vilela, and S. R. Souza, "Evaluating the user acceptance testing for multi-tenant cloud applications." in *CLOSER*, 2018, pp. 47–56.

[34] T. D. Timur, I. K. E. Purnama, and S. M. S. Nugroho, "Deploying scalable face recognition pipeline using distributed microservices," in *2019 Interna-*

tional Conference on Computer Engineering, Network, and Intelligent Multimedia (CENIM), 2019, pp. 1–5.

[35] W. Lu, Q. Xu, C. Lan, L. Lyu, Y. Zhou, Q. Shi, Y. Zhao *et al.*, "Microservice-based platform for space situational awareness data analytics," *International Journal of Aerospace Engineering*, vol. 2020, 2020.

[36] S. P. R. Asaithambi, R. Venkatraman, and S. Venkatraman, "Mobda: Microservice-oriented big data architecture for smart city transport systems," *Big Data and Cognitive Computing*, vol. 4, no. 3, 2020. [Online]. Available: https://www.mdpi.com/2504-2289/4/3/17

[37] S. Ali, M. A. Jarwar, and I. Chong, "Design methodology of microservices to support predictive analytics for iot applications," *Sensors*, vol. 18, no. 12, p. 4226, 2018.

[38] M. Abdel-Basset, H. Hawash, and K. Sallam, "Federated threat-hunting approach for microservice-based industrial cyber-physical system," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 3, pp. 1905–1917, 2022.

[39] A. Alsaedi, N. Moustafa, Z. Tari, A. Mahmood, and A. Anwar, "Ton$_i$$ottelemetrydataset$: $Anewgenerationdatasetofiotandiiotfordata-drivenintrusiondetectionsystems,''$ *IEEE Access*, $vol.$ $8, pp.165130--165150, 2020.$

[40] R. Damaševičius, A. Venčkauskas, S. Grigaliunas, J. Toldinas, N. Morkevicius, T. Aleliūnas, and P. Smuikys, "Litnet-2020: An annotated real-world network flow dataset for network intrusion detection," *Electronics*, vol. 9, p. 800, 05 2020.

[41] S. Trilles, A. González-Pérez, and J. Huerta, "An iot platform based on microservices and serverless paradigms for smart farming purposes," *Sensors*, vol. 20, no. 8, p. 2418, 2020.

[42] R. Xu, G. S. Ramachandran, Y. Chen, and B. Krishnamachari, "Blendsm-ddm: blockchain-enabled secure microservices for decentralized data marketplaces," *2019 IEEE International Smart Cities Conference (ISC2)*, 2019.

[43] V. De Maio and D. Kimovski, "Multi-objective scheduling of extreme data scientific workflows in fog," *Future Generation Computer Systems*, vol. 106, pp. 171–184, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19309197

[44] C. Li, M. Song, M. Zhang, and Y. Luo, "Effective replica management for improving reliability and availability in edge-cloud computing environment," *Journal of Parallel and Distributed Computing*, vol. 143, pp. 107–128, 2020.

[45] R. Florin, A. Ghazizadeh, P. Ghazizadeh, S. Olariu, and D. C. Marinescu, "Enhancing reliability and availability through redundancy in vehicular clouds," *IEEE Transactions on Cloud Computing*, vol. 9, no. 3, pp. 1061–1074, 2019.

[46] H. Chen, X. Zhu, G. Liu, and W. Pedrycz, "Uncertainty-aware online scheduling for real-time workflows in cloud service environment," *IEEE Transactions on Services Computing*, vol. 14, no. 4, pp. 1167–1178, 2018.

[47] N. Kherraf, S. Sharafeddine, C. M. Assi, and A. Ghrayeb, "Latency and reliability-aware workload assignment in iot networks with mobile edge clouds," *IEEE Transactions on Network and Service Management*, vol. 16, no. 4, pp. 1435–1449, 2019.

[48] Z. Liu, G. Fan, H. Yu, and L. Chen, "An approach to modeling and analyzing reliability for microservice-oriented cloud applications," *Wireless Communications and Mobile Computing*, vol. 2021, pp. 1–17, 08 2021.

[49] ——, "Modelling and analysing the reliability for microservice-based cloud application based on predicate petri net," *Expert Systems*, vol. 39, no. 6, p. e12924, 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/exsy.12924

[50] W. Ha, "Reliability prediction for web service composition," in *2017 13th International Conference on Computational Intelligence and Security (CIS)*. IEEE, 2017, pp. 570–573.

[51] Z. Zang, Q. Wen, and K. Xu, "A fault tree based microservice reliability evaluation model," in *IOP Conference Series: Materials Science and Engineering*, vol. 569, no. 3. IOP Publishing, 2019, p. 032069.

[52] H. Gao, C. Liu, Y. Li, and X. Yang, "V2vr: reliable hybrid-network-oriented v2v data transmission and routing considering rsus and connectivity probability," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 6, pp. 3533–3546, 2020.

[53] A. Sharif, M. Nickray, and A. Shahidinejad, "Energy-efficient fault-tolerant scheduling in a fog-based smart monitoring application," *International Journal of Ad Hoc and Ubiquitous Computing*, vol. 36, no. 1, pp. 32–49, 2021.

[54] J. Yao, Q. Lu, H.-A. Jacobsen, and H. Guan, "Robust multi-resource allocation with demand uncertainties in cloud scheduler," in *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2017, pp. 34–43.

[55] B. K. Ray, A. Saha, S. Khatua, and S. Roy, "Proactive fault-tolerance technique to enhance reliability of cloud service in cloud federation environment," *IEEE Transactions on Cloud Computing*, vol. 10, no. 2, pp. 957–971, 2020.

[56] G. Fan, L. Chen, H. Yu, and D. Liu, "Modeling and analyzing dynamic fault-tolerant strategy for deadline constrained task scheduling in cloud computing," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 50, no. 4, pp. 1260–1274, 2017.

[57] T. Shi, H. Ma, and G. Chen, "A genetic-based approach to location-aware cloud service brokering in multi-cloud environment," in *2019 IEEE International Conference on Services Computing (SCC)*. IEEE, 2019, pp. 146–153.

[58] L. Wang, Q. He, D. Gao, J. Wan, and Y. Zhang, "Temporal-perturbation aware reliability sensitivity measurement for adaptive cloud service selection," *IEEE Transactions on Services Computing*, vol. 15, no. 4, pp. 2301–2313, 2020.

[59] R. Pietrantuono, S. Russo, and A. Guerriero, "Run-time reliability estimation of microservice architectures," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, 2018, pp. 25–35.

[60] W. Hasselbring and G. Steinacker, "Microservice architectures for scalability, agility and reliability in e-commerce," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 243–246.

[61] T. Wang, W. Zhang, J. Xu, and Z. Gu, "Workflow-aware automatic fault diagnosis for microservice-based applications with statistics," *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2350–2363, 2020.

[62] W. Yang, L. CHENG, and S. Xin, "Design and research of microservice application automation testing framework," in *2019 International Conference on Information Technology and Computer Application (ITCA)*. IEEE, 2019, pp. 257–260.

[63] M. Camilli, A. Guerriero, A. Janes, B. Russo, and S. Russo, "Microservices integrated performance and reliability testing," in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, ser. AST '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 29–39. [Online]. Available: https://doi.org/10.1145/3524481.3527233

[64] J. Yao and N. Ansari, "Fog resource provisioning in reliability-aware iot networks," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8262–8269, 2019.

[65] R. K. Behera, K. H. K. Reddy, and D. S. Roy, "Reliability modelling of service oriented internet of things," in *2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)(Trends and Future Directions)*. IEEE, 2015, pp. 1–6.

[66] S. Sinche, O. Polo, D. Raposo, M. Femandes, F. Boavida, A. Rodrigues, V. Pereira, and J. S. Silva, "Assessing redundancy models for iot reliability," in *2018 IEEE 19th International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*. IEEE, 2018, pp. 14–15.

[67] L. Li, Z. Jin, G. Li, L. Zheng, and Q. Wei, "Modeling and analyzing the reliability and cost of service composition in the iot: A probabilistic approach," in *2012 IEEE 19th International Conference on Web Services*. IEEE, 2012, pp. 584–591.

[68] D. Ursino and L. Virgili, "Humanizing iot: Defining the profile and the reliability of a thing in a multi-iot scenario," *Toward Social Internet of Things (SIoT): Enabling Technologies, Architectures and Applications: Emerging Technologies for Connected and Smart Social Objects*, pp. 51–76, 2020.

[69] I. Eroshkin, L. Vojtech, and M. Neruda, "Resource efficient real-time reliability model for multi-agent iot systems," *IEEE Access*, vol. 10, pp. 2578–2590, 2022.

[70] G. Araújo, A. Sabino, L. Lima, V. Costa, C. Brito, P. Rego, I. Fé, and F. A. Silva, "Energy consumption in microservices architectures: a systematic literature review," 2023.

[71] X. Chen and S. Xiao, "Multi-objective and parallel particle swarm optimization algorithm for container-based microservice scheduling," *Sensors*, vol. 21, p. 6212, 2021.

[72] Z. Liu, H. Yu, G. Fan, and L. Chen, "Reliability modelling and optimization for microservice-based cloud application using multi-agent system," *IET Communications*, vol. 16, pp. 1182–1199, 2022.

[73] "Two month's worth of all http requests to the nasa kennedy space center," http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html, available online; accessed on DATE.

[74] R. Hyndman and Y. Khandakar, "Automatic time series forecasting: The forecast package for r," *Journal of Statistical Software*, vol. 27, no. 3, pp. 1–22, 2008.

[75] M. Borkowski, S. Schulte, and C. Hochreiner, "Predicting cloud resource utilization," in *Proceedings of the 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*. Shanghai, China: IEEE/ACM, dec 2016, pp. 37–42.

[76] "1998 worldcup website access logs," http://ita.ee.lbl.gov/html/contrib/WorldCup.html, accessed: *insert date here*.

[77] M. Yan, X. Liang, Z. Lu, J. Wu, and W. Zhang, "Hansel: Adaptive horizontal scaling of microservices using bi-lstm," *Applied Soft Computing*, vol. 105, p. 107216, 2021.

[78] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Adaptive ai-based auto-scaling for kubernetes," in *Proceedings of the 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. Melbourne, Australia: IEEE/ACM, may 2020, pp. 599–608.

[79] J. Dogani, F. Khunjush, and M. Seydali, "K-agrued: A container autoscaling technique for cloud-based web applications in kubernetes using attention-based gru encoder-decoder," *Journal of Grid Computing*, vol. 20, p. 40, 2022.

[80] N.-M. Dang-Quang and M. Yoo, "Multivariate deep learning model for workload prediction in cloud computing," in *2021 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 2021, pp. 858–862.

[81] ——, "A study on deep learning-based multivariate resource estimation with feature selection in cloud computing," , pp. 366–369, 2021.

[82] D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of kubernetes pods," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–5.

[83] L. Ju, P. Singh, and S. Toor, "Proactive autoscaling for edge computing systems with kubernetes," in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, 2021, pp. 1–8.

[84] M. Tahir, Q. M. Ashraf, and M. Dabbagh, "Towards enabling autonomic computing in iot ecosystem," in *2019 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. IEEE, 2019, pp. 646–651.

[85] M. Różańska and G. Horn, "Marginal metric utility for autonomic cloud application management," in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, ser. UCC '21. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3492323.3495587

[86] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking significance of software components based on use relations," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 213–225, 2005.

[87] T. Shi, H. Ma, G. Chen, and S. Hartmann, "Location-aware and budget-constrained application replication and deployment in multi-cloud environment," in *2020 IEEE International Conference on Web Services (ICWS)*. IEEE, 2020, pp. 110–117.