Employing Software Diversity in Cloud Microservices to Engineer Reliable and Performant Systems

Nazanin Akhtarian, Hamzeh Khazaei and Marin Litoiu
Department of Electrical Engineering & Computer Science
York University, Toronto, Canada
{nakhtari,hkh,mlitoiu}@yorku.ca

ABSTRACT

In the ever-shifting landscape of software engineering, we recognize the need for adaptation and evolution to maintain system dependability. As each software iteration potentially introduces new challenges, from unforeseen bugs to performance anomalies, it becomes paramount to understand and address these intricacies to ensure robust system operations during the lifetime. This work proposes employing software diversity to enhance system reliability and performance simultaneously. A cornerstone of our work is the derivation of a reliability metric. This metric encapsulates the reliability and performance of each software version under adverse conditions. Using the calculated reliability score, we implemented a dynamic controller responsible for adjusting the population of each software version. The goal is to maintain a higher replica count for more reliable versions while preserving the diversity of versions as much as possible. This balance is crucial for ensuring not only the reliability but also the performance of the system against a spectrum of potential failures. In addition, we designed and implemented a diversity-aware autoscaling algorithm that maintains the reliability and performance of the system at the same time and at any scale. Our extensive experiments on realistic cloud microservice-based applications show the effectiveness of the proposed approach in this paper in promoting both reliability and performance.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**.

KEYWORDS

Software Multi-versioning, Microservices, Reliability, Dynamic Scaling, Diversity Factor

ACM Reference Format:
Nazanin Akhtarian, Hamzeh Khazaei and Marin Litoiu. 2024. Employing Software Diversity in Cloud Microservices to Engineer Reliable and Performant Systems. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2024 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the contemporary software landscape, microservices have emerged as a preferred architectural style, enabling modularity, scalability, and easy maintainability. These fine-grained services encapsulate specific functionalities, making systems more flexible and manageable. As the adoption of microservices grew, so did the need for efficient orchestration tools to manage these modular components, especially when housed within containers. Kubernetes[1] is a leading orchestration platform designed specifically for container management, ensuring smooth deployment, scaling, and operation of applications built using microservices.

Beyond architecture and orchestration, the appreciation for multi-version software systems is on the rise. This approach, involving the concurrent maintenance of multiple software versions, enables systems to intelligently select and deploy the most suitable software version based on real-time conditions and metrics, addressing challenges related to performance optimization, fault tolerance, reliability, security vulnerabilities, and continuous availability [1–5]. Central to this strategy is the challenge of preserving system reliability, especially given the intricate layers added by multiple software versions. We introduce several novel contributions aimed at enhancing system reliability and performance:

- **Conceptualization of Multi-version Containers:** Introducing transparency in multi-versioning at the container level, allowing users to interact with services without concern for underlying version differences.
- **Dynamic Controller Implementation:** A controller to adjust the population of each software version based on its reliability score, aiming to maintain a higher replica count for more reliable versions while preserving version diversity.
- **Diversity Emphasis:** Introduction of the Diversity Factor (DF), which quantifies the distribution of software versions, advocating for a balanced deployment approach over a solely reliability-centric one.
- **Diversity-Aware Auto-Scaling Algorithm:** Drawing inspiration from natural selection, this framework dynamically allocates resources to the most reliable deployments, mirroring the survival of the fittest in natural ecosystems. Our diversity-aware autoscaling algorithm is designed to maintain system reliability and performance simultaneously, regardless of scale.

This work proposes employing software diversity to enhance system reliability and performance simultaneously. Using the calculated reliability score, we implemented a dynamic controller responsible for adjusting the population of each software version. The

[1] https://kubernetes.io

goal is to maintain a higher replica count for more reliable versions while preserving the diversity of versions as much as possible. This balance is crucial for ensuring the reliability of the system against a spectrum of potential failures. Our extensive experiments on realistic cloud microservice-based applications show the effectiveness of our proposed approach. All the artifacts, including source codes and documentation, related to this paper are publically available to facilitate the reproducibility of our proposed methodology[2].

The rest of the paper is organized as follows. Section 2 provides background information about multi-version software systems, microservices, auto-scaling, and load balancing. In Section 3, we present the concept of our approach. In Section 4, we explain our experimental setup. Section 5 discusses the results of our experiments. Section 6 gives an overview of the related work, and Section 7 explains the threats to validity and the future work. Finally, Section 9 concludes the paper.

2 BACKGROUND

This section provides the central concepts in our study and establishes the link between them and software diversity.

2.1 Microservice Architecture

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 [6]. 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.

2.2 Auto-Scaling

In the context of software multi-versioning and microservices, understanding the role of auto-scaling strategies is vital for performance and resource optimization. Elasticity in cloud computing addresses unpredictable workloads, allowing for the adjustment of resources in tune with workloads to balance service performance and costs [7, 8]. This dynamic adjustment is crucial not only for meeting demand but also for selecting and scaling from the most reliable and efficient versions of services, thereby enhancing the overall stability and performance of the system. In environments where workloads fluctuate, the capability to dynamically scale resources, both at the microservices level and across different software versions, becomes essential for maintaining service stability.

The importance of auto-scaling algorithms in cloud computing is increasing. Proper provisioning of replicas is crucial: underprovisioning can deteriorate performance and risk service unavailability, causing revenue losses [9], while over-provisioning leads to resource waste and higher costs. Therefore, auto-scaling is essential in optimizing Pod replica resources and avoiding service level objective (SLO) violations.

Auto-scaling may be divided into two categories: proactive and reactive [10]. **Reactive techniques** analyze the system's current status and decide the scaling based on predefined rules or thresholds. **Proactive techniques** examine the historical data, predict the future, and perform scaling decisions in advance. In this work, we primarily employ a reactive auto-scaling approach. The choice is influenced by the desire to balance efficiency, simplicity, and robustness.

2.3 Load Balancing

Load balancing techniques distribute incoming network traffic across multiple servers to ensure no single server is overwhelmed with too much traffic. This helps maintain the availability and reliability of applications and services. Common methods encompass Round Robin, Least Connections, Least Response Time, IP Hash, and URL Hash [11]. In situations with uneven server loads, adaptive load balancing methods, like Weighted Round Robin or Weighted Least Connections, prove advantageous [12]. These adaptive methods dynamically distribute traffic based on factors such as server performance or workload fluctuations, ensuring efficient resource usage and maintaining system performance.

In our work, we adopt adaptive load balancing, specifically using a Weighted round-robin for request distribution. The load balancer in our approach distributes the load based on the reliability measures of underlying software versions.

3 OUR APPROACH

This work employs an evolutionary strategy based on natural selection for Kubernetes Pod scaling, rewarding more reliable deployments with additional Pods. We employ a self-adaptive MAPE (Monitor, Analyze, Plan, Execute) framework:

- Monitor: Using Prometheus, we continuously monitor metrics like Pod restarts and memory consumption. Nginx logs are used for client-side response times, and system workloads are observed for scaling insights.
- Analyze: We assess Kubernetes deployments' reliability based on a weighted average of three metrics, evaluating workload for scaling needs.
- Plan: Based on reliability scores, we decide on adjusting Pod replica counts or traffic distribution among software versions.
- Execute: Changes are made to replica counts and load balancer configurations.

3.1 System Architecture

Our solution has been materialized in two fundamental components: the **Load Balancer** and the **Scaling Engine**, depicted in Figure 1. The architecture distributes the workload dynamically based on microservice version reliability scores and manages replicas accordingly. The load generator drives the traffic to an Nginx-powered[3] load balancer, which uniformly divides requests across

[2] https://github.com/nazanin97/ReplicaBalancer/tree/main

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

multi-version backend servers. The Scaling Engine calculates the reliability score for microservice versions and adjusts replicas based on this score. It also continually assesses and adjusts the overall Pod replica count based on workload monitoring, enhancing system resilience to variable demands and optimizing the allocation of resources. Additionally, it provides configurations to Nginx for weight distribution.

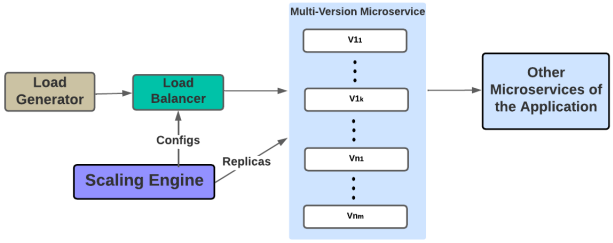

Figure 1: System architecture. The diagram illustrates the structure of the proposed solution, detailing component interactions and data flow paths.

3.2 Auto-scaling Engine

Our scaling engine has two core functionalities:

(1) It dynamically updates the load balancer configurations based on deployment reliability scores to ensure balanced traffic distribution across Kubernetes Pods.
(2) It employs the monitoring system to construct a real-time reliability scoring mechanism based on three metrics: restart counts, response time variability, and memory usage.

3.2.1 ***Continuous Configuration of Load Balancer at Runtime.*** The system proactively updates the load balancer settings to capture the reliability scores continuously. This dynamic adaptation ensures deployments with superior reliability scores handle a larger share of traffic, thereby improving overall system performance.

3.2.2 ***Reliability Scoring System.*** We devised a system to evaluate the reliability of Kubernetes deployments methodically, calculating a weighted average of key metrics to produce a reliability score. This score is derived from a linear utility function that integrates monitored metrics into an overarching reliability evaluation, consistent with methodologies in the literature [13].

Utility Function Definition. The reliability utility function, denoted as $U_{\text{reliability}}$ is formulated at a specific time t and is expressed as:

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

where $\boldsymbol{\theta}(t)$ represents the metrics vector at time t, $\boldsymbol{\phi}$ symbolizes additional parameters affecting the utility function, w_i corresponds to the weight of each metric, ensuring $\sum_{i=1}^{N} w_i = 1$, and $u_i(\theta_i(t)|\phi)$ denotes the individual utility functions for each metric. These utility functions capture reliability by translating monitored metrics into reliability rates, thus reflecting the probability of system stability at time t.

Scoring Methodology. Our scoring system involves continuous monitoring of three primary metrics: restart count, variability in response time, and variability in memory usage. The reliability scores derived from these metrics inform the dynamic adjustment of Pod replicas, enabling the system to respond promptly to workload fluctuations and reliability changes. The steps are as follows:

(1) **Metric Retrieval:** Metrics are collected using specific queries from the monitoring system (step 7 in Algorithm 1).
(2) **Metric Normalization:** The metrics are normalized linearly between 0 and 1, utilizing the utility function u specific to each metric m in the metric set I. For every version v in the version set V, the function is defined as:

$$\forall v \in V, \forall m \in I : u_m(v) = 1 - \frac{\text{metric}_m(v) - \min(\text{metric}_m(V))}{\max(\text{metric}_m(V)) - \min(\text{metric}_m(V))}$$

(3) **Reliability Score Calculation:** The overall reliability score for each deployment is computed by combining individual metrics with their respective weights (step 20 in Algorithm 1):

$$\begin{aligned}\text{Reliability\_Score}(v) =& \text{responseTimeWeight} \times u_{\text{responseTime}}(v) \\ &+ \text{restartWeight} \times u_{\text{restarts}}(v) \\ &+ \text{memoryWeight} \times u_{\text{memoryUsage}}(v) \quad (1)\end{aligned}$$

Replica Allocation Strategy. The system's architecture is designed to fine-tune the distribution of replicas among software versions in alignment with their reliability scores, maintaining at least one replica per version to prevent single-version failures. 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. The replica allocation process, as detailed in `AdjustReplicaDistribution` function in Algorithm 3, involves a two-step adjustment: firstly, the proportional number of replicas for each software version is calculated based on its reliability score, and secondly, it fine-tunes this distribution to match the total number of replicas required.

3.2.3 ***Adaptive Scaling for Changing Workloads.*** Another factor that may impact the system's reliability is the changing workload. Autoscaling is primarily done to maintain performance, but indirectly, if done properly, it will also improve system reliability. When scaling out/in, the main question would be which version should be scaled. Algorithm 1 details the logic of our scaling engine.

We use a threshold-based approach based on aggregate CPU utilization for dynamic scaling decisions (see `scalingAction` function in Algorithm 2). To mitigate the ping-pong effect in scaling, our approach includes a historical analysis, as outlined in the `decideScaleBasedOnHistory` function in Algorithm 2.

Algorithm 1 Version-Aware Autoscaling

Require: *MONITORING_TIME*, *ACTION_TIME MAX_REPLICAS*, *MIN_REPLICAS TOTAL_REPLICAS*

while True **do**
  if elapsed = *MONITORING_TIME* **then**
    $currentCPU \leftarrow getCPU()$
    $currentScalingState \leftarrow scalingAction(currentCPU)$
    $scaleHistory$.add($currentScaleState$)
    for deploymentVersion in deploymentVersions **do**
      getPrometheusData(deploymentVersion)
    end for
  else if elapsed = *ACTION_TIME* **then**
    $scaleAction \leftarrow decideScaleBasedOnHistory(scaleHistory)$
    if $scaleAction$ = Increase **and**
  $TOTAL\_REPS < MAX\_REPS$ **then**
      $TOTAL\_REPLICAS$ =+ 1
    else if $scaleAction$ = Decrease **and**
  $TOTAL\_REPS > MIN\_REPS$ **then**
      $TOTAL\_REPLICAS$ =- 1
    end if
    $reliability\_scores \leftarrow$ empty array
    for deploymentVersion in deploymentVersions **do**
      Compute metrics & reliabilityScore for deploymentVersion
      $reliability\_scores$.add($reliabilityScore$)
    end for
    AdjustReplicaDistribution(deploymentVersions)
  end if
end while

Algorithm 2 Selecting the Scaling Action

Require:

$MAX\_CPU \leftarrow Y\%$
$MIN\_CPU \leftarrow X\%$
function scalingAction(cpu)
  if $cpu > MAX\_CPU$ **then**
    return Increase
  else if $cpu < MIN\_CPU$ **then**
    return Decrease
  else
    return NoChange
  end if
end function
function decideScaleBasedOnHistory($history$)
  $increaseCount \leftarrow 0$
  $decreaseCount \leftarrow 0$
  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

Algorithm 3 Adjusting Replicas Based on Reliability Scores

Require: *TOTAL_REPLICAS*

function AdjustReplicaDistribution(deploymentVersions)
  $total\_score \leftarrow 0.0$
  for score in reliability_scores **do**
    $total\_score$ =+ $score$
  end for
  newReplicas $\leftarrow$ array of zeros for each deployment version
  fractionalParts $\leftarrow$ array of zeros 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] $\leftarrow$ floor of $proportionalReplica$
    fractionalParts[index] $\leftarrow$ $proportionalReplica$ – newReplicas[index]
    if newReplicas[index] = 0 **then**
      newReplicas[index] $\leftarrow$ 1
    end if
    $allReplicas$ =+ 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

3.3 Diversity Factor: Quantifying Version Variation

In the adaptive scaling process, especially under variable workloads, a primary consideration is determining which software version to scale. A straightforward approach might be to favor scaling the most reliable versions. However, this strategy, while seemingly efficient, overlooks the inherent unpredictability of software behaviour. Sole reliance on a single version may expose the system to unforeseen vulnerabilities or issues specific to that iteration. In this context, the Diversity Factor (DF) emerges as an essential metric, serving as a measure of version diversity across our deployments. We define the DF as:

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

where $\sigma(R)$ is the standard deviation of replica distribution. For instance, with versions $V1$, $V2$, and $V3$ and replica counts $R1$, $R2$, and $R3$:

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

where $\bar{R}$ is the average replica count. DF's importance becomes evident in the context of the Algorithm 1. When a scaling action is determined based on workload changes, the total number of replicas in the system is adjusted. However, instead of selectively increasing or decreasing specific versions, we apply the Algorithm 3. This synergy between the two algorithms ensures that the distribution of these new total replicas takes into account both the reliability scores and the DF.

4 EXPERIMENTAL SETUP

This section presents an in-depth overview of our experimental setup designed to evaluate the performance of our reliability engine and the version-aware autoscaling engine. First, we examine how our reliability engine maintains/improves the reliability of the software system under different types of chaos scenarios. Then, we put our version-aware scaling engine under the test to show that it not only maintains the performance but also maintains/improves the reliability of software systems at different scales. More specifically, the scaling engine increases/decreases the population in such a way as to optimize the diversity factor.

4.1 Cluster Deployment and Configuration

We deployed a cluster including two Virtual Machines (VM) in the Compute Canada cloud[4] to evaluate our proposed solution. 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 Ubuntu 22.04.2 and 15 GB RAM, powered by 4 VCPUs and 20 GB of primary disk storage, plus 83 GB ephemeral storage.

4.1.1 ***Initial Deployment and Configuration.*** To operationalize our system, a deployment named "ReplicaBalancer" is created as in Listing 1. "ReplicaBalancer" acts as our reliability/scaling engine. The deployment is initiated with a Kubernetes command and configured with Listing 2 environment variables.

Listing 1: Initiating the "ReplicaBalancer" deployment

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

Once "ReplicaBalancer" is up and running, the following command sets the environment variables that steer its behaviour.

Listing 2: Setting environment variables for "ReplicaBalancer"

```
$ 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
```

4.1.2 ***Parameterization Overview.*** The parameters employed in our experiments such as `TOTAL_REPLICAS`, `MONITORING_TIME`, and CPU thresholds, were selected to demonstrate the capabilities of our approach within the context of our experimental environment. It is important to note that the specific choice of these parameters was not the focal point of this research. These parameters can be fine-tuned according to the specific requirements of the user and the system.

Here's an overview of environment variables:

- `DEPLOYMENT_IMAGES_REPLICAS`: Defines replica distribution for Docker images. If not specified, the system evenly allocates the total replicas among provided images.
- `MONITORING_TIME`: Sets monitoring frequency.
- `ACTION_TIME`: Specifies the interval for system response actions.
- `TOTAL_REPLICAS`: Indicates initial replica count, adjustable based on workload.
- `MAX_REPLICAS & MIN_REPLICAS`: Define the maximum and minimum replica limits.
- `MAX_CPU & MIN_CPU`: Set CPU utilization thresholds for scaling.
- `SCALING`: Toggles autoscaling based on observed workload.

4.2 Subject Systems

We analyzed the Online Boutique application[5], a cloud-native microservices application from Google. The architecture comprises 11 microservices, facilitating product browsing, cart addition, and purchases. The overall architecture is depicted in Figure 2.

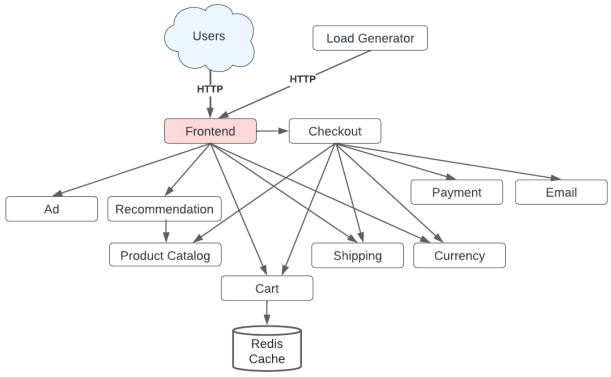

Figure 2: Online Boutique microservices layout. This visualization showcases the layout and interconnections of various microservices in the Online Boutique application.

4.3 Workload

We used Locust to simulate a typical e-commerce user flow. The workload consisted of 100 virtual users performing actions such as browsing products, adding items to the cart, and completing purchases. Requests were sent at random intervals to mimic real user behavior. During the experiments, we collected metrics such as response time, CPU usage, and memory consumption. The data was then analyzed to assess the performance and reliability of our proposed autoscaling approach.

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

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

4.4 Critical Microservice Identification

Choosing critical microservices is essential in multi-versioning, as it can impact resource allocation, budget and overall system reliability. Our study designates the frontend service as crucial because it is the primary user interface and function in aggregating data from various backend services. This decision aligns with PageRank-based methods that assess microservice importance ([14–16]). While we do not delve into these methods, interested readers can consult the referenced studies. By concentrating on the frontend, we aim to deliver a robust user experience and maintain the operational integrity of the application on a manageable budget.

4.5 Weight Configuration for Reliability Scoring

An integral part of our reliability scoring system is the assignment of specific weights to different metrics, which play a pivotal role in the overall reliability assessment. The weights assigned to each metric are as follows:

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

The weights assigned in Equation 1 are based on the influence of each metric on the overall reliability of the Kubernetes system. Pod failure, weighted most heavily at 0.5, is crucial because it directly impacts system reliability. Frequent pod restarts can indicate underlying stability issues, leading to service disruption and reduced reliability [17]. The weight of 0.3 for memory usage variability addresses the importance of detecting memory leaks. Memory leaks can lead to system degradation or crashes, critically impacting reliability [18]. Response time variability, with a lower weight of 0.2, affects user experience and system responsiveness. Although important, it's less directly related to the core operational reliability than pod failures and memory leak issues, hence has lower priority.

4.6 Chaos Mesh Overview

Chaos Mesh[6] is an open-source platform for Chaos Engineering on Kubernetes. It simulates system faults and measures recovery processes, facilitated by a user-friendly web interface. The platform provides diverse chaos scenarios like network disruptions, system call failures, and resource constraints. Our experiments utilize Pod Chaos, HTTP Chaos, and Stress Chaos to emulate common failure modes in microservices.

4.7 Chaos Experimentation

Given that different software versions may exhibit unique bugs affecting reliability, we simulate multi-versioning by injecting specific types of chaos into each version. This approach allows us to model distinct performance and reliability characteristics without the need to develop separate version functionalities. We align our chaos types with the key metrics defining reliability: restart counts, response time variability, and memory usage variability. Consequently, in our experimentation, three frontend microservice versions named "Faulty", "InconsistentResponse", and "MemoryLeak" are considered. We start the experiment with 3 healthy identical versions, and then, using Chaos Mesh, we introduce the above-mentioned bugs in the replicas to imitate the multi-versioning in software systems. Each version suggests the chaos linked to it:

(1) **Faulty Version:** This version simulates the impact of bugs that cause frequent Pod restarts. Frequent restarts disrupt service continuity, leading to higher downtime and reduced reliability. This is implemented using **Pod Chaos** that targets the "frontend-faulty-deployment" every 3 minutes, terminating associated Pods for 30 seconds.
(2) **InconsistentResponse Version:** Shows variable response times. Through **HTTP Chaos**, which introduces a 2-second delay in responses from the "frontend-inconsistent-response-deployment" every 4 minutes for 2 minutes, we demonstrate how variability in response times can reduce the predictability and performance standards expected from reliable systems. Such inconsistencies also impact user experience.
(3) **MemoryLeak Version:** Faces a memory leak, pointing out its robustness limitations. This is explored using **Stress Chaos**, imposing memory stress on "frontend-memory-leak-deployment" every 4 minutes with two workers consuming 20MB memory each for 2 minutes. Memory leaks threaten the reliability of a system by compromising its ability to maintain operational performance over time.

5 EXPERIMENTAL EVALUATION

To comprehend the system's behaviour, we conducted two experiments. The first experiment observed changes in replica distribution based on the reliability of individual components by introducing specific chaos. The second experiment examined the system's response to varying workloads, focusing on adaptability in replica scaling with changes in CPU utilization.

5.1 Experiment 1: Evolution under Constant Workload

In the first experiment, we studied the system under constant workload conditions with 20 concurrent users over 2 hours (Figure 3). 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 allow the system to run undisturbed to observe the replica distribution across different versions. As shown in Figure 7, 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 and versions were identical.

Subsequently, we introduce Pod chaos, detailed in Section 4.7. Following this, a noticeable restart increase for the frontend-faulty-deployment was evident, as depicted in Figure 5. Additionally, as seen in Figure 6, memory usage patterns for this version fluctuated. Simultaneously, Figure 7 captures the system's adaptive response regarding 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

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

the Pod chaos, which was an expected outcome. It's crucial to note that, at this juncture, only the Pod chaos was in play.

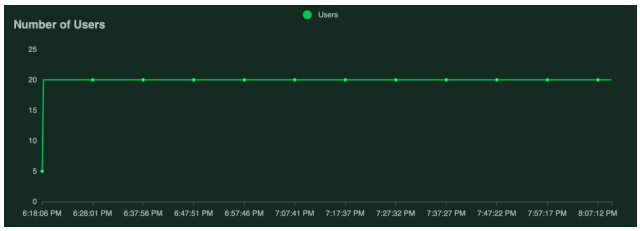

Figure 3: Number of users. This chart presents the number of users accessing the system during the first experiment.

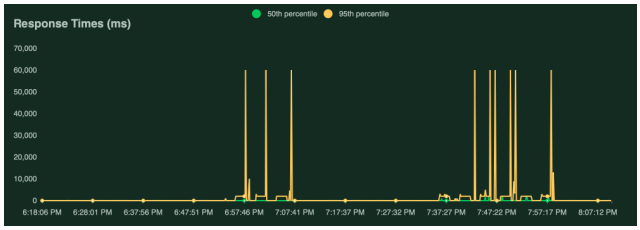

Figure 4: Application's response time during the first experiment.

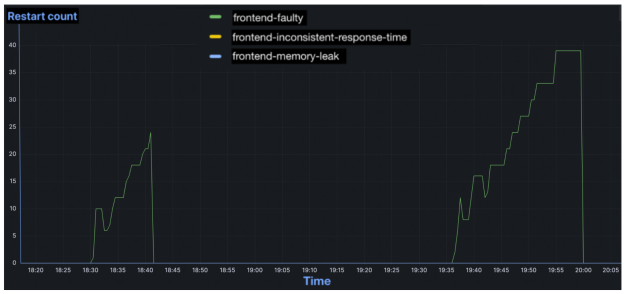

Figure 5: Restart count of frontend microservice versions. This chart illustrates the frequency and patterns of system restarts over a specific period.

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. 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.

In a subsequent test, we introduced stress chaos to the frontend-memory-leak-deployment. As seen in Figure 6, captured a rise in memory consumption. The result on replica distribution is illustrated in Figure 7, which involves the subtraction of a replica from

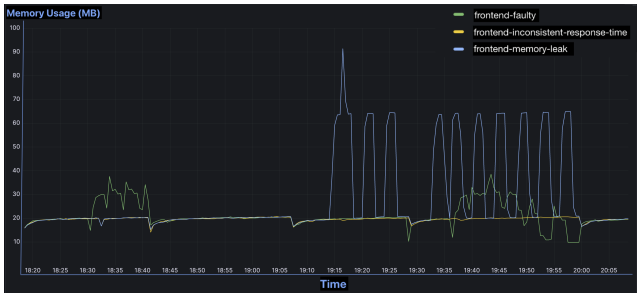

Figure 6: Memory usage over time (measured in MB). This graph provides a comprehensive look at the memory consumption patterns for different frontend deployments.

frontend-memory-leak and its addition to frontend-inconsistent-response. Following 16 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 4.5. After stopping all chaos injections, the system eventually returned to its initial balanced state.

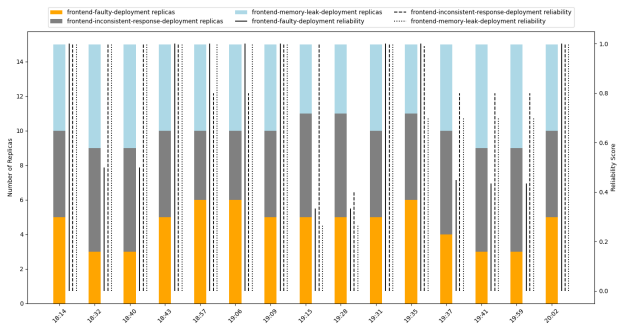

Figure 7: Replica and reliability over time. The bar chart illustrates the number of replicas for different frontend microservice versions over time. Adjacent vertical lines, differentiated by line style, represent the reliability scores for each version.

5.2 Experiment 2: Evolutionary-Aware Auto-Scaling

Our second experiment was designed to understand the system's dynamic scaling capabilities in relation to variable workloads, with a focus on CPU usage. We aimed to see how this adaptive scaling behaviour impacts software diversity, reliability, and performance.

As the system was subjected to different user loads (Figure 8), we closely monitored the frontend microservice Pods' CPU usage as illustrated in Figure 10. Over time, a direct correlation was observed between the workload and CPU usage as we changed the number of users. As the workload intensified, there was a consequent increase in CPU usage. In line with the user-configurable thresholds discussed in Section 4.1.2, 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 a scale-out in the number of Pod replicas.

The early phase of the experiment, characterized by lower user numbers and CPU usage below the 20% mark, saw a reduction in the total number of replicas, showcasing the system's efficiency and cost-effectiveness (Figure 11). As the experiment progressed with an incremental user load, the system dynamically scaled up, adding more replicas to handle the increased demand, thereby demonstrating its capability to maintain performance under varying workload conditions.

As seen in Figure 11, during both scale-out and scale-in phases, the system's scaling decisions were informed by the reliability scores of different software versions, ensuring that replicas were allocated to preserve software diversity. Therefore, all three objectives, namely performance, cost, and reliability, were achieved simultaneously.

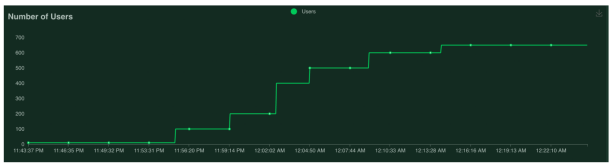

Figure 8: Number of users. This chart presents the number of active users accessing the system over a specific duration for the second experiment.

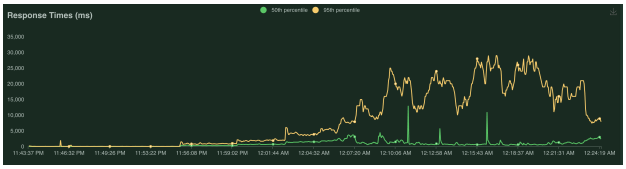

Figure 9: Application's response time during the second experiment.

Figure 10: Average CPU utilization of frontend microservice Pods over time.

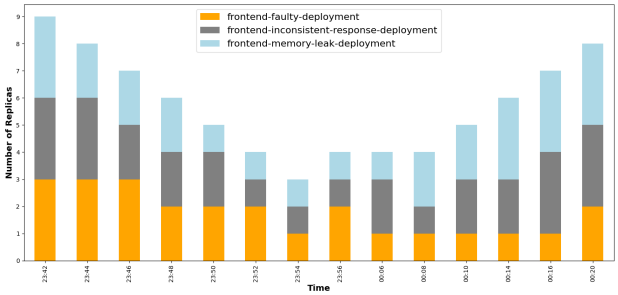

Figure 11: Dynamic scaling of frontend microservice Pods. This chart visualizes the system's dynamic scaling capabilities in response to varying workloads, highlighting the changes in the number of replicas over time. The system's adaptability to workload fluctuations is evident from the shifts in replica counts.

6 RELATED WORK

This section discusses prior work related to our research on software multi-versioning, containerized systems, microservices, reliability for microservices, and auto-scaling approaches.

6.1 Software Multi-Versioning

Software multi-versioning has been extensively researched. Larsen et al. [4] were among the pioneers, examining the effects of automated software redundancy on system security. The work of Franz et al. followed this [19], who advocated for multi-versioning as a strategic defence against targeted cyber attacks. Persaud et al. [20] added a new dimension by integrating genetic algorithms with software redundancy for enhanced security. Cigsar et al. [21] explored multi-versioning to augment the reliability of repairable systems, while Gracie et al. [22] discussed its applications in ensuring system safety. Gorbenko et al. [5] utilized multi-versioning to improve the availability and reliability of web services. Borck et al. [23] introduced FEVIS, a program diversification method for detecting cyber attacks.

6.2 Software Multi-Versioning for Containerized Systems

In the realm of containerized systems, multi-versioning has been pivotal in enhancing system robustness. Wang et al. [24] advocated for its use in critical cloud software components to support fault tolerance, emphasizing cost-effective and selective application. Zheng et al. [2, 3] demonstrated the efficacy of multi-versioning in improving reliability and fault tolerance in service-oriented architectures. Gholami et al. [1] made a significant contribution with DockerMV, integrating multi-versioning into the Docker framework to facilitate application scaling. Dhaliwal and Khazaei [25] focused on optimizing the performance of microservice systems by combining software multi-versioning with dynamic load balancing, demonstrating improvements in response times and resource utilization. The studies by Mohamed and El-Gayar [26] and Pinto et al. [27] further enriched this domain by assessing latency prediction and user acceptance in containerized environments.

6.3 Microservices in Action

The surge in microservices research has led to diverse applications. Timur et al. [28] proposed a distributed microservices-based solution for scalable deep learning facial recognition, leveraging Docker for data management. Lu et al. [29] developed a microservice platform for SSA data analytics, demonstrating its application through satellite-related tasks. Asaithambi et al. [30] introduced the Microservice-Oriented Big Data Architecture (MOBDA) for processing large-scale transportation data, focusing on Singapore's public transport system. Ali et al. [31] explored modular microservices for real-time IoT-based health monitoring, exemplifying the adaptability of microservices in various sectors.

6.4 Reliability for Microservices

The aspect of microservice reliability has been a focal point in recent research endeavours. Vincenzo and Dragi [32] introduced a novel task unloading approach using Pareto optimization, targeting key performance metrics. Clab et al. [33] proposed a delay-adaptive strategy for replica synchronization, a critical factor in software reliability. Subsequent studies by Chen et al. [34], Liu et al. [14, 35], and Pietrantuono et al. [36] have significantly contributed to enhancing the reliability of microservice-based cloud applications, employing various methodologies including redundancy and circuit breakers. CoScal employs a reinforcement learning-based approach to optimize resource scaling for microservices, combining horizontal scaling, vertical scaling, and brownout techniques to handle complex workload scenarios effectively

6.5 Auto-Scaling Approaches

The evolution of auto-scaling in cloud computing has seen a variety of innovative approaches. Al-Dhuraibi et al. [37] introduced ELASTICDOCKER, a reactive scaling strategy in Kubernetes, focusing on resource optimization and cost-effectiveness. Rodrigo et al. [38] utilized the ARIMA model for accurate cloud workload prediction, while Messias et al. [39] combined statistical methods with genetic algorithms for improved forecasting accuracy. Advanced predictive models like Bi-LSTM and AI-influenced techniques have been developed by researchers like Tang et al. [40], Ming Yan et al. [41], and Laszlo Toka et al. [42]. Dang-Quang and Yoo [10, 43, 44] and Dogani et al. [45] have made notable contributions to proactive auto-scaling in Kubernetes, showcasing the potential of deep learning and attention-based models in this domain. Xu et al. [46] proposed CoScal, a reinforcement learning-based approach that combines horizontal scaling, vertical scaling, and brownout techniques to handle complex workload scenarios effectively. While these auto-scaling mechanisms primarily focus on optimizing performance and cost, it is equally important to ensure that reliability is not compromised.

7 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.

7.1 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.

7.2 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.

7.3 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.

8 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.
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.

9 CONCLUSION

Due to the substantial costs associated with implementing multiple system versions, multi-versioning has often been restricted to critical-mission systems. However, microservice architecture allows for selective multi-versioning of critical components, making it viable across various systems. In this paper, we proposed a reliability engine that transparently maintains/improves the reliability of the system based on the reliability score of microservice versions. This engine can be augmented to any microservice-based application

regardless of the internal logic of the underlying system. We also extend the reliability engine to conduct the scaling with the system's reliability in mind. Unlike conventional autoscaling, in which we are only concerned about performance, the Evolutionary-aware (i.e., version-aware) autoscaling engine proposed in this paper simultaneously satisfies performance and reliability at any scale. We conducted realistic experiments on the cloud to validate the proposed approach in this paper.

REFERENCES

[1] 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. Association for Computing Machinery, 2020, p. 150–160.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] M. Elgili, "Load balancing algorithms round-robin (rr), least-connection and least loaded algorithm," *ResearchGate*, 2020.

[12] 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*, 2022.

[13] M. Różańska and G. Horn, "Marginal metric utility for autonomic cloud application management," ser. UCC '21. Association for Computing Machinery, 2022.

[14] 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.

[15] 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.

[16] 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.

[17] Z. Li, H. Wei, Z. Lyu, and C. Lian, "Kubernetes-container-cluster-based architecture for an energy management system," *IEEE Access*, vol. 9, pp. 84 596–84 604, 2021.

[18] J. Costa, R. Matos, J. Araujo, J. Li, E. Choi, T. A. Nguyen, J.-W. Lee, and D. Min, "Software aging effects on kubernetes in container orchestration systems for digital twin cloud infrastructures of urban air mobility," *Drones*, 2023.

[19] M. Franz, "E unibus pluram: Massive-scale software diversity as a defense mechanism," ser. NSPW '10. Association for Computing Machinery, 2010.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] P. Dhaliwal and H. Khazaei, "Reviving software diversity in microservices to optimize the performance of software systems," in *2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, 2023.

[26] 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.

[27] 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.

[28] T. D. Timur, I. K. E. Purnama, and S. M. S. Nugroho, "Deploying scalable face recognition pipeline using distributed microservices," in *2019 International Conference on Computer Engineering, Network, and Intelligent Multimedia (CENIM)*, 2019, pp. 1–5.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] Z. Liu, G. Fan, H. Yu, and L. Chen, "Modelling and analysing the reliability for microservice-based cloud application based on predicate petri net," *Expert Systems*, vol. 39, no. 6, 2022.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

[42] 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)*. IEEE/ACM.

[43] 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.

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

[45] 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*, 2022.

[46] M. Xu, C. Song, S. Ilager, S. S. Gill, J. Zhao, K. Ye, and C. Xu, "Coscal: Multifaceted scaling of microservices with reinforcement learning," *IEEE Transactions on Network and Service Management*, 2022.