Misconfiguration of Cluster and IoT Systems Recovery: Extended Experiments

Areeg Samir[1][0000−0003−4728−447X], Håvard Dagenborg[2][0000−0002−1637−7262], and Nabil EL Ioini[3][0000−0002−1288−1082]

Computer Science Department, The Arctic University of Norway, Tromsø Hansine Hansens veg 18, 9019 {areeg.s.elgazazz, havard.dagenborg}@uit.no
School of Computer Science, University of Nottingham, Malaysia, Semenyih 43500 {Elioini.Nabil}@nottingham.edu.my

Abstract. Containerized cluster systems and edge devices are vulnerable to security breaches when configuration errors occur. In this paper, we propose more experiments on the self-healing approach for misconfigurations that adopt a Markov Decision Process to determine the optimal recovery action or policy that maximizes performance metrics. The experiments conducted are an improvement on our previous self-healing approach. Our findings demonstrate the effectiveness of the proposed controller in enhancing performance and reliability, leading to precise and improved results.

1 Introduction

Complex computational environments are susceptible to security breaches when misconfigurations occur. Misconfigurations can manifest in various forms, such as improperly configured firewalls, access control lists, or security settings on IoT devices and servers. These misconfigurations create unintended vulnerabilities that malicious actors can exploit. As a result, sensitive data, intellectual property, and user information can be compromised, leading to privacy violations, financial losses, and reputational damage for organizations and individuals.

1.1 Problem Statement

Several factors contribute to the prevalence of misconfigurations: (1) human error: Misconfigurations frequently arise due to human errors during system setup, updates, or changes. The dynamic nature of these environments increases the likelihood of oversight. (2) Lack of automation: Manual configuration processes are error-prone and time-consuming. The absence of automated tools and practices can exacerbate the misconfiguration problem. (3) Scale and complexity: Modern online services and IoT applications often involve a multitude of interconnected components, making it challenging to maintain a comprehensive understanding of the entire environment and its configurations. Developing proactive recovery strategies, implementing robust security measures, and using automation tools are critical steps to mitigate these risks and safeguard sensitive data and systems.

1.2 Objective

In this paper, we extend the self-healing controller experiments in [13] that fix the misconfiguration of edge devices and containerized services in the back end by using Markov Decision Processes (MDPs) to model cluster recovery management. By using MDP, informed recovery decisions that account for uncertainty and complexity can lead to more efficient resource utilization and improved cluster performance. The extension includes additional experimentation to mitigate the impact of configuration errors on system performance.

The paper is organized as follows. Section 2 provides a background on the main concepts used in the paper. Section 3 discusses the related work. Section 4 evaluates the controller. Section 5 provides prospective research directions. Section 6 explains a use case of the controller. Section 7 concludes the paper and presents the direction for future work.

2 Background

2.1 Configuration Errors

Configuration errors occur when settings, parameters, or options within a system are incorrectly set, leading to undesirable behavior or suboptimal performance. These errors can stem from a variety of sources, including human errors during setup, changes in requirements, or software updates. Resolving configuration errors has often required manual intervention from system administrators or operators. This process can be time-consuming, error-prone, and disruptive to system availability. Configuration errors in Kubernetes can lead to various issues and challenges, including (1) application failures. Incorrect configurations can cause applications not to start, crash, or behave unexpectedly, leading to downtime and degraded performance. (2) Security vulnerabilities. Misconfigured security settings can expose sensitive data or grant unauthorized access to resources, potentially compromising the entire cluster. (3) Resource wastage. Poorly configured resource requests and limits can lead to inefficient resource utilization, affecting both performance and cost. (3) Scaling problems. Inaccurate scaling configurations can prevent applications from automatically adjusting to changes in demand, affecting responsiveness and user experience. (4) Operating overhead. Managing complex configurations across multiple applications and environments can become a significant operational burden for DevOps teams.

2.2 Self-Healing

Self-healing is closely associated with autonomous computing to develop a system that could manage itself without human intervention. Self-healing equips a system with the ability to automatically resume its activities in the case of a malfunction through three steps: the process of detecting failures, diagnosing, and performing remedial operations. The primary objective of incorporating self-healing functionality is to enhance the stability and maintainability of

the system. To achieve that, it is essential to have an autonomous system with self-healing capabilities that can identify and address faults in real-time without human intervention.

In the context of configuration errors, self-healing mechanisms aim to detect misconfigurations and restore the system to a working state. Self-healing systems typically rely on continuous monitoring to detect deviations from the expected configuration. This can include tracking performance metrics and system states or comparing the current configuration with a predefined proper configuration. Once a configuration error is detected, the self-healing system should have automated routines or scripts in place to correct the issue. This may involve reverting to a previously known good configuration, adjusting settings, or implementing workarounds to reduce the need for manual intervention, saving time and resources.

2.3 Markov Decision Processes

Markov Decision Processes (MDPs) is a mathematical framework used in the field of reinforcement learning and decision-making under uncertainty. MDPs are widely used in various areas, including artificial intelligence, economics, operations research, and robotics, to model and solve problems where an agent interacts with an environment to achieve a goal. The goal of solving an MDP is to find a policy which is a strategy or a mapping from states to actions. The policy determines the agent's decision-making at each state, with the aim of maximizing the expected cumulative reward over time.

MDPs provide an efficient method to perform inference over time given non-deterministic action effects (e.g., scaling a deployment or updating a service) due to various factors like network latency, application-specific behavior, resource constraints, competition among workloads, configuration errors, unpredictable cluster behavior, and variable response time. These factors of uncertainty can be directly modeled by MDPs to adapt to changing conditions and to search for optimal policies that lead to suboptimal cluster behavior.

2.4 Previous Work

Our self-healing controller [13] consists of (1) state space that represents the current configuration and health status of the clusters. This includes information on pods, services, nodes, resources, and edge devices. (2) The action space represents the recovery operations that can be performed to address configuration errors. Actions can include terminating, reconfiguring, redeploying, and restarting containers. If the action is applied successfully, the controller marks the component as recovered and keeps a profile for the applied actions to enhance the recovery procedure in the future. If the applied action did not enhance the anomalous behavior of the misconfigured state (e.g., network congestion), the controller applies another action from the recovery action space defined for that type of misconfigured state. For each type, the controller follows a set of predefined recovery steps. Based on the actions applied, the observed observations, and

the rewards obtained from the applicable actions, the controller continuously updates the recovery policy to learn an optimal policy that maximizes the expected cumulative long-term reward received during the recovery process. The policy is used by an agent that decides and executes actions to rectify the configuration errors automatically. The performance of the recovery is periodically evaluated, and the reward function, action space, and other parameters are fine-tuned to improve the system's ability to handle configuration errors effectively.

The controller aims to recover configuration errors and mitigate their impact on workload (overloaded resources) to prevent the spreading of anomalous behavior to the whole system. We focus on common configuration errors [12] in Kubernetes, Azure, and Docker Swarm reported in 2022 and 2023 by CVE, the National Institute of Standards and Technology (NIST) SP 800-190, OWASP Container Security Verification Standards, OWASP Kubernetes Security Testing Guide and OWASP A05:2021 – Security Misconfiguration.

We adopted the mechanism of MDP in the self-healing process of configuration error for the following reasons: (1) It is ideal for optimization problems through reinforcement learning. (2) It allows us to manage the uncertainty of system performance while reducing the total number of actions required by allowing multiple components to select the same actions. (3) It allows the controller to make decisions about workload placement, load balancing, and failover strategies across clusters. For instance, when performance evaluation metrics, such as utilization, latency, response time, network congestion, and throughput, are not met, the controller applies a recovery by replacing or reconfiguring the settings of edge devices and container-based clusters to dynamically optimize workloads in a cluster based on some rules or performance objectives. (4) It helps in customizing fault tolerance strategies beyond Kubernetes defaults. For example, determine whether to tolerate transient pod failures or take more aggressive actions, such as relocating pods to other nodes or scaling up replica sets in response to recurring failures. (5) It allows the controller to control the rate of traffic shifting, rollback decisions, and version promotion based on metrics such as error rates, latency, and user satisfaction to ensure safe and more controlled deployments, especially when rolling updates or canary deployments in Kubernetes.

3 Related Work

This section delves into the topic of misconfiguration recovery in the literature.

Several frameworks have been created to handle the workload and information flow in Edge/Fog environments [7], [16], [5], [3]. However, these frameworks have limitations in integrating different policies to dynamically manage the configurations of edge devices and clusters. The existing frameworks have not paid enough attention to the importance of efficient recovery management. In [11], the study uses principal component analysis to identify misconfigured cloud services by analyzing the relationship between service behavior and performance metrics. In [4], a protocol is presented to reconfigure cloud applications by addressing failures in interconnected components hosted on remote VMs.

In [2], a performance-centric configuration framework for containers in Kubernetes is proposed. The framework gives unified key-value data, including configurations and metrics, to analysis plugins by providing a built engine for processing defined rules in analysis plugins. A performance-centric configuration framework for managing containers within Kubernetes is introduced. This framework offers a unified repository of key-value data, encompassing both configurations and metrics, and is made accessible to analysis plugins through a purpose-built engine designed for processing predefined rules within these plugins. In [1] work, a dynamic and reconfigurable workflow framework is presented, to effectively handle failures caused by unavailable resources and fluctuations in service quality. This resilient framework is capable of recovering from such failures within long-running workflows, either through human intervention or the execution of predefined tasks. In a similar way [17] study introduces an architecture designed to govern the behavior of applications deployed in cloud environments through the utilization of a set of meticulously defined rules. These rules are instrumental in orchestrating the creation and configuration of virtual machines. Moreover, this architectural design empowers flexible redefinition of policies at both the service and virtual machine levels, allowing for a comprehensive description of their respective behaviors. In the study by [11], the authors identify a crucial link between service behavior and specific performance metrics. They employ principal component analysis to uncover instances of misconfiguration within cloud services. The work in [4] presents a protocol focused on reconfiguring cloud applications. This protocol primarily targets the adjustment of interconnected components hosted on remote virtual machines. In [2] a notable performance-centric configuration framework tailored for containers within Kubernetes is introduced. This framework facilitates the provision of unified key-value data, encompassing both configurations and metrics, to analyze plugins. It accomplishes this by offering a dedicated engine to process predefined rules within these analysis plugins. In the work of [1], a dynamic reconfigurable workflow framework is presented, specifically designed to manage failures arising from unavailable resources and variations in service quality. In particular, this framework exhibits the capability to autonomously recover from failures in long-running workflows, either through human intervention or by executing predefined tasks. In [6] Farooq et al. presented a fault predictive framework that is based on a developed model and subsequent analysis considering a substate as a middle state between the optimal operation state and system failure; however, it does not take into account the effect of several actions that can be done in the state. In [17], an architecture is proposed to govern the behavior of applications deployed in cloud environments. This architecture leverages a set of well-defined rules to orchestrate the creation and configuration of virtual machines (VMs). Furthermore, it empowers the flexible redefinition of policies at both the service and virtual machine levels to comprehensively describe their respective behaviors. Palacios et al. [10] conducted a study that focused on the selection of performance indicators as input to the network. Their proposal introduced a supervised technique that automatically selects indicators to support

self-healing functions. The approach involves analyzing the statistical behavior of indicators in various states of the network to evaluate their differences.

Our work aligns with the concepts discussed above. However, we expand on the ideas presented in [15], [14] by developing a controller that extends their principles. This controller effectively correlates observed performance degradation (failures) with underlying abnormal information flows (faults) and identifies the type of misconfiguration (error) within IoT edge devices and a cluster of containers operating on cluster nodes [12]. Through this mapping process, the presented controller takes on the crucial task of rectifying misconfigurations. It does so by determining the optimal recovery action or policy that maximizes the performance of the system (rewards) and the reliability within the observed environment.

4 Evaluation

This section assesses recovery, focusing on the measurement of performance and security. This section extends the recovery evaluation by training the controller on a newly obtained dataset.

4.1 Environment Setup

The system was managed by a virtual machine running on a Linux OS (Ubuntu 18.10 version), with one VCPU and 2GB of VRAM. The virtual machine was assigned to a physical PC that had Windows 11, an Intel Core i7-1260P 2.10 GHz processor, and 32GB of RAM; more details in [13].

4.2 Data Collection and System Monitoring

A group of agents was installed to collect information on CPU, memory, network, and any changes made to the file system, without disrupting the flow of the system. The agent includes a data interval function to determine the time interval of the collected data. The agent removes outliers from the collected data and monitors them using selected resource performance monitoring tools. The agent is configured to connect to the system automatically with valid credentials for authentication.

We used the Logman command in Kubernetes and Docker to trace remote procedure call (RPC) events to forward container logs as event tracing in the window. We used NNM iSPI Performance to collect data about the information flow from the system under observation (e.g., device ID, device type, max/mean/min size of the packet sent, total packets, max/mean/min amount of time of active flow, duration of flow).

The configuration files of the components are stored in the GitOps version control to simplify the rollback of a configuration change. We wrote our configuration files using YAML. We managed the configurations, deployments, and dependencies using kubectl and Skaffold. We used Datadog to access a live data

stream of running components and capture request-response tuples and associated metadata. The data collected were grouped and stored in a time series database using Timescale-DB via Prometheus. The data collected was used to train the models and provide more targeted and specialized training data sets. The data was divided into 70% training data and 30% testing data. More details on environment evaluation settings can be found in [12].

4.3 Test Environment

Kubernetes clusters were created with and without configuration errors to test the behavior of clusters under different scenarios. Clusters were created using Kubeadm to set up clusters on our own machine. We created 1 master node and 3 worker nodes with a resource capacity for each node of 2 CPU cores and 4 GB RAM. We created Kubernetes deployment for a sample application called "test-scenario" by deploying 15 replicas of the application including the configuration error scenarios in the following section.

4.4 Configuration Errors Scenarios

The controller is trained to recover configuration errors [13] that can lead to a privilege escalation on the host [8], [9]. We focus on the configuration error scenarios that target cluster security settings and contribute to resource saturation.

The [8] error allows a malicious container to escalate privileges on the host machine when a user runs the exec command to execute an operation in a running instance of that container (e.g., a new container with an attacker-controlled image). To achieve that, we set the privileged setting to true to grant container-escalated privileges to execute commands inside the container using the kubectl exec command.

The [9] error allows malicious cluster containers to carry out man-in-the-middle (MitM) attacks. Here, a malicious container can exploit this flaw through the use of rogue IPv6 (i.e., Internet Protocol version 6 that allows communication to take place over the network) router advertisements. This can redirect traffic to the malicious container by sending these advertisements to the host or other containers. In this scenario, we created a network policy that allows all pods to communicate with each other, regardless of namespace boundaries. The policy allow-all is too permissive because it selects all pods with "$podSelector$: ", allowing any pod to communicate with any other pod. The policy does not specify any restrictions on IPv6 traffic, which means that if rogue containers in the cluster start using IPv6 to intercept or manipulate network traffic, there are no controls in place to prevent this.

These errors can consume many system resources, such as CPU, memory, and the network, causing slow requests and decreased request rates. To achieve that, we created the following scenarios.

- Resource Request Exceeds Node Capacity

  - for 7 out of 15 replicas of the "test-scenario" application, we set resource requests for CPU, memory, and network that exceed node capacity.
- Resource Limits Exceed Resource Requests
  - for 5 out of 15 replicas of the "test-scenario" application, we set resource limits that exceed the resource requests defined in the pod spec.
- Resource Request and Limit
  - for 10 out of 15 replicas, we omit resource requests and limits in the Pod spec entirely. Here, we specified resource requests and limits for the container to be 16 GB memory and 8 CPU cores for requests, and 32 GB memory and 16 CPU cores for limits so that the sum of resource requests and limits exceeds the available cluster resources.

For each scenario, (1) we monitor cluster health and resource utilization using Kubectl and Prometheus tools. (2) We observe the behavior of clusters, pods, and containers with misconfigured resources. (3) We check the cluster logs for error messages.

We created such settings to test how the controller will handle different misconfiguration scenarios with a reasonable degree of coverage, however, those settings can be configured (i.e., the number of replicas and error scenarios) based on the specific testing goals and the level of detail that we need to capture.

To make it easier to undo configuration changes, the component configuration files are saved in GitOps version control.

4.5 The Recovery Assessment

The Mean Time to Recover (MTTR) metric is used to evaluate the average recovery time to recover a misconfigured component after observing a failure in the monitored metrics (i.e., a component cannot meet its expected performance metrics). A higher MTTR indicates the existence of inefficiencies within the recovery process or within the component itself. We conducted two scenarios:

Optimal Policy Selection In this scenario, the controller tries to select the optimal policy using a reinforcement learning algorithm to learn the best actions to take at each error based on previous experiences and expected rewards, such as minimizing response time or CPU utilization. The optimal policy would guide the agent to make the best decisions at each error to enhance the performance of the system under observation. Optimal policy selection is typically favored when the goal is to maximize long-term rewards. Here, we define different policies such as (1) Policy for Minimizing Response Time (Policy 1), which is designed to quickly recover errors that directly impact the response times of a component. The policy prioritizes restarting or reallocating resources to ensure rapid responses. (2) Policy to Enhance CPU Utilization (Policy 2): This policy aims to optimize CPU utilization across the cluster, focusing on long-term efficiency. It might involve adjusting resource allocation and scaling decisions to maximize CPU usage, even if it does not provide immediate relief to response-time issues. The system continuously monitors server performance, response times, and CPU

utilization during and after the error recovery process. Based on the observation, the controller learns whether Policy 1 or Policy 2 was more effective for the specific error scenario. Over time, it adapts its decision-making to better align with the goal of either minimizing response time or enhancing CPU utilization based on the context. When response time issues are critical, the system prioritizes actions that directly benefit the user experience. When CPU utilization is the primary concern, the controller takes actions that lead to long-term efficiency gains, ensuring stable performance.

In the given environment, for the configuration error in [8], the Mean Time To Recover (MTTR) for the edge device was approximately 209 seconds. On the other hand, the MTTR for container recovery was approximately 224 seconds. However, for the configuration error in [9], the MTTR for the edge device was approximately 240 seconds. On the other hand, the MTTR for container recovery was approximately 250-273 seconds. In such settings, the time it takes to recover from a configuration error depends on several factors, such as the speed with which the error is detected, the identification time, how long it takes to deploy the recovery to the affected cluster(s), and ongoing monitoring. Assigning more rewards during the recovery process can decrease the average recovery time, as it shortens the detection time and improves the controller's performance. However, an increase in the failure rate can lead to longer recovery times needed to efficiently recover from failures.

Hence, the average redeployment time is measured for the component after observing anomalous behavior until a successful recovery of a component. The average container redeployment time was approximately 120-170 seconds, without observing overhead associated with Kubernetes and Docker Swarm. For the edge device, the average redeployment time required to send a redeployment request and to receive a response to the corresponding edge gateway successfully was approximately 47-250 seconds. In multiple runs, the average redeployment time was reduced by 16%, and performance improved by 54% depending on the content and structure of the container image and the available network bandwidth. The controller performance was almost the same, with a minor recovery time deviation of around 100 seconds for some error types, such as container-privileged access and incorrect pod labels. The deviation returned to the correlation with the failure of the system [12]. Hence, sequence of failures [13] is used to reflect the type of failure, which represents failures that share the same observations corresponding to a unique fault. If the container privileged access and wrong pod label sequence of failures occurred, the focus is on the container privileged access failure to represent its failure type and relate it to its fault, which is Privilege Access Escalation Management. In such settings, the occurrence of an initial failure is chosen because it is representative enough of the observations to which it belongs, allowing us to save recovery time without trying many recovery actions.

Random Policy Selection In this scenario, in certain situations, selecting a random policy could be useful for exploration or to add an element of uncer-

tainty. Here, instead of always selecting a fixed deterministic policy based on performance optimization, an element of randomness for certain errors could be used to improve fault tolerance and discover potentially better solutions such as scaling resources to reduce network congestion to improve resource allocation and cost optimization. When the random policy is selected, it makes resource scaling decisions randomly within predefined limits. For example, it might randomly increase the number of nodes during regular intervals, even if the optimal policy would not normally trigger such a scaling action. Here, the agent randomly selects one or more actions with a uniform distribution to explore or introduce variability when appropriate (e.g., select different actions while still considering the misconfigured component's past status). At specific, scheduled times, or with a defined probability, the system activates the random policy for a limited duration within predetermined limits. The controller continuously monitors the system performance and cost during the random policy activation. Based on feedback and data collected during activation, the controller learns whether random scaling actions have positive or negative effects. If random actions lead to cost savings without sacrificing performance or if they help distribute resources more evenly, the system may use them more frequently in the future.

In these settings, the MTTR for the edge device was 188 seconds in this scenario, while the MTTR for the container was approximately 301 seconds under the same conditions. However, both the container and the edge device were able to function normally after this period of time and their assigned rewards remained unaffected. Hence, to enhance the results delivered, we created a hierarchy of random policies that address different aspects of system performance and cost optimization. For example, separate random policies were created for network congestion, CPU utilization, memory management, and cost. Each policy would be activated based on specific conditions and objectives related to its area of focus. Each policy operates independently, monitoring the specific metric or condition relevant to its objective. When a policy's activation condition is met, it triggers a random action based on its focus area. Random actions introduce variability into resource allocation or scaling decisions within predefined limits. The system continuously monitors the performance, cost, and impact of these random actions. For example:

Network Congestion Policy: Objective: Minimize network congestion to ensure low latency and smooth user experience. Activation Condition: When network congestion exceeds a predefined threshold [13]. Random Actions: Randomly increase or decrease network bandwidth allocations for specific services or components to alleviate congestion.

CPU Utilization Policy: Objective: Optimize CPU utilization for cost savings and performance improvement. Activation Condition: When the CPU utilization for one or more virtual machines (VMs) consistently exceeds a specified limit. Random Actions: Randomly adjust the number of CPU cores allocated to virtual machines within predefined limits, redistributing resources to alleviate CPU bottlenecks.

Memory Management Policy: Objective: Manage memory usage efficiently to prevent out-of-memory errors and ensure stable performance. Activation Condition: When memory usage on specific servers or containers approaches critical levels. Random Actions: Randomly allocate additional memory to affected servers or containers, or initiate memory-clearing processes to optimize memory usage.

Cost Optimization Policy: Objective: Identify opportunities to reduce cloud infrastructure costs while maintaining acceptable performance. Activation Condition: Periodically, at scheduled intervals, or when cost metrics suggest potential resource savings. Random Actions: Randomly select virtual machines, containers, or services and adjust their instance types, scaling them up or down based on historical cost and performance data. Additionally, the policy could randomly pause or terminate instances during off-peak hours.

MTTR is measured at the edge and container levels. The results obtained indicated that at the edge level, the MTTR was approximately reduced by 20%, and at the container level, the MTTR was approximately reduced by 43% under the same conditions. The results indicated that by providing a flexible and adaptive recovery policy, the recovery process could be managed and optimized accordingly under various factors that have a direct impact on configuration errors, such as resource utilization and network congestion. As each policy within the hierarchy has its own parameters, thresholds, and activation conditions, the hierarchy of random policies allows us to adapt to these changing conditions by activating different policies based on specific configuration error cases and recovery objectives.

4.6 Discussion

The recovery phase was trained in a controlled environment with specific configuration errors to test the controller's performance and reliability. However, the recovery time not only varies from one configuration error to another, but also depends on several factors such as the complexity of the affected systems (e.g., the number of containers, their interdependencies, and the network configuration), the deployment time, which varies according to the system's change management process, the urgency of the vulnerability, time of error discovery and initial response, the scope of affected containers, the choice of container orchestration tools (e.g., Kubernetes, Docker Swarm) can impact recovery time, and resource redundancy such as backup containers or failover mechanisms, can minimize downtime and reduce recovery time.

The recovery time may be shorter or longer depending on factors unique to the system under test. Hence, it is important to prioritize proper detection, identification, testing, verification, and monitoring to ensure that recovery does not introduce new vulnerabilities or disruptions in the clusters. The recovery time depends on how quickly an organization can progress through those steps. In some cases, organizations with robust security practices and automation in place can recover relatively quickly, while others may take longer to address

vulnerability, especially if they face challenges such as complex infrastructure or regulatory requirements.

As the complexity of the system grows, a hierarchical approach (hierarchy of random policies) can be scaled effectively to minimize recovery time and enhance the recovery process. Here, new policies can be added to address emerging challenges or changing requirements to manage the risk associated with recovery actions more effectively. Each policy can be designed to collect data and feedback during its activation. This information can be used to assess the effectiveness of different recovery strategies, leading to data-driven decision-making and continuous improvement in recovery processes. For example, recovery policies focused on cost optimization can introduce variability in resource allocation decisions, allowing experimentation with resource scaling and allocation strategies to find the most cost-effective solutions. Instead of relying on a single deterministic optimal policy, the system can choose the most appropriate random policy based on the specific error conditions. This adaptability can lead to more efficient recovery processes and better resource utilization by ensuring that different aspects of performance are effectively addressed while also learning from past applied recovery actions to improve the future of recovery. In some cases, multiple errors occur simultaneously in a cluster. The random policies hierarchy allows for the activation and execution of multiple policies in parallel, addressing multiple errors concurrently rather than sequentially. This parallel processing can significantly reduce recovery time, especially in complex environments where the optimal recovery policy is not immediately obvious.

In summary, the recovery time for configuration errors is context-dependent and can vary widely. Organizations should aim to continuously improve their recovery processes, reduce MTTR, and ensure that they have the necessary resources, tools, and expertise to handle configuration errors efficiently.

5 Self-Healing of Configuration Error Open Research Directions

Self-healing in Kubernetes refers to the ability of the system to automatically detect and correct configuration errors and operational issues without human intervention. Recovering from configuration errors is a complex task, as it depends on the specific error and its impact on a cluster. Achieving a high level of self-healing capability is an ongoing research area, and the following represent some future research directions.

5.1 Advanced Machine Learning and AI

The use of machine learning and artificial intelligence techniques to develop predictive models can help to anticipate configuration errors and take corrective actions in advance. When a configuration error is predicted, the system can automatically apply corrective actions based on predefined or dynamically generated solutions. Reinforcement learning can help the system make better decisions in

response to configuration errors by enabling Kubernetes clusters to learn from their own actions and adapt to changing conditions in real-time. Combining information from different data sources, such as logs, telemetry data, and event streams, can continuously improve the accuracy of the learning process. A reinforcing feedback loop allows Kubernetes clusters to learn from their self-healing actions. If a particular action repeatedly fails or causes issues, the system should adapt and adjust its behavior accordingly so that the action does not introduce vulnerabilities or compromise the integrity of the system.

5.2 Cost-Effective Self-Healing

Automated recovery actions should not lead to excessive resource consumption or unnecessary costs. Integrating cost-benefit analysis into predictive models helps to determine the trade-offs between different remediation actions. For example, the system can evaluate whether the cost of a particular action is justified by the potential benefits. This could be approached by (1) identifying the cost metrics relevant to a Kubernetes environment such as infrastructure costs (e.g., compute, storage, network), operational costs (e.g., labor, maintenance), and potential business impact costs (e.g., revenue loss during downtime). (2) Define benefit metrics that quantify the optimistic results of remediation actions. This could include metrics related to system stability, performance improvements, reduced downtime, and improved user experience. (3) Establish cost-benefit thresholds or criteria that determine whether a remediation action should be initiated. For example, you might require that the expected benefits outweigh the associated costs by a certain margin. (4) Train machine learning models to consider cost and benefit factors when predicting and recommending remediation actions. These models should take into account the cost-benefit trade-offs for different potential actions. (5) Ensure that cost assessment is dynamic and adaptive to continuously evaluate cost and benefit factors in real-time as the Kubernetes environment changes. (6) Utilize optimization algorithms that can automatically determine the most cost-effective remediation actions based on the current cost-benefit landscape considering constraints and priorities. (7) Allocate costs accurately to different components, applications, or teams within the Kubernetes environment to understand the cost implications of specific actions on different parts of the system. (8) Establish a feedback loop that collects data on the actual benefits achieved by remediation actions. These data can be used to refine cost-benefit models and improve decision-making over time. (9) Implement robust cost monitoring and reporting capabilities to track the actual costs incurred by remediation actions. This helps validate the accuracy of cost predictions and allows for ongoing optimization.

5.3 Application Workloads

Different application workloads may have varying requirements for self-healing of configuration errors. Some applications may be resource-intensive, while others are lightweight. Self-healing mechanisms should consider resource constraints

and scale resources appropriately to prevent configuration errors related to resource exhaustion. Some applications may have unique policies for error handling and recovery (e.g., scaling policies) that are based on metrics such as traffic patterns, load, resource constraints, and user demand. Defining application-specific metrics should be aligned with the specific performance and health indicators of the application workload to trigger appropriate self-healing actions. Certain workloads may require performance optimization strategies as part of self-healing. For example, optimizing caching mechanisms in response to configuration errors that impact performance. In a multi-cluster environment, application workloads have various configurations and dependencies, self-healing should consider this variation and take action to restore them when configuration errors disrupt them. For example, customized rollback mechanisms could be developed for application workloads in the case of configuration errors introduced by updates or changes to protect data integrity and minimize data loss during workload failures. Hence, self-healing policies should provide customizable parameters to align with specific workload needs, as some applications may prioritize high availability, while others may prioritize data integrity or cost optimization. For example, high-availability policies might have different thresholds than cost-optimization policies. Another factor that could be considered is enabling the definition of resource allocation policies that consider both resource constraints and workload priorities. For example, certain workloads may prioritize CPU resources over memory. Machine learning models can assist in determining the most appropriate policies for different situations and could predict how different application workloads are likely to behave under various conditions. For example, machine learning models that capture the historical behavior of different application workloads could be developed. These models should consider factors such as resource utilization, performance metrics, and event logs to optimize performance for workloads, right-size resources, and manage cloud costs effectively. Learning models could be trained to detect anomalies in workload behavior and trigger proactive self-healing actions before issues become critical. Hence, by combining machine learning-driven workload behavior prediction with self-healing policies, Kubernetes can become more adaptive and responsive to the unique requirements of each application workload.

5.4 Policy-Driven Self-Healing

Policy-driven self-healing in Kubernetes involves defining and enforcing specific policies and rules for automated configuration error detection and correction. These policies could be based on compliance requirements, best practices, or specific application needs. Some future research directions in this area could be developing standard policy definition languages (e.g., domain-specific languages) for Kubernetes that allow administrators to express policies in a structured and uniform manner to promote interoperability and ease of policy management. Such policies can target specific aspects of Kubernetes resources, such as security, performance, resource utilization, and application-specific requirements. For example, dynamically adjusting resource quotas and limits according

to application-specific policies. Policy languages should be extensible, allowing organizations to add custom policies when needed, be compatible with various Kubernetes distributions, and include managed Kubernetes services offered by different cloud providers. They should be updateable without breaking the compatibility of the Kubernetes environment (e.g., integration with Kubernetes APIs). Machine learning could be integrated into policy-driven self-healing to allow policies to evolve based on observed patterns and behaviors within the Kubernetes environment to optimize policy settings over time. It helps in validating and testing written policies to identify policy errors and misconfigurations before they impact the cluster. The prediction mechanism of machine learning models can ensure that the policy enforcement process is predictable and coherent by early predicting the conflicts that may arise when multiple policies intersect or contradict each other, especially with the hierarchical policy structure. Here, machine learning can predict global and local policies that need to be customized, analyze historical policy configurations and resource utilization data, and suggest policy customization based on the characteristics of individual clusters or namespaces to automatically adjust policy parameters to optimize resource allocation. For example, recommending resource quota adjustments based on analyzing historical usage patterns within clusters or namespaces to determine if policies need to be adjusted. These recommendations can consider factors such as past performance, compliance requirements, and workload behavior. Here, a confidence score could help to understand the reasoning behind recommendations to adjust a specific policy. This score indicates the model level of certainty that the recommendation is appropriate. For example, if the threshold is set at 0.6, the model may recommend the policy if the score is greater than or equal to 0.6 which indicates greater confidence in the recommendation's success.

5.5 Blockchain Technology for Self-Healing Configuration Errors

Blockchain technology can be used to help recover configuration errors in Kubernetes by providing a decentralized tamperproof ledger of changes and configurations. Blockchain maintains an immutable ledger of all transactions and changes made to the system. Each time a configuration change is made in a Kubernetes environment, a record of that change is created. This record includes metadata such as the configuration changes made, who made them, and a timestamp. To ensure the immutability and integrity of these records, a cryptographic hash (e.g., SHA-256) or a specific version of the configuration at the time of the change is calculated or associated to uniquely represent the configuration at that moment. The hash/version and associated metadata can be stored on a blockchain. The blockchain serves as a ledger that keeps a record of all these hashes/versions and their associated metadata, making it extremely difficult to alter or delete previous configurations. This immutability ensures that a historical record of all changes is available for audit and recovery, which simplifies the process of identifying the root cause of a configuration error and determining which version of the configuration was in effect at a specific point in time. Here, smart contracts can be programmed to continuously monitor the state of the Kubernetes cluster

and the configuration settings by employing predefined rules or conditions to detect configuration errors or inconsistencies. These rules could include checks for resource constraints, security vulnerabilities, or policy violations. When an error is detected, the smart contract initiates the recovery process. The recovery process could be defining specific rollback procedures to revert the configuration to a known stable state when an error occurs. The rollback process may involve reverting to a previous configuration version, adjusting resource allocations, or applying specific patches. The smart contract should determine the appropriate rollback strategy based on the type and severity of the error. Beyond rollback, smart contracts can be designed to perform self-healing actions to correct errors automatically. This might include dynamically adjusting resource allocations, restarting malfunctioning containers, or triggering auto-scaling to handle increased loads. Self-healing actions should be tailored to specific configuration error scenarios and recovery objectives. To effectively execute recovery actions, smart contracts need to interact with external systems and tools, such as configuration management systems, monitoring solutions, APIs, or notification services. Here, governance mechanisms should be used to maintain and update smart contracts. For example, developing clear policies and guidelines for smart contract development, and updates. These policies should cover aspects such as security standards, coding practices, and compliance requirements. A smart contract versioning scheme policy (e.g., Major.Minor.Patch) could be used to track changes and updates. For example, major version updates could include changes that introduce backward-incompatible changes to the contract's interface, data structures, or behavior. These updates may require modifications to the contract's users and may entail a significant migration effort. Minor version updates include new features or functionality that are backward compatible with previous versions. These updates add to the contract's capabilities without breaking existing integrations. Patch version updates cover backward-compatible bug fixes, security patches, and minor improvements. They address issues without changing the contract's external behavior. The backward compatibility should be compatible with existing deployments and integrations and cover both new and existing functionality. By implementing versioning policies, organizations can maintain stability and trust in their smart contract, ensuring that changes and updates are managed in a structured and user-friendly manner.

It is essential to note that the self-healing technique should align with the specific requirements of the Kubernetes configuration management system. Here, implementing feedback loops is essential for evolving self-healing capabilities in Kubernetes clusters, as they enable Kubernetes clusters to learn from their self-healing actions (success or failure of self-healing actions), which is crucial for improving the overall reliability and efficiency of the system. Feedback can come from metrics, logs, user interactions, and other sources, which can be used to fine-tune machine learning and AI models and improve confidence estimations.

Research and development efforts in this area can significantly improve the reliability, resilience, and automation of Kubernetes configuration error recovery, reducing manual intervention and minimizing downtime in the event of errors.

5.6 Blockchain Powered multi-cluster and multi-tenancy Kubernetes

Multi-cluster and multi-tenancy Kubernetes represent a sophisticated and highly flexible orchestration paradigm within the containerized application management space. Multi-cluster Kubernetes refers to the deployment and coordination of multiple Kubernetes clusters, each one functioning as an independent orchestration environment. This architecture allows organizations to isolate and separate workloads, improving resource security and scalability. Multi-tenancy Kubernetes on this other hand refers to the same concept used in could computing, which is accommodating multiple tenants or user groups within a single Kubernetes cluster, with each tenant operating as if it has its dedicated Kubernetes environment. This approach optimizes resource utilization and operational efficiency by allowing the sharing of cluster infrastructure while ensuring security and isolation among tenants.

Both multicluster and multitenancy Kubernetes are pivotal in facilitating the dynamic, scalable, and efficient deployment of containerized applications across diverse computing environments, from on-premises data centers to cloud platforms, in a way that aligns with the modern principles of microservices architecture and agile software development practices.

These paradigms enable Kubernetes to operate on a larger scale, seamlessly accommodating the demands of diverse use cases, such as the integration with multiple cloud providers and the incorporation of IoT-based online resources. However, at the same time, this increases the complexity of the infrastructure and the management overhead.

In this context, blockchain technology becomes particularly relevant, especially when clusters are hosted in a cloud environment or a decentralized environment (e.g., IoT-based resources). Cloud and decentralized-based systems follow a shared responsibility model between users and cloud providers. In such complex and distributed ecosystems, the need for transparency, trust, and accountability is paramount, making the blockchain a valuable tool that can be used as a single source of truth.

In multicluster/multitenancy Kubernetes, users are responsible for managing their applications, configurations, and access controls within their own cluster; however, they rely on the cloud provider to guarantee the correct function of the underlying infrastructure. This shared responsibility model can cause concerns about data integrity, security, and potential misconfigurations. The blockchain can play a central role by recording the initial setup and subsequently every change that takes place in an immutable and transparent manner. This provides both users and cloud providers with a history of all activities within the environment, allowing for a clear audit trail and reducing disputes over accountability.

Additionally, the whole process can be implemented as a set of smart contracts to automate and enforce predefined rules and policies, ensuring that both users and cloud providers adhere to their respective responsibilities. For instance, in a multicluster Kubernetes setup, smart contracts can automatically verify that the user's configurations comply with security standards and policies, minimiz-

ing the risk of misconfigurations or security breaches. Additionally, blockchain technology inherently supports financial transactions with a high degree of efficiency and security. Its native design is centered on the concept of distributed ledgers, which record and validate transactions in a decentralized and transparent manner. This brings a huge benefit since smart contracts can facilitate real-time billing and auditing, streamlining financial transactions, and making it easier for users to monitor their usage and costs.

6 Self-Healing with Blockchain-Based Configuration Error: Use-Case

This section shows the feasibility of using the blockchain concept to enhance the controller self-healing mechanism for configuration errors.

6.1 Configuration Rollback use case

If there is a vulnerability due to misconfiguration, the blockchain keeps track of all clusters and tenants' configuration history. A self-healing mechanism can be employed to automatically detect and rectify misconfigurations while maintaining an immutable record of changes. In such a case, a private or consortium blockchain can be used to maintain an immutable ledger of configuration changes and errors. Smart contracts can be developed to define rules and conditions for configuration changes and recovery actions. When configuration changes are proposed (e.g., firmware updates, parameter adjustments), they are submitted to the blockchain network. The blockchain network validates the proposed configuration changes against predefined rules in smart contracts. If the changes are valid, they are approved and recorded on the blockchain. If a configuration change results in an error or unexpected behavior (e.g., a device becomes unresponsive), the smart contracts can trigger self-healing actions to autonomously take corrective actions. These actions may include restarting a specific service or component if a device experiences high CPU usage, or rolling back to a previously known good configuration. For instance, we can automatically activate and rollback to a known secure configuration. The rollback process could include validation mechanisms to ensure that the rollback is successful and that the system has returned to a secure state. Here, the rollback confirmation and audit logs are recorded on the blockchain for future transparency and reference. This feature helps to quickly recover to a secure state, reducing the potential exposure window to threats, reducing the need for manual intervention, and minimizing the possibility of human error during the recovery process. For example, when a system administrator attempts to modify the network security rules to open a port, the blockchain system checks the proposed change against a set of predefined rules. If the change is in compliance with security policies, it is approved and recorded on the blockchain. However, if the change violates the security policies, the blockchain system denies the change and reverts to the previous network configuration, maintaining the security posture of the system.

In addition, such a mechanism could be used at the IoT edge device level. Consider that we have a set of IoT edge devices (e.g., sensors) that are deployed with the ability to communicate with the blockchain network securely. These sensors are configured to collect data and communicate it to a central server. If a sensor experiences a configuration error that leads to data transmission failures, the blockchain can detect the error, trigger a recovery action (e.g., resetting the sensor's configuration to a known good state), and record the incident on the blockchain ledger. This self-healing process reduces data loss and ensures the reliability of the IoT network.

Using blockchain to recover from configuration errors reduces manual intervention and improves the up-time of clusters and IoT edge devices. However, it is important to consider the resource overhead required to maintain blockchain records and execute smart contracts.

6.2 Decentralized Threat Detection use case

The use of blockchains can provide a distributed and autonomous solution for managing security incidents. This involves monitoring security events, anomalies, and potential vulnerabilities across Kubernetes clusters, which are then promptly recorded as blockchain transactions. By decentralizing threat detection throughout the blockchain network, the controller recovery mechanism could be improved to detect and mitigate threats while also increasing security resilience. This distributed nature of the blockchain network makes it more challenging for adversaries to compromise or disable security measures since the detection and self-healing mechanism spreads over clusters. If one cluster is compromised, other clusters continue to operate autonomously. For example, in a multicluster environment, if a security vulnerability is detected in one cluster, the monitoring agent could create a blockchain transaction to record the incident (e.g., the nature of the threat, timestamp, and affected resources). Here, a smart contract could be used to analyze the transaction based on predefined rules to trigger a self-healing mechanism. This mechanism can involve isolating vulnerable pods, applying a security patch, and restarting affected workloads. In such settings, the blockchain ledger maintains a record of the incident and the actions taken, ensuring transparency and auditability.

In the end, we provided these use cases to show how blockchain technology, combined with self-healing mechanisms, can enhance the security management process in clusters while distributing security controls across the network for increased resilience.

7 Conclusions and Future Work

A self-healing controller was introduced as a promising solution to address the performance and security vulnerabilities associated with containerized cluster systems and edge devices arising from configuration errors. The paper sheds

light on the controller recovery process, which takes advantage of the ability of a Markov Decision Process (MDP) to identify optimal recovery actions or policies that maximize performance metrics.

Using the capabilities of MDP, the recovery of configuration errors enables adaptive recovery decisions that take into account the uncertainties and complexities inherent in these dynamic environments. As a result, we anticipate significant improvements in resource utilization and cluster performance, leading to enhanced efficiency and reliability.

The objective of this paper is to extend the experiments carried out on the self-healing controller [13]. The results presented in this paper show the efficacy of the proposed self-healing controller. Through various evaluations, we have observed remarkable enhancements in both performance and reliability for containerized clusters and edge devices under observation.

In the future, our goals are (1) to integrate the controller with Kubernetes to automate various tasks related to the recovery of configuration errors and to meet the specific requirements of applications by managing resource utilization and cost. (2) handle different types of configuration errors, regardless of being predefined previously. (3) automatically validate Kubernetes configuration files for correctness, security, and compliance considering the error line number.

References

1. Assuncao, L., Cunha, J.C.: Dynamic workflow reconfigurations for recovering from faulty cloud services. vol. 1, pp. 88–95. IEEE Computer Society (2013). https://doi.org/10.1109/CloudCom.2013.19
2. Chiba, T., Nakazawa, R., Horii, H., Suneja, S., Seelam, S.: Confadvisor: A performance-centric configuration tuning framework for containers on kubernetes. pp. 168–178 (2019)
3. Dass, S., Namin, A.S.: Reinforcement learning for generating secure configurations. Electronics 2021, Vol. 10, Page 2392 **10**, 2392 (9 2021). https://doi.org/10.3390/ELECTRONICS10192392, https://www.mdpi.com/2079-9292/10/19/2392/htm https://www.mdpi.com/2079-9292/10/19/2392
4. Durán, F., Salaün, G.: Robust and reliable reconfiguration of cloud applications. Journal of Systems and Software **122**, 1339–1351 (12 2016). https://doi.org/10.1016/j.jss.2015.09.020
5. Fairwinds: Kubernetes benchmark report security, cost, and reliability workload results (2023), https://www.fairwinds.com/kubernetes-config-benchmark-report
6. Farooq, H., Parwez, M.S., Imran, A.: Continuous time markov chain based reliability analysis for future cellular networks. In: 2015 IEEE Global Communications Conference (GLOBECOM). pp. 1–6. IEEE (2015)
7. Mascellino, A.: Nearly one million exposed misconfigured kubernetes instances could cause breaches (2022), https://www.infosecurity-magazine.com/news/misconfigured-kubernetes-exposed/
8. NVD: Cve-2019-5736 (2019), https://nvd.nist.gov/vuln/detail/CVE-2019-5736
9. NVD: Cve-2020-10749 (2020), https://nvd.nist.gov/vuln/detail/cve-2020-10749
10. Palacios, D., de-la Bandera, I., Gómez-Andrades, A., Flores, L., Barco, R.: Automatic feature selection technique for next generation self-organizing networks. IEEE Communications Letters **22**(6), 1272–1275 (2018)

11. Pranata, A.A., Barais, O., Bourcier, J., Noirie, L.: Misconfiguration discovery with principal component analysis for cloud-native services. pp. 269–278. Institute of Electrical and Electronics Engineers Inc. (12 2020). https://doi.org/10.1109/UCC48980.2020.00045
12. Samir, A., Dagenborg, H.: A self-configuration controller to detect, identify, and recover misconfiguration at iot edge devices and containerized cluster system. pp. 765–773 (2023). https://doi.org/10.5220/0011893700003405, https://owasp.org/www-project-kubernetes-security-
13. Samir, A., Dagenborg, H.: Self-healing misconfiguration of cloud-based iot systems using markov decision processes. pp. 244–252 (2023). https://doi.org/10.5220/0011966700003488
14. Samir, A., Pahl, C.: A controller architecture for anomaly detection, root cause analysis and self-adaptation for cluster architectures. pp. 75–83 (2019), https://www.researchgate.net/publication/333235851
15. Samir, A., Pahl, C.: Autoscaling recovery actions for container-based clusters. Concurrency and Computation: Practice and Experience **33**, 1–13 (12 2020). https://doi.org/10.1002/CPE.5955, https://onlinelibrary.wiley.com/doi/full/10.1002/cpe.5955 https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5955 https://onlinelibrary.wiley.com/doi/10.1002/cpe.5955
16. Taft, D.K.: Armo: Misconfiguration is number 1 kubernetes security risk (2022), https://thenewstack.io/armo-misconfiguration-is-number-1-kubernetes-security-risk/
17. Vaquero, L.M., Morán, D., Galán, F., Alcaraz-Calero, J.M.: Towards runtime reconfiguration of application control policies in the cloud. Journal of Network and Systems Management **20**, 489–512 (12 2012). https://doi.org/10.1007/s10922-012-9251-3