Self-Healing Misconfiguration of Cloud-Based IoT Systems Using Markov Decision Processes

Areeg Samir[a] and Håvard Dagenborg[b]
Computer Science Department, The Arctic University of Norway, Tromsø. Norway

Keywords: Markov Decision Process, Self-Healing, Recovery, Performance, Threat, Edge Computing, Cluster, Containers, Medical Devices, Multi-Cluster.

Abstract: Misconfiguration of IoT devices and backend containerized-cluster systems can expose vulnerable areas at the network level, potentially allowing attackers to penetrate the network and disrupt workload and the flow of data between system components. This paper describes a self-healing model based on a Markov decision process that can recover the misconfiguration and its impact on the workload and data flow at the network level. The results show that the proposed controller led to accurate results in performance and reliability.

1 INTRODUCTION

Misconfiguration might emerge when essential security settings of in edge devices and system clusters parameters are either not implemented or implemented with errors such as leaving the default configuration settings unchanged, erroneous configuration changes, or other technical issues (Samir and Dagenborg, 2023) that impact the workload and data flow within and across systems. Multi-layered vulnerabilities can leave devices vulnerable and allow unauthorized access. Thus, the change in workload and data flows due to misconfiguration might result in poor quality assessment, ineffective resource utilization, latency, high cost, and a decline in service quality.

Several works have looked at the management of workload and information flow (Moothedath et al., 2020), (Kraus et al., 2021), (Nie et al., 2021b), (Nie et al., 2021a), (Tang et al., 2018), (DCMS, 2018), (Jin et al., 2022), (George and Thampi, 2019). However, more work is needed to recover the misconfiguration of edge devices and containerized clusters and to optimize the system's performance and reliability (Dass and Namin, 2021), (Mascellino, 2022), (Rahman et al., 2023), (Wang et al., 2018).

In this paper, we proposed a self-healing controller that fixes the misconfiguration of edge devices and backend containerized services to mitigate its impact on the workload and the data flow. The proposed controller is based on Markov Decision Processes (MDPs), which have been shown useful for a wide range of optimization problems via reinforcement learning. We chose MDP to manage the uncertainty of the system's performance at a particular time and to allow multiple components to select the same actions to reduce the total number of actions required. Recovery occurs by replacing or reconfiguring the edge devices and containers-based cluster settings when they do not meet the performance evaluation metrics (e.g., utilization, latency, response time, network congestion, and throughput). The controller applies the recovery based on the type of misconfiguration (Samir and Dagenborg, 2023), considering the selection of the optimal recovery policy.

The rest of the paper is organized as follows. Section 2 discusses the misconfiguration under cluster and edge device levels in our system. Section 3 introduces the self-healing controller and describes the mechanism of finding the optimal recovery policy. Section 4 evaluates the controller. Section 5 discusses the related work. Section 6 concludes the paper and presents the direction of future work.

2 OVERVIEW OF THE SYSTEM MODEL

We consider IoT-enabled systems that consist of various edge devices that connect via the Internet (e.g.,

[a] https://orcid.org/0000-0003-4728-447X
[b] https://orcid.org/0000-0002-1637-7262

Samir, A. and Dagenborg, H.
Self-Healing Misconfiguration of Cloud-Based IoT Systems Using Markov Decision Processes.
DOI: 10.5220/0011966700003488
In *Proceedings of the 13th International Conference on Cloud Computing and Services Science (CLOSER 2023)*, pages 244-252
ISBN: 978-989-758-650-7; ISSN: 2184-5042

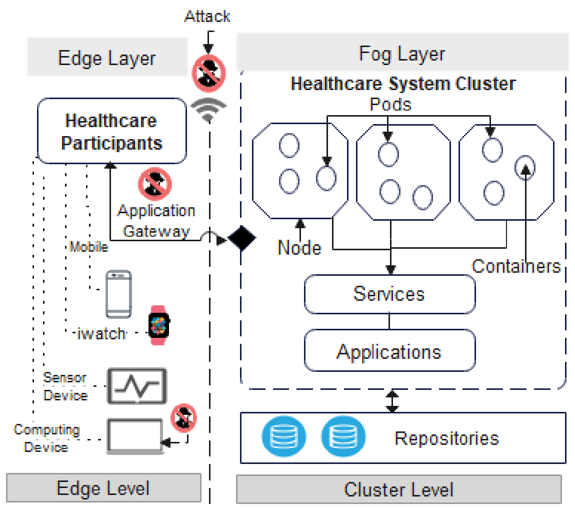

Figure 1: An example of a multi-cluster architecture for edge-based fog computing.

5G or WiFi) to backend clusters with nodes that contain one or more services. For instance, in the healthcare industry, heart monitoring software might be deployed to a container on a backend server, while medical sensors and mobile applications run on the edge to collect health metrics from patients. Misconfiguration makes such systems and edge devices prone to several attacks (e.g., distributed denial of service attacks, ransomware, security compliance breaches, privilege escalation attack) that facilitate further breaches, stress the system clusters and devices, and degrade system performance. A network gateway could provide the starting point to launch malicious activities that affect the system's resources. In our medical example, see Figure 1, the cluster contains a pool of nodes, pods, and containers that provide resources for the healthcare system. The cluster includes the healthcare application, which resides in the Docker Hub repository to share and access containers' images. The system includes services for its control, continuous integration, and deployment. To configure Kubernetes, configuration files are used to describe clusters, users, and contexts. They are stored in version control before being pushed to the cluster to simplify the configuration change and aid cluster re-creation and restoration. Misconfiguration might occur at different levels; at the edge device level, misconfiguration (e.g., CVE-2019-6538) could affect the scores of the heart rate monitor (e.g., Medtronic device) that allows a nearby attacker to change the settings of a patient's cardiac device by manipulating radio communications between it and control devices (inject, replay, modify, and intercept the telemetry data to reprogram the cardiac device). At the cluster level, misconfiguration (e.g., CVE-2019-5736, CVE-2022-0811) might occur due to network rules, root/less privileged user access, wrong pod label specifications, manual errors like typos, or forgetting to enforce network policies after writing them. Hence, misconfigured might allow an adversary to exploit container processes.

The controller in our system model aims to recover such misconfiguration and mitigate their impact on workload (overloaded resources) and prevent sequences of abnormal data flow paths. We focused on the common misconfiguration at Kubernetes, Azure, and Docker Swarm (Samir and Dagenborg, 2023) 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 adopt the Monitor, Analysis, Plan, Execute, and Knowledge (MAPE-K) architecture for self-healing systems as shown in Figure 2. Our controller consists of (1) *Monitor* that collects the performance and the configurations of edge devices and cluster-based nodes and containers; (2) *Analysis* that detects misconfiguration and identifies its type to apply a suitable recovery action. The detection and identification are not the scopes of this paper; more details in (Samir and Dagenborg, 2023); (3) *Plan and Execute* that selects the optimal recovery policy from a set of recovery actions to heal the misconfiguration and its impact on workload considering the provision of system resources. The recovery actions are stored in the Knowledge storage to keep track of the number of applied actions to the anomalous component. We created pre-defined misconfiguration description profiles with common misconfiguration types and stored them in the knowledge storage to be used in the recovery process.

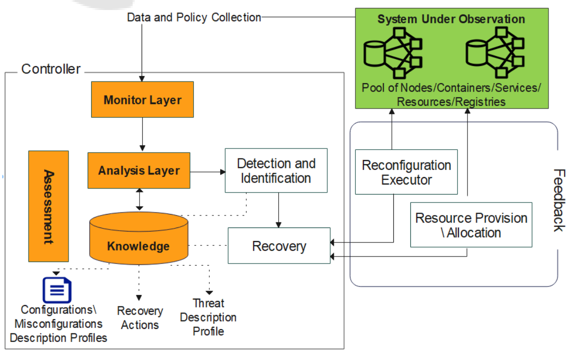
Data and Policy Collection
System Under Observation
Controller
Monitor Layer
Pool of Nodes/Containers/Services/ Resources/Registries
Assessment
Analysis Layer
Detection and Identification
Reconfiguration Executor
Feedback
Resource Provision \ Allocation
Knowledge
Recovery
Configurations\ Misconfigurations Description Profiles
Recovery Actions
Threat Description Profile

Figure 2: System model.

3 SELF-HEALING CONTROLLER

In this section, the self-healing controller is presented that selects the optimal recovery policy with the most proper recovery action(s) for the containers-based cluster or edge device according to its status. The following extends our previous controller (Samir and Pahl, 2019), (Samir and Pahl, 2020) to detect and identify the misconfiguration at the edge device(s) and containers-based cluster (Samir and Dagenborg, 2023).

3.1 Recovery Process Assumption

Our controller consists of the agent, which is a learner or decision-maker, and the environment, which is everything surrounding the agent (system under observation). In our case, the agent represents the Recovery phase and the environment reflects both the Kube Nodes (nodes, container, services) and the patient's edge device. The controller, at the recovery phase, chooses an action according to the misconfiguration types identified in (Samir and Dagenborg, 2023) and observes the performance of the environment after applying the action as shown in Figure 3. Then, corresponding to the applied action, the controller receives a reward, which refers to the performance enhancement for the observed monitoring metrics. If the action is applied successfully, the controller marks the component as recovered and keeps a profile for the applied actions in the knowledge to learn the optimal action at each state in the environment and to enhance the recovery procedure in the future. Here, the state refers to clusters, nodes, containers, services, or edge devices. However, if the applied action didn't enhance the anomalous behavior of the misconfigured state (e.g., network congestion), the controller applies another action from the recovery actions list defined for that type of misconfigured state through the Reconfiguration Executor. For each type, The Executor follows a set of configuration steps predefined and stored in the Knowledge storage. Based on the applied actions, observations, and rewards obtained from the applicable actions, the controller continuously updates the recovery policy to find an optimal policy that maximizes the expected cumulative long-term reward received during the recovery process.

3.2 Recovery Actions

The applied action could be one or more possible actions, denoted as a_i, such as terminate, reconfigure, redeploy, restart, or do nothing. Here, terminate is represented as $a_{i=1}$, which denotes that the system's

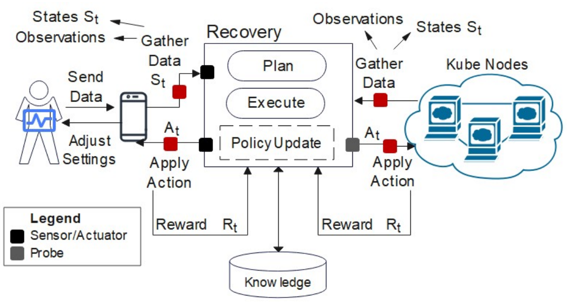

Figure 3: Self-Healing Controller - Recovery Using MDP.

state (e.g., containers deployed services) will be terminated if a container escapes vulnerability and allows an attacker to obtain host root access, and the recovery actions don't heal the anomalous behavior. The $a_{i=2}$ represents a combination of two actions, reconfigure and redeploy, and it indicates that the anomalous state will be reconfigured and redeployed. In case the $a_{i=2}$ action didn't heal the anomalous state, another action, $a_{i=3}$, will be applied to reconfigure and restart the whole cluster. If the status of the state cannot be obtained, do nothing action $a_{i=4}$ will be applied. The restart action $a_{i=0}$ is the default applicable action to avoid trying multiple actions and to minimize the time and cost of the recovery process.

3.3 Recovery States: Environment

For each misconfigured state, denoted as s, an action a_i from the action space A at time t where A contains a set of actions available for the misconfigured state, $A_t \in A(s_t)$. The misconfigured state s at a specific period constitutes the controller's input, which belongs to a state space S that includes possible situations for each misconfigured state: 1) the state is at the initial status (misconfigured) s_{Int}, 2) the state is successfully recovered s_{SR}, 3) the state recovery failed s_{RF}, and 4) the state is recovered s_R. Hence, if the state is s_{Int}, then a recovery action from the A is applied. In case the action is applied successfully, the state transits to s_{SR} to indicate the success of recovery, and the state is marked as recovered s_R; otherwise, the state turns to s_{RF}, and another action is applied. The controller monitors the performance evaluation metrics after applying the action. Whenever the metrics are beyond the dynamic threshold, the controller considers it a recovered state; otherwise, a recovery process will be continued until the misconfigured state is recovered. For each applicable action on state s, the controller receives a reward r at time t, where $r \subset R$, and R is the cumulative reward. The recovered state s_R returns to its optimal status at s_{t+1}. The transition from one state status to another, denoted as $P_s^a = P(s_{SR}, s_R, a_{i=2}) = 1$, and referred to the probability of transiting from state

s_{SR} to state s_R upon applying action $a_{i=2}$ and receiving reward r_s^a. If the state recovery failed s_{RF}, the probability of transition $P_s^a = P(s_{Int}, s_{RF}, a_{i=2}) = 0$, and we assign a constant failure rate CFR for the state under recovery with an exponential failure distribution as shown in (1) to measure the conditional probability of failure per time unit.

$$F_{(t)} = CFR \times exp^{-CFR(t+t')} \quad (1)$$

3.4 Recovery Rewards

The reward could be a positive value, which refers to a successfully applied recovery action that enhanced the observed metrics, or a negative value which refers to an unsuccessfully applied recovery action that declined the observed metrics. The value of the reward is determined by measuring the performance for every anomalous state based on the action applied at each time slot. We obtained the number of rewards for each anomalous state, as shown in (2).

$$r_s = [r_{t+1} \mid s_t] \quad (2)$$

To optimize the controller rewards, we computed the total amount of rewards, as shown in (3), received by the controller at time t while performing action $a_{i=2}$ to transits from state status (s_{SR}) to (s_R).

$$R(t) = r(t+1) + r(t+2) + \cdots + r(T) \quad (3)$$

For the recovery failed s_{RF} state status, the controller applies recovery actions until the state is recovered and the controller transits to (s_R). However, if the s_{RF} can't be recovered by the applicable actions, the controller enters into a loop as it isn't able to transit to (s_R), and a self-transitions happens to the (s_R) with zero rewards. Thus, to avoid infinity, we defined a discount factor DF to give immediate or future rewards according to the status of the s_{RF} state so that the controller could exit the loop of the anomalous state and flag the state as not-recovered. The optimal value for DF lies between 0.2 to 0.8 so that the sum of the rewards received at time steps u over the future is maximized according to a discount rate DF that determines the value of future rewards as shown in (4).

$$R(t) = \sum_{u=0}^{T-t-1} DF^u r_{t+u+1} \quad (4)$$

3.5 Appropriate Actions Selection

To select the proper recovery action for each state in terms of the rewards, we expanded (4) so that the controller follows a recovery policy π to take action a in state s at time t, as shown in (5), with a probability distribution over the actions taken for each state as shown in (6).

$$R^\pi(s,a,t) = [\sum_{u=0}^{T-t-1} DF^u \, r_{t+u+1} \mid S_t = s, \, A_t = a] \quad (5)$$

$$\pi(a|s) = \rho[A_t = a|S_t = s] \quad (6)$$

From (5) and (6), the controller selected action a_2 as a suitable recovery option for s_2 as the greater the value of $R^\pi(s,a,t)$, the better a specific action for a state is, as shown in Table 1.

Table 1: State Action Value.

Misconfigured Components	s2	
Actions	a1	a2
Values	0.3	0.8

From (5) and (6) given any state s and action a at t, we computed the probability of each possible pair of the next state s' and reward r as shown in (7). Here, the controller responds at $(t+1)$ to transit to the next state and reward. Given that we obtained: (1) the reward for any state action pair as shown in (8), (2) the state transition probabilities as shown in (9), and the expected rewards for the state action, next state as shown in (10).

$$p(s',r|s,a) = Pr\{R_{t+1} = r, S_{t+1} = s'|S_t = s, A_t = a\}, \quad s \in S_t, a \in A_t \quad (7)$$

$$r(s,a) = \varepsilon[R_{t+1}|S_t = s, A_t = a] = \sum_{r \in R} r \sum_{s' \in S} p(s',r|s,a) \quad (8)$$

$$p(s'|s,a) = Pr\{S_{t+1} = s'|S_t = s, A_t = a\} = \sum_{r \in R} p(s',r|s,a) \quad (9)$$

$$r(s,s',a) = \varepsilon[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s'] = \sum_{r \in R} r * p(s',r|s,a)/p(s'|s,a) \quad (10)$$

For each state s_i, we measured the average recovery time ART, which refers to the mean time μ of applying a specific action a_j under a particular policy π knowing the monitored metric type $MoM_{\{type\}}$ (e.g., type: CPU, Memory, Network) that exists in the performance function Per and belongs to the state s as shown in (11). Hence, the performance of the recovery process for a specific state is the overall performance for recovering a state s during the whole recovery process at time T with state probability matrix $P[s_i]$.

$$ART = \sum_{i=1}^{N} P[s_i] \sum_{j=1}^{M} \mu^{i}_{t,a_j} \times \pi(a_j|s_i), \quad \forall\, MoM_{\{CPU\}} \exists\, Per, \in s_i \tag{11}$$

3.6 Find the Optimal Recovery Policy

To find the optimal recovery policy for each state considering the recovery actions, we considered the policy that achieves more rewards than other policies and optimizes the performance over time as follows:

3.6.1 Policy Selection and Evaluation

We assumed that a policy denoted π is defined to be better than or equal to another policy denoted $\pi^{'}$ if the value of state s under the policy π denoted $\upsilon_{\pi}(s)$ is the expected return for taking action a in state s and is greater than or equal to $\pi^{'}$ for all states and actions under that policy. We measured the expected return from each state under a given policy by obtaining the transition probabilities and the expected reward r for applying action a in state s under a given policy π and a discount factor value DF of the next state $s^{'}$ as shown in (12) and (13). Then, we obtained the maximum return overall policies that can be achieved at any state s by obtaining the optimal state value as shown in (14) and the optimal state-action value as shown in (15). A policy whose states and actions values are optimal is an optimal policy; we can find that by taking their maximum as shown in (16). These steps are repeated for each state-action pair that belongs to the state space and action space under a selected policy to evaluate the selection of the optimal actions in a state to maximize $Q_{optimal}(s,a)$. The evaluation is stopped when the convergence between the old and the new values is small.

$$\upsilon_{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s^{'},r} p(s^{'},r|s,a)[r + DF\,\upsilon_{\pi}(s^{'})] \tag{12}$$

$$Q_{\pi}(s,a) = \sum_{s^{'},r} p(s^{'},r|s,a)[r + DF\,\upsilon_{\pi}(s^{'})] \tag{13}$$

$$\upsilon_{optimal}(s) = max_{\pi}\upsilon_{\pi}(s), \forall\, s \in S \tag{14}$$

$$Q_{optimal}(s,a) = max_{\pi}Q_{\pi}(s,a), \forall\, s \in S, \forall\, a \in A \tag{15}$$

$$\pi^{optimal}(s) = arg\ max_a\ Q_{optimal}(s,a) \tag{16}$$

3.6.2 Policy Update

We update the optimal policy obtained from the previous steps to select the action a that maximizes the future rewards if we take action a in state s and follow policy π as shown in (17). The policy $\pi^{'}$ must be better than the previous policy $\pi^{optimal}$, which indicates that $\pi^{'}$ improves the chances of getting more future rewards starting from the state s as shown in (18). The policy update stops when the policy stops improving, as shown in (19), or when the optimal policy obtained from the previous section $\pi^{optimal}(s)$ is better than the $\pi^{'}(s)$, in such a case, we consider $\pi^{optimal}(s)$ is the best policy that maximizes the future rewards for the state action values.

$$\pi^{'}(s) = arg\ max_a\ Q_{\pi}(s,a) \tag{17}$$

$$Q_{\pi}(s,\pi^{'}(s)) = max_a\ Q_{\pi}(s,a) \geq Q_{\pi}(s,\pi^{optimal}(s)) = \upsilon_{\pi}(s) \tag{18}$$

$$Q_{\pi}(s,\pi^{'}(s)) = max_a\ Q_{\pi}(s,a) = Q_{\pi}(s,\pi^{optimal}(s)) = \upsilon_{\pi}(s) \tag{19}$$

4 EVALUATION

In this section, a brief discussion is stated for the recovery evaluation.

4.1 Environment Setup

To evaluate the effectiveness of the proposed controller, our setup consisted of four nodes. Three main nodes (i.e., VM instances). One node is for the heart rate monitor; one is for correctly configured containers-based clusters, and one is for the controller. The fourth node is for the misconfiguration scenarios. For each node, we deployed a set of containers and services. Each node is equipped with LinuxOS (Ubuntu 18.10 version), one VCPU, and 2GB VRAM. A K8s cluster running on VMs consisting of one master node and three worker nodes was deployed using Kubeadm, running K8s version 1.19.2. We created 30 namespaces, each with 4 microservices (pods) used for performance measurements, and assigned the same number of network policies. The number of created policies was 900, which were ordered, managed, and evaluated using Calico, Open Policy Agent, and Styra DAS, respectively. We verified the ability of the controller using a long time-span dataset (from 1 July 2021 to 1 November 2022). The model was trained for all-day and daytime-only on the collected data.

4.2 Data Collection and System Monitoring

Agents are installed to collect data about CPU, Memory, Network, filesystem changes, information flow (i.e., no. of flows issued to component), patient health information, device operation status, the device id, and service status from the system components. The agents exposed log files of system components to the storage to be used in the analysis. The agent adds a data interval function to determine the time interval at which the data collected belongs. The agent clears 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. Edge devices with similar functionality are grouped and allocated to a respective group (pool of heart monitor edge devices).

We used the Datadog tool to obtain a live data stream for the running components and to capture the request-response tuples and associated metadata. The captured data were streamed out of the cluster to be analyzed further. Prometheus is used to group the collected data (latency, throughput, bandwidth, throttling congestion, errors, number of requests, and resource utilization) and to store them in a time series database using Timescale-DB. 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 configuration change. We wrote our configuration files using YAML. We managed the configurations, deployments, and dependencies using kubectl and Skaffold.

The datasets were extracted from the monitoring tools and log files, and they were in a variety of sizes. The dataset used to train the model was divided into a 70% training set and 30% testing set.

4.3 Misconfiguration Scenarios

We trained our model on one of the identified misconfiguration types in (Samir and Dagenborg, 2023), which is the unauthenticated connection that happens because the pod was incorrectly configured with parameters made true for *privileged* and *hostPID*. Moreover, some misconfiguration types (e.g., CVE-2019-5736, CVE-2022-0811, CVE-2019-6538, CVE-2021-21284, CVE-2019-9946, and CVE-2020-10749) that allow privilege escalation was considered during the performance evaluation. We chose these types as they allow root access to the host, and they resulted in a sudden increase at the cluster level in request latency and the request rate falling, which caused excessive consumption of resource usage (CPU, memory, network). Furthermore, some of these types of misconfiguration might lead to improper access occurring at the edge level (e.g., CVE-2019-6538) due to no encryption to secure the communication protocol, and the protocol lacks authentication for legitimate devices.

4.4 The Recovery Assessment

This section focuses on evaluating the controller by measuring the reliability of recovery, deployment, and performance of the controller.

4.4.1 Reliability Evaluation

We used Mean Time to Recovery (MTTR) to evaluate the average time the recovery process takes to recover a component after observing a failure on the monitored metrics. The failure refers to a component that cannot meet its expected performance metrics. A higher MTTR indicates the existence of inefficiencies within the recovery process or the component itself. We conducted two scenarios. The first one corresponds to the selection of the optimal policy. The second relates to selecting a random policy, where the agent randomly selects one or more actions with uniform distribution. For each scenario, we aimed to assess the average time that the recovery process took to recover a container and an edge device. In the first scenario, the MTTR for the edge device was 20 s, and the MTTR to recover the container was approximately 43 s with a grace period of 80 s (default 30 s in Kubernetes) for service image size (110 MB) with service image number 30. For the second scenario, the MTTR for the edge device was 53 s, and the MTTR to recover the container was roughly 71 seconds under the same settings. We noticed that the container and the edge device function normally after that interval in both scenarios regarding the assigned rewards. The result of the first scenario led to a significantly short recovery time as the average achieved rewards through the optimal policy were remarkably higher than the random policy. Moreover, for some actions, such as reconfiguring and redeploying action, the more rewards are assigned during the recovery process, the average time decreases as the detection time was short, demonstrating a significant difference in the controller performance. However, to recover from failure efficiently, the average recovery time increased when the failure rate increased.

4.4.2 Deployment and Performance Evaluation

We verified if the captured variation in performance was due to a misconfiguration within the clusters and edge devices. Hence, spearman's rank correlation coefficient is used to estimate the correlation between valid configurations and normal system performance based on observing monitored metrics as the correct system behavior (more details, see (Samir and Pahl, 2019)). The decrease in the correlation degree tells that the observed degradation is not due to misconfiguration; otherwise, it refers to its existence. The general observation indicated a higher number of noticeable correlations between the misconfiguration and the performance indicators. The highest correlation was 0.82, and the lowest correlation was 0.34.

The configuration settings were checked against the benchmarks for misconfigurations during the deployment of a component (edge devices, containers, edge gateway, and clusters). The controller iterated all the security policies and guidelines (Azure, CIS Docker, and Kubernetes Benchmarks). In case of a mismatch between the settings and the requirements of secure deployment in one or more component(s), the controller reevaluates the deployment of the impacted component, applies the required reconfiguration, and redeploys the component. Otherwise, the component settings are secure as per security guidelines, and the controller proceeds with the deployment. The controller checked the misconfiguration, which needs to be addressed in components as a flag. Hence, we measured the average redeployment time for the component after observing anomalous behavior until the successful recovery of a component. The container redeployment average time was 210 seconds, with no observing overheads 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 185 seconds for the redeployment package with 110 MB. For the edge gateway, the average redeployment time was 95 seconds. Over multiple runs, the average redeployment time was reduced by 15∼30%, and the performance improved by 20% depending on the content and structure of the container's image and the available network bandwidth. In this sense, the platform had a significant impact on the redeployment time.

Moreover, the results show that the average amount of resource consumption (CPU, memory, network), with no misconfiguration, was approximately the same, with respective values varying around 30%∼60% (normal behavior). Resource consumption due to misconfiguration increased and was over 98% (overloaded resources), demonstrating the impact of improper configuration on the system resources. The recovered misconfiguration impacted the saturated resource as the values of the monitored resources varied around 38.4%∼64.6% (normal behavior). The controller performance was almost the same, with a minor recovery time deviation of around 100 seconds for some failure types, like container privileged access and wrong pod label. The deviation returned to the correlation with the failure in the system. Hence, we used the sequence of failures occurring during the recovery process to reflect the type of failure, which represents the failures that share the same observations corresponding to a unique fault. If the container privileged access and wrong pod label sequence of failures occurred, we focus on the container privileged access failure to represent its failure type and relate it to its fault, which is Privilege Access Escalation Management. We choose the initial failure that occurred as it is representative enough of the observations to which it belongs, which allows us to save the recovery time without trying many recovery actions.

In the end, we found that some anomalous behavior in the test set, such as CVE-2022-0811, is not covered by the training set, which might impact accuracy. The result stated that the controller performed better with the increase in the training dataset size. Moreover, we measured the average rate of successfully recovered components to the total number of misconfigurations in all anomalous components. After multiple runs, the average rate was around 97.66%, which means that the recovery could not handle a small number of misconfigurations, though the unhandled anomalous behavior decreased dramatically with more training data.

5 RELATED WORK

This section explores the recovery of misconfiguration in literature.

Various frameworks for managing workload and information flow in Edge/Fog environments have been developed; however, they provided limited scope for integrating different policies to manage the configurations of medical edge devices and clusters dynamically. In particular, existing frameworks have paid limited attention to the critical role of efficient recovery management (Mascellino, 2022), (Nie et al., 2021b), (Nie et al., 2021a), (Tang et al., 2018), (Taft, 2022), (Fairwinds, 2023), (Pelletier, 2020), (Alspach, 2022), (Dass and Namin, 2021).

In Pranata et al. (2020), the relationship between the service behavior and the value of some performance metrics is identified to discover the misconfiguration of cloud services using principal component analysis. In Durán and Salaün (2016), a protocol that reconfigures cloud applications is presented. It focuses on reconfiguring a set of interconnected component failures hosted on remote VMs. In Chiba et al. (2019), a performance-centric configuration framework for containers on 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. In Assuncao and Cunha (2013), a dynamic reconfigurable workflow framework is provided to manage failures of unavailable resources and variations in service quality. The framework recovers from failures in long-running workflow either by human intervention or a predefined task. In Vaquero et al. (2012) provides an architecture to control the behavior of the applications deployed in the cloud by using a set of defined rules. The rules are defined to create and configure the VMs. The architecture enables the re-definition of policies at the service and VM levels to describe their behavior.

Our work is similar to the ideas presented above. However, the presented controller extended the work in (Samir and Pahl, 2020), (Samir and Pahl, 2019) by mapping the observed performance degradation (failure) on performance metrics to its hidden abnormal flow of information (fault) and misconfiguration type (error) to analyze the misconfiguration and its consequence of threat within IoT edge devices and a cluster of containers running on cluster nodes; more details see (Samir and Dagenborg, 2023). Based on the mapping, the controller that is presented in this paper recovers the misconfiguration by selecting the optimal recovery policy with optimum actions to optimize the performance and the reliability of the system under observation.

6 CONCLUSIONS AND FUTURE WORK

Securing workloads and information flow in containers-based clusters Kubernetes and edge medical devices is an important part of overall system security. This paper presented a self-healing controller that recovers the misconfiguration of edge medical devices and container-based cluster systems using Markov Decision Processes (MDPs). The proposed controller optimizes the recovery process by selecting the optimal recovery policy with optimum actions to maximize the performance of observed metrics. The results show that the proposed controller is able to recover the misconfiguration with more than 97%, which demonstrates the suitability of the solution.

The aim of this paper was to introduce the recovery part of the controller architecture with its key processing steps. In the future, we will expand the recovery mechanism and carry out further experiments to fully confirm these given conclusions.

ACKNOWLEDGMENT

This research was funded in part by The Research Council of Norway under grant number 263248.

REFERENCES

Alspach, K. (2022). Major vulnerability found in open source dev tool for kubernetes.

Assuncao, L. and Cunha, J. C. (2013). Dynamic workflow reconfigurations for recovering from faulty cloud services. volume 1, pages 88–95. IEEE Computer Society.

Chiba, T., Nakazawa, R., Horii, H., Suneja, S., and Seelam, S. (2019). Confadvisor: A performance-centric configuration tuning framework for containers on kubernetes. pages 168–178.

Dass, S. and Namin, A. S. (2021). Reinforcement learning for generating secure configurations. *Electronics 2021, Vol. 10, Page 2392*, 10:2392.

DCMS (2018). Mapping of iot security recommendations, guidance and standards to the uk's code of practice for consumer iot security.

Durán, F. and Salaün, G. (2016). Robust and reliable reconfiguration of cloud applications. *Journal of Systems and Software*, 122:1339–1351.

Fairwinds (2023). Kubernetes benchmark report security, cost, and reliability workload results.

George, G. and Thampi, S. M. (2019). Vulnerability-based risk assessment and mitigation strategies for edge devices in the internet of things. *Pervasive and Mobile Computing*, 59.

Jin, X., Katsis, C., Sang, F., Sun, J., Kundu, A., and Kompella, R. (2022). Edge security: Challenges and issues. pages 1–21.

Kraus, S., Schiavone, F., Pluzhnikova, A., and Invernizzi, A. C. (2021). Digital transformation in healthcare: Analyzing the current state-of-research. *Journal of Business Research*, 123:557–567.

Mascellino, A. (2022). Nearly one million exposed misconfigured kubernetes instances could cause breaches.

Moothedath, S., Sahabandu, D., Allen, J., Clark, A., Bushnell, L., Lee, W., and Poovendran, R. (2020). Dynamic information flow tracking for detection of advanced persistent threats: A stochastic game approach. *arXiv:2006.12327*.

Nie, L., Ning, Z., Obaidat, M. S., Sadoun, B., Wang, H., Li, S., Guo, L., and Wang, G. (2021a). A reinforcement learning-based network traffic prediction mechanism in intelligent internet of things. *IEEE Transactions on Industrial Informatics*, 17:2169–2180.

Nie, L., Sun, W., Wang, S., Ning, Z., Rodrigues, J. J., Wu, Y., and Li, S. (2021b). Intrusion detection in green internet of things: A deep deterministic policy gradient-based algorithm. *IEEE Transactions on Green Communications and Networking*, 5:778–788.

Pelletier, J. (2020). Common kubernetes misconfiguration vulnerabilities.

Pranata, A. A., Barais, O., Bourcier, J., and Noirie, L. (2020). Misconfiguration discovery with principal component analysis for cloud-native services. pages 269–278. Institute of Electrical and Electronics Engineers Inc.

Rahman, A., Shamim, S. I., Bose, D. B., and Pandita, R. (2023). Security misconfigurations in open source kubernetes manifests: An empirical study. *ACM Transactions on Software Engineering and Methodology*.

Samir, A. and Dagenborg, H. (2023). A self-configuration controller to detect, identify, and recover misconfiguration at iot edge devices and containerized cluster system. pages 765–773.

Samir, A. and Pahl, C. (2019). A controller architecture for anomaly detection, root cause analysis and self-adaptation for cluster architectures. pages 75–83.

Samir, A. and Pahl, C. (2020). Autoscaling recovery actions for container-based clusters. *Concurrency and Computation: Practice and Experience*, 33:1–13.

Taft, D. K. (2022). Armo: Misconfiguration is number 1 kubernetes security risk.

Tang, F., Fadlullah, Z. M., Mao, B., and Kato, N. (2018). An intelligent traffic load prediction-based adaptive channel assignment algorithm in sdn-iot: A deep learning approach. *IEEE Internet of Things Journal*, 5:5141–5154.

Vaquero, L. M., Morán, D., Galán, F., and Alcaraz-Calero, J. M. (2012). Towards runtime reconfiguration of application control policies in the cloud. *Journal of Network and Systems Management*, 20:489–512.

Wang, S., Li, C., Hoffmann, H., Lu, S., Sentosa, W., and Kistijantoro, A. I. (2018). Understanding and auto-adjusting performance-sensitive configurations. volume 53, pages 154–168. Association for Computing Machinery.