MicroRAS: Automatic Recovery in the Absence of Historical Failure Data for Microservice Systems

Li Wu, Johan Tordsson, Alexander Acker, Odej Kao

MicroRAS: Automatic Recovery in the Absence of Historical Failure Data for Microservice Systems

Li Wu*
Elasisys AB, Sweden
TU Berlin, Germany
li.wu@elastisys.com

Johan Tordsson
Elasisys AB, Sweden
Umeå University, Sweden,
johan.tordsson@elastisys.com

Alexander Acker
TU Berlin, Germany
alexander.acker@tu-berlin.de

Odej Kao
TU Berlin, Germany
odej.kao@tu-berlin.de

Abstract

Microservices represent a popular paradigm to construct large-scale applications in many domains thanks to benefits such as scalability, flexibility, and agility. However, it is difficult to manage and operate a microservice system due to its high dynamics and complexity. In particular, the frequent updates of microservices lead to the absence of historical failure data, where the current automatic recovery methods fail short. In this paper, we propose an automatic recovery method named MicroRAS, which requires no historical failure data, to mitigate performance issues in microservice systems. MicroRAS is a model-driven method that selects the appropriate recovery action with a trade-off between the effectiveness and recovery time of actions. It estimates the effectiveness of an action in terms of its effects of recovering the pinpointed faulty service and its effects of interfering with other services. The estimation of action effects is based on a system-state model represented by an attributed graph that tracks the propagation of effects. For the experimental evaluation, several types of anomalies are injected into a microservice system based on Kubernetes, which also serves a real-world workload. The corresponding benchmarks show that the actions selected by MicroRAS can recover the faulty services by 94.7%, and reduce the interference to other services by at least 44.3% compared to baseline methods.

Keywords

automatic recovery, microservices, performance issues, cloud computing, Kubernetes

1 Introduction

Microservices architecture design is increasingly deployed in large scale software systems, particularly in cloud-based systems [1], [2]. The state of the art literature shows that microservices-based architectures can enhance the adaptability to technological changes, improve the scalability, and more importantly, reduce the time-to-market [3]. However, microservice systems tend to be fragile due to the highly-distributed nature and the large number of messages passed between services [4].

To achieve resilient microservices, proposed solutions ranging from fault-tolerant service design and development, service resilience testing, and self-healing exist or have to be developed yet. Several investigations have focused on resiliency patterns in microservices design [5]–[8] and testing [9]–[11]. By contrast, less attention has been placed on automatic recovery techniques.

One of the key problems arising in the automatic recovery is to determine: **given a detected service performance anomaly, which action(s) should be taken to mitigate the issue?** In a microservice system, the selection of recovery actions is difficult to achieve because of the following challenges: (1) *delusive corrective actions*: Due to the dynamics and complexity of microservice systems, the analysis of performance anomaly detection and root cause localization include frequently false positives. Such an incorrect analysis results in delusive corrective actions, thus reducing the probability to select the best possible recovery actions and increasing the risk of executing incorrect actions; (2) *frequent updates*: Microservices are updated frequently to meet customers' needs, (e.g., Netflix updates thousands of times per day [12]). These dynamic microservices make the historical data of recovery unavailable, decreasing the precision of the existing data-driven methods, thus aggravating the difficulty of action selection; (3) *a large number of metrics*: Due to the large-scale of microservices, the number of monitoring metrics is very high (e.g., Netflix exposes 2 million metrics [13]). It would cause significant overhead and delay if all these metrics were to be used for action selection; (4) *uncertainty*: the dynamics of the infrastructures and microservices introduce a great uncertainty to the system, it is hard to foresee the impact of the applied recovery actions. Therefore, to ensure the selected action is effective to a detected performance issue, it is crucial to develop a method to predict its effects in the absence of historical failure data.

In the literature, different approaches have been proposed to recover issues in cloud, networks, and distributed systems. For example, rule-based approaches select recovery actions by matching the user-defined rules [14]. However, the rules require frequent updates following the corresponding changes in the microservices, which conflicts the goal of automatic recovery. Case-based approaches identify recovery actions by matching previous failure cases [15], [16]. However, their overhead and delay are high due to the numerous metrics in microservices. This also holds true for the learning-based approaches [17]. Further suggested methods select recovery actions by analyzing the action properties [18]–[20]. However, they are highly dependent on the probabilistic parameters learned from recovery history (e.g., the success rate of an action to a given failure), and can be misled by delusive corrective actions. Notably, all

above approaches assume that historical failure data is available to learn, which is not always true in microservice systems.

To overcome the shortcoming of requiring historical failure data in the existing work, we propose an automatic recovery selection method, **Micro**services **R**ecovery **A**ction **S**election – **MicroRAS**, to mitigate the performance issues. MicroRAS is a model-based method that can adapt to the frequent changes of microservices without requiring historical data of previous failures and can reduce the potentially destructive consequences of recovery actions by assessing their side effects. MicroRAS firstly models the system state with an attributed graph used to track the propagation of positive and negative effects of recovery actions. Next, it estimates the benefit (positive effects) and the risk (negative effects) associated with each action by predicting the future state, where the system would transit with the selected action. Lastly, it aggregates all these effects into an effectiveness value with a fuzzy logic and selects the best possible action with a trade-off between action effectiveness and the time for the action to mitigate the issue. We evaluate our MicroRAS method by applying the selected recovery actions to mitigate different types of performance anomalies injected into a microservice system where the Sock-shop[1] microservices benchmark is deployed on Kubernetes running in Google Cloud Engine (GCE)[2]. The results show that the actions selected by MicroRAS can mitigate the performance issues well, with recovering the performance of faulty services by 94.7% and minimizing the affect on other service within 15.2%. In conclusion, our main contributions are the following:

- We propose a recovery action selection method based on real-time data collection and action properties observed during non-anomalous operation instead of historical failure data.
- We propose an action effects estimation model to capture the positive and negative effects associated with a recovery action, which is adaptive to the anomalous context of the system.
- We evaluate MicroRAS by mitigating different types, levels, and contexts of anomalies. The experimental results show that the actions selected by MicroRAS can mitigate the faulty service well, with affecting other services 44.3% less, and are completed at least 4 times faster than baseline recovery strategies.

The remainder of this paper is organized as follows. Section 2 illustrates the motivation of our method with a concrete example. Section 3 shows the overview of our method and Section 4 explains the details. Evaluations are described in Section 5. Section 6 discusses the related work and Section 7 concludes this paper.

2 Motivating Example

In this section, we use a concrete example to illustrate the motivation of our proposed method. For the sake of simplicity, we focus on a small part of a large-scale microservice-based application shown in Figure 1. This application consists of five microservices (MS) deployed on two hosts, where *MS 1*, *3* and *5* are co-located

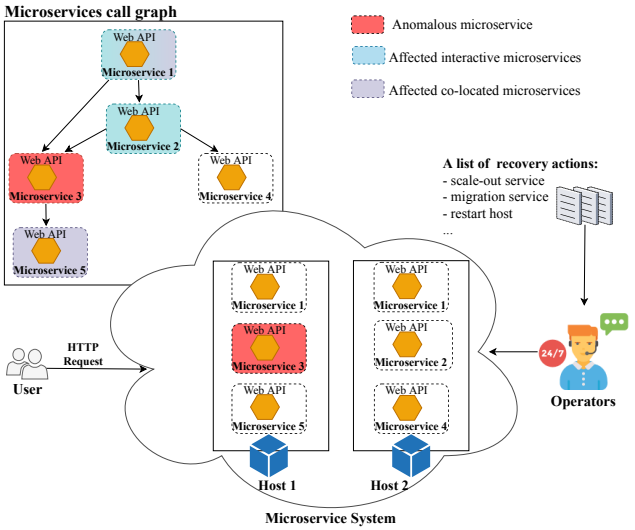


Figure 1: Motivating example: when scaling out Microservice 3 (*MS 3*) to recover a performance degradation, the consequences can be: 1) *MS 3* is recovered - when resources in the cluster are sufficient and recovery time is short; 2) Performance of *MS 1* and *2* also degrades - when the recovery time is long and anomaly propagates from *MS 3* to *MS 1* and *2*; and 3) Performance of *MS 1* and *5* (or *MS 1, 2,* and*5*) also degrades - when the resources of *Host 1* (or *Host 2*) where the new service instance runs is insufficient.

on *Host 1*, *MS 1*, *2* and *4* are on *Host 2*. Requests to *MS 1* are load-balanced across two replicas. The interactions among microservices are henceforth referred to as the *microservices call graph*.

Subsequently, slower response times of *MS 3* are observed and classified as a performance anomaly. The operators identify the root cause as *MS 3*, by manually debugging or root cause analysis tools. Meanwhile, they obtain a list of feasible recovery actions based on their expert knowledge and previous experience, the latter commonly maintained as scripts or playbooks [21]. The recovery actions can be restart service, scale-out service, restart host, etc.

Let us take *scale-out service* as an example of recovery action. When *MS 3* scales out, the consequences of this action can be diverse. If the recovery time (the time it takes for the action to have an effect and the microservice to recover from the performance issue) of scale-out is very short and the available resources are sufficient, the performance issue would be mitigated. However, if the recovery time is too long, the performance anomaly could propagate to upstream microservices, i.e., *MS 1* and *2* in the blue box in the call graph, increasing the response times of *MS 1* and *2*. Even worse, as *Microservice 1* is a user-facing microservice, it could cause service disruption for end-users directly. Besides, if the available resources on the host are insufficient, the scale-out action might affect the co-located microservices i.e., *MS 1* and *5* (purple in Figure 1), or even the entire application.

To mitigate performance issues in microservice systems without causing significant downtime, it is crucial to identify the appropriate action that can recover the anomalous services but also minimize

[1]Sock-shop - https://microservices-demo.github.io/
[2]Google Cloud Engine - https://cloud.google.com/compute/

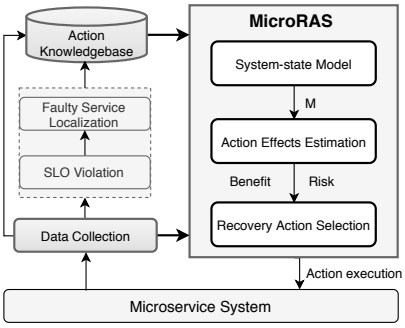


Figure 2: The workflow of recovery action selection.

the side effects on other services and the recovery time. In this paper, we propose a model to assess the positive and negative effects of potential recovery actions, and select the best possible action with a trade-off between the effects and the recovery time.

3 Overview of MicroRAS

To adapt to the dynamics of the microservice systems, where the recovery history is not always available, we propose MicroRAS, a recovery action selection method to automatically mitigate the performance issues based on real-time contextual information of the system. The main idea of our method is to understand the anomalous context of the system with a system-state model, using the data collected in real-time, then selecting recovery action by predicting the effects on the recovered system state if an action were to be applied. Recovery time is an important factor in this decision, as unattended anomalies can propagate quickly, and it is a common objective in the literature [17], [22], [23], MicroRAS takes both the action effects and recovery time as objectives and models the selection as an optimization problem.

Before the recovery process is initiated, detection of slower response times of microservices, location of the faulty service, and a set of feasible recovery actions (a knowledge base) are required. The methods for anomaly detection and faulty service localization have been well addressed in the literature [24]–[28], and the actions a knowledge base can be configured based on properties of the available recovery actions as observed in non-anomalous operations.

Figure 2 shows the workflow of recovery action selection. Once the selection process is triggered, MicroRAS selects the recovery action with the following steps: (1) it gathers the run-time contextual information of the system and models the system-state with an attributed graph that can also track the propagation of action effects across services and hosts. (2) it predicts the benefit and risk of each potential recovery action, by estimating the future state the system would transit into by applying that action. (3) it aggregates the action benefit and risk into an effectiveness value and formulates the action selection as an optimization problem with effectiveness and recovery time as the objectives. After the action execution, the observed state change caused by the action and the recovery time are used to update the action knowledge base. We remark that the runtime complexity of MicroRAS is low, as the time complexity of the operations in our method is linear to the number of the services, nodes and potential actions. Thus it scales well with the size of the microservice system.

4 The MicroRAS Method

In this section, we describe the three key modules in MicroRAS to select the best appropriate recovery action without historical failure data, namely system-state model (Section 4.1), action effects estimation (Section 4.2), and recovery actions selection (Section 4.3).

4.1 System-state model

To estimate the potential effects of an action on the system, awareness of the system context in different states is necessary. We build a system-state model to capture the context, including the dependency among components in the system and their states of resources.

In a microservice system, services inter-communicate through lightweight protocols and are deployed across multiple hosts. An action applied to one service does thus not only influence the service itself but also other services, either through invocation paths or their co-located hosts. Understanding service influence is similar to the anomaly propagation problem [29]. Therefore, we model the system-state with an attributed graph which can not only show the dependencies among services and hosts but also track the propagation of action effects. In addition, we define the state of services and hosts in the system as a set of variables SV. As MicroRAS aims at performance issues caused by resource bottlenecks, we store in SV the resource usage sv^{RU} and resource allocation sv^{RA}, in terms of CPU, memory, etc., The major notations used in the paper are summarized in Table 1. We define the system state model as follows:

System-state model: A set of system states M, including normal m^N, abnormal m^A, and recovered m^R state, is defined using an attributed graph G together with a set of system state-variables SV, including resource usage sv^{RU} and resource allocation sv^{RA}. Notably, the normal and abnormal states are fully observable, whereas accurate prediction of m^R is key to select the appropriate recovery action.

Once the response time between two services is slow and classified as a performance anomaly, MicroRAS constructs an attributed graph that holds the normal m^N and abnormal m^A system states, using the method proposed in our previous work [28]. In addition, the state variables SV are stored in the node attributes. The data for graph construction and state variables are gathered from the runtime monitoring of hosts and services, including use of a service mesh.

The attributed graph in Figure 3(a) corresponds to our motivating example in Figure 1. In Figure 3(a), the solid lines indicate service invocations and the dashes lines show which host the service runs on. For each service and host, we collect resource usage and allocation in normal and abnormal states. In particular, for service s_i which runs with multiple replicas (pods), we collect the resource data for each of the c pods s_{ij}. In Figure 3(a), $c = 2$ for service s_1, and 1 for the other services.

Table 1: Notations used in MicroRAS.

Notation	Description
A	a list of n_a feasible actions, $A = \{a_i\}_{i=1}^{n_a}$
G	an attributed graph
S	a set of n_s services $S = \{s_i\}_{i=1}^{n_s}$
H	a set of n_h hosts $H = \{h_j\}_{j=1}^{n_h}$
s_i	a service with c pods, $s_i = \{s_{ij}\}_{j=1}^{c}$
s_{ij}	a pod of service s_i, the pod runs on host h_j
M	a set of system state models, $M = \{m^i\}_{i \in \{N,A,R\}}$ each m^i is represented by a unique $\{G, SV\}$
m^N,m^A, m^R	normal, abnormal, and recovered system state
SV	a set of state variables of n_s services/pods and n_h hosts, $SV = \{sv_k\}_{k=1}^{n_s+n_h}$
sv_k	state variables of a pod/host, $sv_k = \{sv^{RU}, sv^{RA}\}$
sv^{RU}	resource usage (1 vCPU, 1GB memory, etc)
sv^{RA}	resource allocation such as host capacity, pod limits (2 vCPU, 2GB memory)
E, E_b, E_r	action effectiveness and its compositions: benefit, risk
$UT(s_{ij}), UT(h_j)$	resource utilization of pod s_{ij}, host h_j (%)
T	recovery time of an action

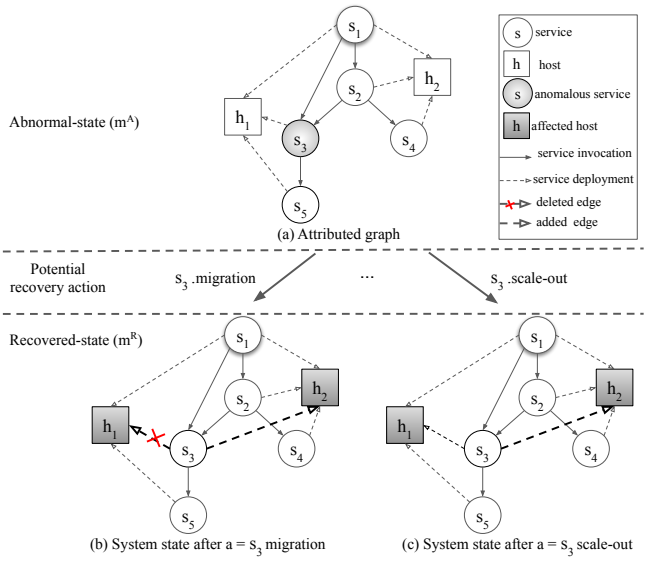


Figure 3: System states prediction.

4.2 Action effects estimation

Based on the observed normal and abnormal system states, we estimate effects associated with each potential recovery action by predicting the future state that the system would transit into if applying the action.

Action effects in MicroRAS are composed of positive and negative effects. We define the positive effect as the benefit E_b that the identified anomalous service would achieve in terms of service performance and the negative effect as the risk E_r that the affected hosts would have in terms of resource contention, thus affecting the services that run on the hosts. Due to the uncertainty and complexity of microservice systems, it is difficult to accurately estimate the performance of a service after recovery action execution, we first estimate the resource utilization UT of a service and next map the estimated UT into fuzzy sets of service performance using a fuzzy inference system described in Section 4.3. Hence, in order to estimate the action effects, we need to predict the recovered state, including the attributed graph and system state-variables after an action execution, in order to identify the potential affected hosts and compute the resource utilization of the faulty service (action benefit E_b) and affected hosts (action risk E_r).

To predict the recovered state of an action, we need to know not only the current system state but also the properties of the action, as the action affects the system state in different ways. Although there is a wide range of recovery actions, we only consider how the action modifies the system-state model. For a recovery action a_i in the action set A, we include the following properties:

- Topology: Some actions change the service location, thus changing the topology of the attributed graph.
- Resource usage: Some actions change the resource consumption of the service or host. For example, *restart* can recover anomalies caused by memory leaks, thus reducing resource usage.
- Resource allocation: Some actions change the capacity of hosts or the resource limits of pods. Example include *scale-up* and *scale-out* actions that increase the allocated resources of a host or a service.

Note that these properties, including the recovery time used in Section 4.3, are stored in the *action knowledge base* in Figure 2. All properties are obtained in non-anomalous operations and can be updated after the action is executed.

Based the abnormal state m^A and the *topology* property of a recovery action, MicroRAS predicts the graph changes in recovered state m^R. Figures 3(b) and (c) show the recovered states after migrating and scaling out anomalous service s_3, where service migration removed the link between s_3 and h_1 and adds a new link between s_3 and h_2, whereas service scale-out adds a new link between s_3 and h_2. In these two recovery actions, the affects hosts are h_1 and h_2.

After the attributed graph is predicted, MicroRAS estimates the resource utilization of the faulty service and affected hosts. When an action applies to pod s_{ij} of faulty service s_i or affects host h_j, the resource utilization in recovered state m^R is defined as the ratio between resource usage and allocation:

$$UT(h_j, m^R) = \frac{sv^{RU}(h_j, m^R)}{sv^{RA}(h_j, m^R)}, \quad UT(s_{ij}, m^R) = \frac{sv^{RU}(s_{ij}, m^R)}{sv^{RA}(s_{ij}, m^R)}. \tag{1}$$

Assuming an ideally equal load balancing between pods of the same service, the utilization of service s_i is defined as:

$$UT(s_i, m^R) = \frac{1}{c} \sum_{j=1}^{c} UT(s_{ij}, m^R). \tag{2}$$

where c is the total number of pods. As some actions may under-provision the resource, the estimated UT can be over 1, thus its range is defined as $UT > 0$.

Based on the action properties and system state-variables in normal and abnormal states, MicroRAS estimates the resource usage and resource allocation of pod s_{ij} and host h_j in the recovered state as follows.

The future resource usage of a pod in recovered state varies with the recovery action. If the action modifies the pod, it is the configured resource usage Δsv^{RU}; If the pod is newly created by the action, it is assigned with the service normal resource usage after load-balancing. Otherwise, the pod keeps the abnormal resource usage. Taking Figure 3 as an example, the resource usage of pod s_{32} in action *migration* in Figure 3(b) is Δsv^{RU}; The resource usage of pod s_{31} in action *scale-out* in Figure 3(c) keeps the abnormal resource usage $sv^{RU}(s_{31}, m^A)$, and pod s_{32} is assigned with the load-balanced normal resource usage of service s_3, which is $sv^{RU}(s_3, m^N)/2$. We summarize the pod resource usage in Equation 3.

$$sv^{RU}(s_{ij}, m^R) = \begin{cases} \Delta sv^{RU}, & \text{if } s_{ij} \text{ is modified,} \\ sv^{RU}(s_{ij}, m^A), & \text{if } s_{ij} \text{ is not modified,} \\ \frac{sv^{RU}(s_i, m^N)}{c}, & \text{if } s_{ij} \text{ is a new pod.} \end{cases} \tag{3}$$

Pod future resource allocation $sv^{RA}(s_{ij}, m^R)$ depends on the pod limits and the available resources of host h_j it runs on. If the available resources in h_j is sufficient (exceeds the pod limits), $sv^{RA}(s_{ij}, m^R)$ is equal to the pod limits and otherwise to the available resources in h_j. The pod limits can be modified by the action with Δsv^{RA} or kept in abnormal state. The available resource of h_j is the host resource allocation $sv^{RA}(h_j, m^R)$ with a consumption of $sv^{RU}(h_j, m^A)$, where $sv^{RA}(h_j, m^R)$ can be modified by the action or remain the same as $sv^{RA}(h_j, m^A)$:

$$sv^{RA}(h_j, m^R) = \begin{cases} \Delta sv^{RA}, & \text{if } h_j \text{ is modified,} \\ sv^{RA}(h_j, m^A) & \text{otherwise.} \end{cases} \tag{4}$$

Once the future pod resource usage (Equation 3) and host resource allocation (Equation 4) are determined, we can estimate the future resource utilization of the affected host h_j where the pod s_{ij} runs on, as shown in Equation 5. Host resource usage $sv^{RU}(h_j, m^A)$ increases by pod resource usage $sv^{RU}(s_{ij}, m^A)$ if pod s_{ij} is migrated to h_j, or decreases by $sv^{RU}(s_{ij}, m^A)$ if s_{ij} is migrated from h_j to another host.

$$UT(h_j, m^R) = \frac{sv^{RU}(h_j, m^A) \pm sv^{RU}(s_{ij}, m^A)}{sv^{RA}(h_j, m^R)} \tag{5}$$

Future resource utilization of h_j is summarized in Equation 6. For each host that service s_i runs on, we calculate the resource utilization and use the maximum utilization as the risk of the action.

$$UT(h_j, m^R) = \begin{cases} \frac{\Delta sv^{RU}}{sv^{RA}(h_j, m^R)}, & \text{if } h_j \text{ is modified,} \\ \text{as per Equation 5,} & \text{if } s_{ij} \text{ is migrated.} \end{cases} \tag{6}$$

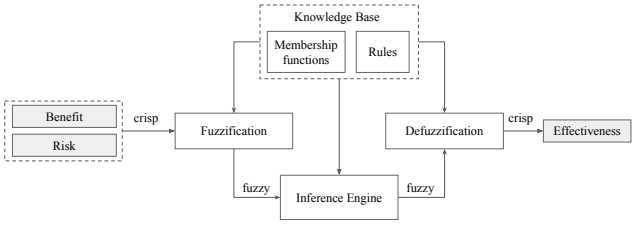


Figure 4: The structure of the fuzzy inference to combine action risk and benefit.

4.3 Recovery action selection

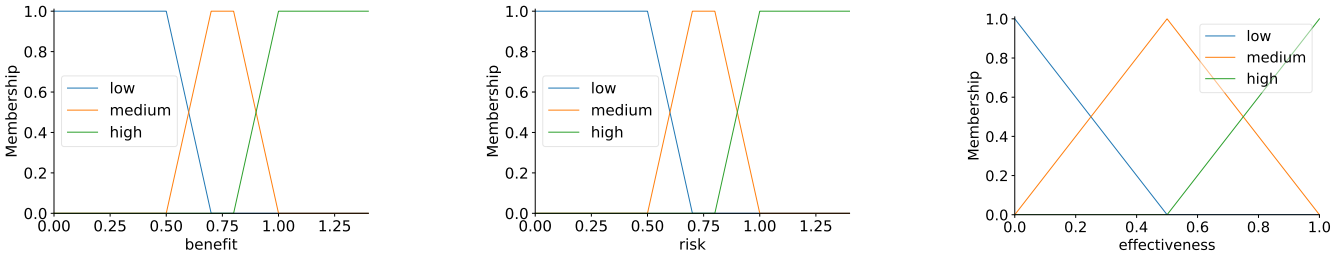
After we estimate the benefit and risk associated with each recovery action in terms of resource utilization, we map these into fuzzy sets of service performance and aggregate them into a single crisp effectiveness value through a fuzzy inference system [30], illustrated in Figure 4. Finally, we formulate the selection problem as an optimization problem and select an appropriate action with a trade-off between action effectiveness and recovery time.

To calculate the effectiveness value, the fuzzy inference system uses membership functions to determine the degree that its inputs belong to each of the relevant fuzzy sets. For this purpose, three overlapping fuzzy sets are created. For the action risk, host resource utilization values between 0 and 70% are in the Low range, values between 50% and 80% are in the Medium range, and values above 80% are in the High range.

A membership function defines how the input value is mapped to the membership degree between 0 and 1, where 0 means the input does not belong to the given fuzzy set, and 1 means the input completely belongs to it. Similar to [19], [31], the membership functions for the three fuzzy sets in inputs are respectively a R-function, a trapezoidal function and a L-Functions, as shown in Figure 5(a). The membership function used in the output is three triangular functions, as shown in Figure 5(b). Taking the action risk 0.7 as an example, according to its membership function in Figure 5(a), it has membership degree 0.2 in Low set, 0.7 in the Medium set, and 0 in the High set. These values are used for the fuzzy rules in the fuzzy reasoning. The fuzzy rules for the inference system are defined based on the microservice system and its administrative policy. MicroRAS uses the fuzzy rules shown in Table 2.

Based on the inputs, some fuzzy rules are fired and integrated. The decisions are made according to the aggregation of the fired fuzzy rules. The aggregated fired fuzzy rules output a single fuzzy set which is the input of the defuzzification procedure. We use the centroid method for defuzzification to convert the fuzzy set into crisp effectiveness value.

After obtaining the effectiveness values of the potential recovery actions, we formulate the action selection as an optimization problem, taking the estimated effectiveness and action recovery time as the objectives. Recovery time of an action is measured as the time between the action initiation and completion in normal status, which initially was obtained by executing the recovery action in non-production environments. Once the action is executed to actually recover an anomaly, the recovery time is updated with the time between action initiation and action taking effect.



(a) membership function of benefit **(b) membership function of risk** **(c) membership function of effectiveness**

Figure 5: Fuzzy membership functions.

Table 2: Fuzzy rules for action effectiveness.

Benefit	Risk	Effectiveness
Low	High	Low
Low	Medium	Low
Low	Low	Medium
Medium	High	Low
Medium	Medium	Medium
Medium	Low	Medium
High	High	Low
High	Medium	Medium
High	Low	High

Given a set of potential recovery actions A, for each recovery action $a_i \in A$, the estimated effectiveness is $E(a_i)$ and its recovery time is $T(a_i)$. For consistency purposes, we normalize the values of effectiveness and recovery time into the range $(0, 1)$ through Min-Max normalization. The performance of action a_i is quantified with a utility function $u(a_i)$:

$$u(a_i) = w_e E(a_i) - w_t T(a_i) \tag{7}$$

where w_e and w_t are user-defined weights for action effectiveness and recovery time ($w_e + w_t = 1, 0 < w_e, w_t < 1$). By setting the weights, users can prioritize the effectiveness and recovery time. We finally select the action that has the highest utility value among the recovery action in A according to Equation 7.

5 Experimental Evaluations

In this section, we evaluate the performance of MicroRAS through experiments on a cloud testbed. The experimental setup, evaluation results, and comparisons are presented.

5.1 Experimental Setup

Testbed: We evaluate MicroRAS in a testbed hosted in Google Cloud Engine (GCE)[2], where we create a Kubernetes cluster, run a microservices benchmark named Sock-shop[1], and deploy data collection tools. In the cluster, there is one master node and four worker nodes; three of them are dedicated for microservices and the last one for data collection. In addition, one VM outside the cluster is used for the workload generator. The detailed configurations of hardware and software are shown in Table 3.

Table 3: Hardware and software configuration used in testbed.

Hardware Configuration			
Component	**Master node**	**Worker node(x4)**	**Workload generator**
Operating System	Container-Optimized OS	Container-Optimized OS	18.04.2 LTS
vCPU(s)	1	4	6
Memory(GB)	3.75	15	12
Software Version			
Kubernetes	**Istio**	**Prometheus**	**Node-exporter**
1.14.1	1.1.5	2.3.1	v0.15.2

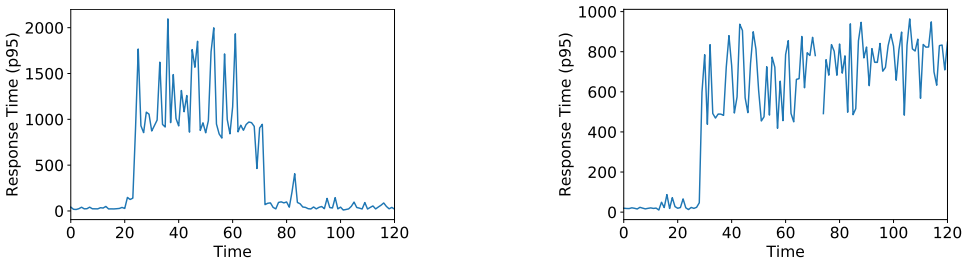
Benchmark: Sock-shop[1] is a widely used microservices benchmark that simulates an e-commerce website that sells socks. It consists of 13 microservices, which are independent and intercommunicate using REST APIs. Seven out of the 13 microservices are for the main business goals, such as frontend and backend services. In the deployment, we limit the CPU resource to 1 vCPU and memory to 1 GB for these seven key microservices. For simplicity, we set the replication factor to one for each microservice. To measure the consequences of different actions in the same anomaly scenario, we taint each microservice to a specific cluster node and reset the environment for each action.

Workload Generator: We use Locust[3] to simulate concurrent users in an application. In each case, 500 users are provisioned and in total about 600 queries are generated per second to Sock-shop in normal state. The queries to different services are selected to reflect real user behavior, e.g., more requests are sent to the entry points *front-end* and *catalogue*, and fewer to the other services.

Data Collection: We collect resources relevant metrics (e.g., CPU usage, memory usage) in container and node levels, using cAdvisor[4] and node-exporter[5], and collect response times for each microservice invocation with the Istio[6] service mesh. We use Prometheus[7] to scrape all the metrics every five seconds and store these in a time-series database.

Faults Injection: We evaluate our MicroRAS with two different types of anomalies (CPU hog and memory leak), different levels of anomalies (stressing services and hosts), and different contexts

[3]Locust - https://locust.io/
[4]cAdvisor - https://github.com/google/cadvisor
[5]Node-exporter - https://github.com/prometheus/node_exporter
[6]Istio - https://istio.io/
[7]Prometheus - https://prometheus.io/



(a) scale-out pod (b) restart pod

Figure 6: Two recovery actions for service performance issue caused by insufficient host resources: (a) scale-out pod recovers the issue, but (b) restart pod has no effect.

Table 4: Details of anomaly scenarios.

anomaly type		host-level	service-level
CPU Hog (vCPU * %)	cluster sufficient	4*95	3*95
	cluster insufficient (other hosts)	4*80	4*80
Memory Leak (vm * %)		1*73	2*50

(with total cluster resources either sufficient or insufficient to resolve the anomaly). To inject the CPU hog and memory leak, we use *stress-ng* [8], a tool to load and stress computer system to exhaust the CPU and memory resources continuously. To inject performance issues in microservices, we customize the existing Sock-shop docker images by installing the faults injection tool. The injected microservice is *catalogue* and the injected host is the host *catalogue* runs on. In the cluster resources sufficient scenario, we only inject anomalies to service or host. In the cluster resources insufficient scenario, we also stress the other hosts. The details of the anomaly scenarios are shown in Table 4.

In each case, we run the microservices in normal status for 2 minutes with the workload generator running. We next introduce the anomaly and let it run for 3 minutes before MicroRAS is used to select and execute a recovery action. After action executed, we collect another 5 minutes of data to measure the action consequences. To increase the generality, we repeat 3-5 times for each anomaly scenario. This produces a total of 23 experimental cases. In each anomaly scenario, we take 6 types of recovery actions, which are: *no action*, *restart pod* (in the same host), *migrate pod* (shutdown and start the pod again), *scale-out pod*, *scale-up pod* and *restart host*. Figure 6 gives two examples of data collected after applying scale-out pod and restart pod when the host CPU resource is insufficient, with pod scale-out (Figure 6(a)) having positive effect while pod restart (Figure 6(b)) did not recover the anomaly.

Evaluation Metrics: To quantify the performance of recovery action selection, we use following metrics:

- Recovered Percentage (RP) quantifies the positive effects of a recovery action a on the anomalous service s_a. It is defined as the percentage of service performance recovered from abnormal state $perf(s_a, m^A)$ to recovered state $perf(s_a, m^R)$ with action a, to the abnormal deviation from normal state $perf(s_a, m^N)$.

$$RP(a) = \frac{perf(s_a, m^A) - perf(s_a, m^R|a)}{perf(s_a, m^A) - perf(s_a, m^N)} \tag{8}$$

- Affected Percentage (AP) quantifies the negative effects of a recovery action on affected services $\{s_i|i = 1, 2, ...N\}$, where N is the number of affected services. AP is defined as the mean percentage of decreased performance for affected services from abnormal state $perf(s_i, m^A)$ to recovered state $perf(s_i, m^R|a)$ with action a, to the normal state $perf(s_i, m^N)$:

$$AP(a) = \frac{1}{N}\sum_{s_i} \frac{perf(s_i, m^R|a) - perf(s_i, m^A)}{perf(s_i, m^N)} \tag{9}$$

- Recovery Time (RT) quantifies time from initiating the mitigation action until the performance of the anomalous service and any affected services have stabilized.

5.2 Experimental Results

In our experiments, under normal workload, the 50th percentile (p50) of service response times is around 10 ms in normal status, and is in range (35 ms, 300 ms) in abnormal status, depending on the anomaly types; the 95th percentile (p95) of service response times is around 40 ms in normal status, and ranges from 160 ms to 2000 ms in abnormal status.

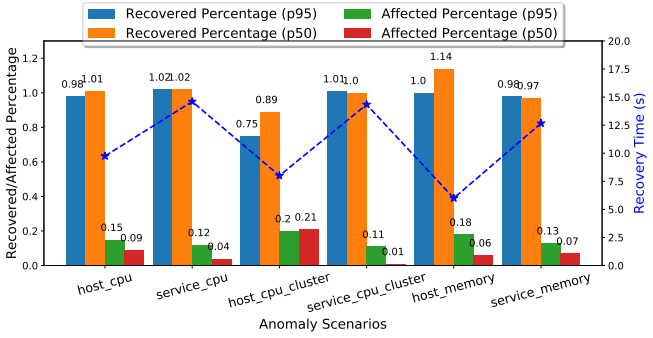


Figure 7: Performance of MicroRAS in terms of p95 and p50.

Figure 7 shows the results of our proposed recovery action selection method for mitigating different anomaly scenarios. For each

[8] stress-ng - https://kernel.ubuntu.com/ cking/stress-ng/

Table 5: Performance of MicroRAS in different types of anomaly scenarios.

Fault Scenario	CPU Hog	Memory Leak	Host-level	Service-level	Cluster Sufficient	Cluster Insufficient	Overall
Recovered Percentage	1.002	0.99	0.91	**1.007**	1.002	0.883	0.947
Affected Percentage	0.137	0.155	0.178	**0.12**	0.137	0.156	0.152
Recovery Time(s)	12.175	9.333	**7.917**	13.867	12.175	11.167	10.913

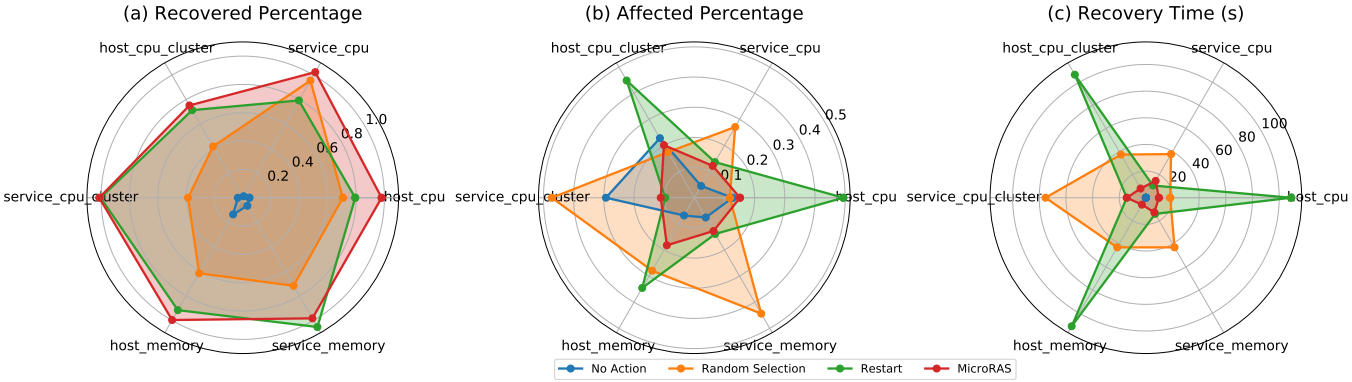


Figure 8: Performance summary for different strategies, performance metrics, and anomaly scenarios.

anomaly scenario, the bar charts show the mean recovered percentage (RP) and affected percentage (AP) in terms of p95 and p50 of response times and the dashed line shows the mean recovery time.

From the results, we can observe that MicroRAS can on average mitigate all the injected performance issues within 15 seconds. Furthermore, after applying the MicroRAS selected recovery actions, the anomalous service recovers at least 0.91 of its degraded performance across all anomaly scenarios, and the degradation in the recovered state was less than 18% from normal performance, except in the *host_cpu_cluster* anomaly scenario. The performance of this *host_cpu_cluster* anomaly scenario is lower than for the others as the resources of the cluster are insufficient, thus any recovery action is bound by overall resource shortage and thus negatively affects other services.

We aggregate RP p95 and AP p95 according to the type of anomaly scenarios in Table 5. We can see that MicroRAS overall can recover 0.947 of anomalous service degraded performance and mitigate the issues on average in 11 seconds. The performance of the service-level is better than host-level. This is because the service-level issues can be recovered entirely with the provided actions. However, the host-level issues can only be mitigated by most of the provided actions, further actions such as cluster scale-out are required to fix the issues entirely. For the same reasons, MicroRAS performs better in cluster sufficient than cluster insufficient cases.

5.3 Comparisons

To evaluate the performance of MicroRAS further, we compare it with three recovery strategies which require no historical data and are commonly used in the comparisons in the literature.

- *No Action*: Here, the operation team just passively observes the system without taking any actions. In our experiments, this strategy shows the potential damages of the injected performance issues, when left unattended.
- *Random Selection*: This strategy might be adopted when the operation team cannot determine the correct recovery action precisely, but urgently are trying to fix the problem. The operating team randomly selects one recovery action from the candidates and applies it to the system [23].
- *Restart*: This is a very popular recovery strategy, which can be applied at various levels. In a production environment, a significant fraction of failures can be cured by restarts [32]. We perform restarts at the host or pod level to resolve host and service level issues, respectively.

We compare the performance of each recovery strategy on different anomaly scenarios in terms of RP, AP, and recovery time in Figure 8. We observe that the performance issues cannot recover, or even deteriorates and affects other services if no action is taken. All the actions selected by the other three strategies can improve the performance of anomalous services. Notably, MicroRAS selects the best action in all scenarios but one, only slightly beaten by restart for memory leaks at the service level (Figure 8(a)) and has a shorter recovery time (Figure 8(c)) than Random Selection and Restart.

Both Restart and our MicroRAS have a good performance in terms of RP to all types of anomaly scenarios. However, Restart has a higher risk of affecting other services and longer recovery times when the anomaly exists at the host level. Figure 9 shows the AP and number of affected services in each anomaly case. The solid lines show the results of MicroRAS, and the dashed lines show the results of Restart. We observe that MicroRAS and Restart have similar AP and affected number for service-level anomalies. However, Restart has a higher AP and affected number for host-level anomalies. This is because compared to service-level operations, restarting a host commonly takes a longer time and all the services running on

the host would be restarted, which introduces fluctuations and uncertainty in the system.

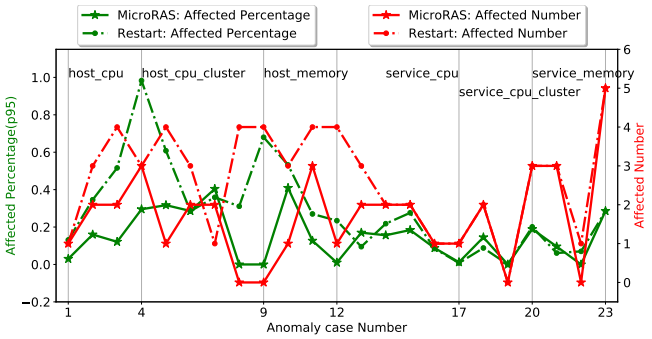


Figure 9: Affected percentage and number comparison.

Finally, we compare the overall performance of all strategies for all types of anomaly scenarios. Table 6 shows the performance, in terms of RP, AP, and recovery time (RT), for the four recovery strategies. We can observe that MicroRCA outperforms other strategies overall. In particular, MicroRAS achieves a recovered percentage of 94.7%, and affects other services at least 44.3% less and is completed at least 4 times faster than other strategies.

Table 6: Overall performance of different strategies.

Metrics	RP	AP	RT(s)
No Action	0.037	0.138	-
Random Selection (RS)	0.646	0.273	40.565
Restart	0.897	0.295	62.652
MicroRAS	0.947	0.152	10.913
Improvement to RS(%)	46.6	44.3	73.1
Improvement to Restart (%)	5.5	48.5	82.6

6 Related Work

A wide variety of techniques and approaches have been proposed to mitigate problems in cloud, networks, and distributed systems [33], [34]. Some of them work on a specific recovery approach, such as reboot [35], check-pointing [36], self-adaptation [37]–[39], placement [22], etc. Some of them focus on general recovery strategies. We herein review the related work in general automatic recovery from the following aspects.

Rule-based approaches transfer the expert knowledge into IF-THEN rules and use policies to match the rules for reacting to the faults [14]. This kind of approach is easy to develop. However, formalizing the rules requires a lot of expertise, and is a difficult and time-consuming task. In addition, frequent human intervention to revise the rules is required to keep them up-to-date in dynamic microservices systems, which is in conflict with the goal of automatic recovery.

Case-based approaches take previous failures as cases and match against the cases when a new fault occurs [15], [40]. This kind of approach can effectively avoid repeating past mistakes and can adapt to the changes in the system. However, similar problems in microservices commonly give rise to different symptoms due to technology heterogeneity and frequent updates. Thus, it is difficult and error-prone to apply the case matching. Furthermore, the number of exposed metrics in microservices is very high, computing the similarity between these metrics would cause significant overhead and delay.

Learning-based approaches use reinforcement learning or deep learning to generate recovery policies [17], [41] or commands [42] without human intervention. This kind of approach views the system as a black-box and can adapt to the changes in microservices. However, it requires a large set of historical failure data to train the model, which is difficult to obtain in microservice systems. Our method can complement this approach to help recover newly updated services. Once the failure data are available, this learning-based method can provide another recommendation for the action.

Model-based approaches model different aspects of a healing process, such as the properties of the fault [43], the properties of the actions [44], or use theoretical techniques, like Markov decision theory [44]–[46]. Similar to our method, consequences of recovery actions are also considered in [18], [19], [47]. M.Fu et al. [47] define the impact of an action on service response times which are caused by the increasing requests introduced by different recovery patterns. However, the impact of a performance issue in microservice systems, such as hardware failure, software bugs, resource contention, etc, cannot manifest in the number of requests, so their impact model is not suitable to our problem. Others [18], [19] define their models based on probabilistic parameters learned from recovery history (e.g., the prior probability of the system being in a stable state after executing an action). However, these probabilistic parameters are difficult to obtain in frequently updated microservice systems. Our MicroRAS system estimates the action consequences based on a system-state model which is built solely on data collected in real-time and action properties defined in normal status.

7 Conclusion and Future Work

In this paper, we propose a method named MicroRAS, to select the best possible recovery action based on an action effectiveness assessment model in order to mitigate the performance degradation in microservice systems. We estimate the positive and negative effects for each action and select the action with the best trade-off between action effects and recovery time based on data collected in real-time and knowledge obtained in normal status, without use of historical failure data. The estimation of the effects utilizes a system-state model, which is represented by an attributed graph used to track the propagation of action effects across services and hosts. The experimental results show that MicroRAS can effectively recover the anomalous services by 94.7% of their degraded performance while affecting the performance of other services at least 44.3% less. Finally, the mitigation is completed at least 4 times faster than baseline recovery strategies.

As our method considers a single-step ahead mitigation only, some performance issues cannot recover completely. In the future, we plan to investigate multi-action recovery strategies and how to schedule the recovery actions to minimize outage time. In addition, we consider to extend our method to a hybrid one with a learning process to be used when historical failure data is available.

8 Acknowledgments

This work is part of the FogGuru project which has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765452. The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

References

[1] D. Gannon *et al.*, "Cloud-native applications", *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21, 2017.

[2] L. Abdollahi Vayghan *et al.*, "Deploying microservice based applications with kubernetes: Experiments and lessons learned", in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 970–973.

[3] A. Balalaie *et al.*, "Microservices architecture enables devops: Migration to a cloud-native architecture", *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[4] P. Jamshidi *et al.*, "Microservices: The journey so far and challenges ahead", *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[5] S. Haselböck *et al.*, "Decision guidance models for microservices: Service discovery and fault tolerance", in *ECBS '17*, 2017.

[6] F. Montesi and J. Weber, *Circuit breakers, discovery, and api gateways in microservices*, 2016. arXiv: 1609.05830.

[7] G. Toffetti *et al.*, "Self-managing cloud-native applications: Design, implementation, and experience", *Future Generation Computer Systems*, vol. 72, pp. 165–179, 2017.

[8] A. Akbulut and H. G. Perros, "Performance analysis of microservice design patterns", *IEEE Internet Computing*, vol. 23, no. 6, pp. 19–27, 2019.

[9] V. Heorhiadi *et al.*, "Gremlin: Systematic resilience testing of microservices", in *ICDCS*, 2016, pp. 57–66.

[10] N. T. Blog. (2014). Fit: Failure injection testing, [Online]. Available: https://netflixtechblog.com/fit-failure-injection-testing-35d8e2a9bb2 (visited on 05/19/2020).

[11] H. S. Gunawi *et al.*, "Fate and destini: A framework for cloud recovery testing", in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11, Boston, MA: USENIX Association, 2011, pp. 238–252.

[12] *Why Netflix, Amazon, and Apple Care About Microservices*, (accessed: 30.05.2020).

[13] J. Thalheim *et al.*, "Sieve: Actionable insights from monitored metrics in distributed systems", in *Middleware '17*, 2017, pp. 14–27.

[14] H. Mfula *et al.*, "Self-healing cloud services in private multi-clouds", in *HPCS*, 2018, pp. 165–170.

[15] S. Montani *et al.*, "Case-based reasoning for autonomous service failure diagnosis and remediation in software systems", in *ECCBR*, 2006, pp. 489–503.

[16] S. Nasir *et al.*, "Optimization of decision making in cbr based self-healing systems", in *2012 10th International Conference on Frontiers of Information Technology*, 2012, pp. 68–72.

[17] Q. Zhu *et al.*, "A reinforcement learning approach to automatic error recovery", in *DSN*, 2007, pp. 729–738.

[18] S. Ossenbühl *et al.*, "Towards automated incident handling: How to select an appropriate response against a network-based attack?", in *IMF*, 2015, pp. 51–67.

[19] J. Shetty *et al.*, "Proactive cloud service assurance framework for fault remediation in cloud environment", *IJECE*, vol. 10, no. 1, p. 987, 2020.

[20] R. Alsoghayer and K. Djemame, "Resource failures risk assessment modelling in distributed environments", *Journal of Systems and Software*, vol. 88, pp. 42–53, 2014.

[21] B. Beyer, *Site reliability engineering : How Google runs production systems*. Sebastopol, CA, 2016.

[22] F. Díaz-Sánchez, S. Al Zahr, and M. Gagnaire, "An exact placement approach for optimizing cost and recovery time under faulty multi-cloud environments", in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 2, 2013, pp. 138–143.

[23] S. Huang *et al.*, "Differentiated failure remediation with action selection for resilient computing", in *PRDC*, 2015, pp. 199–208.

[24] M. Pahl and F. Aubet, "All eyes on you: Distributed multi-dimensional iot microservice anomaly detection", in *2018 14th International Conference on Network and Service Management (CNSM)*, 2018, pp. 72–80.

[25] A. Samir and C. Pahl, "Dla: Detecting and localizing anomalies in containerized microservice architectures using markov models", in *2019 7th International Conference on Future Internet of Things and Cloud (FiCloud)*, 2019, pp. 205–213.

[26] M. Ma *et al.*, "Automap: Diagnose your microservice-based web applications automatically", in *Proceedings of The Web Conference 2020*, ser. WWW '20, Taipei, Taiwan: Association for Computing Machinery, 2020, pp. 246–258.

[27] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification", *ACM Comput. Surv.*, vol. 48, no. 1, 2015.

[28] L. Wu *et al.*, "MicroRCA: Root Cause Localization of Performance Issues in Microservices", in *NOMS*, 2020.

[29] J. Weng *et al.*, "Root cause analysis of anomalies of multitier services in public clouds", *IEEE/ACM Transactions on Networking*, vol. 26, no. 4, pp. 1646–1659, 2018.

[30] S. Frey *et al.*, "Cloud qos scaling by fuzzy logic", in *2014 IEEE International Conference on Cloud Engineering*, 2014, pp. 343–348.

[31] H. Arabnejad *et al.*, "A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling", in *CCGRID*, 2017, pp. 64–73.

[32] C. Wang *et al.*, "Performance troubleshooting in data centers: An annotated bibliography?", *SIGOPS Oper. Syst. Rev.*, vol. 47, no. 3, pp. 50–62, 2013.

[33] P. Garraghan, R. Yang, Z. Wen, A. Romanovsky, J. Xu, R. Buyya, and R. Ranjan, "Emergent failures: Rethinking cloud reliability at scale", *IEEE Cloud Computing*, vol. 5, no. 5, pp. 12–21, 2018.

[34] I. Brandic, "Towards self-manageable cloud services", in *2009 33rd Annual IEEE International Computer Software and Applications Conference*, vol. 2, 2009, pp. 128–133.

[35] G. Candea *et al.*, "Microreboot — a technique for cheap recovery", in *Proceedings of the 6th Conference on Symposium on Operating Systems Design Implementation - Volume 6*, ser. OSDI'04, San Francisco, CA: USENIX Association, 2004, p. 3.

[36] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems", *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 23–31, 1987.

[37] V. Nallur and R. Bahsoon, "A decentralized self-adaptation mechanism for service-based applications in the cloud", *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 591–612, 2013.

[38] J. P. Magalhães and L. M. Silva, "A framework for self-healing and self-adaptation of cloud-hosted web-based applications", in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 1, 2013, pp. 555–564.

[39] M. Ben-Yehuda, D. Breitgand, M. Factor, H. Kolodner, V. Kravtsov, and D. Pelleg, "NAP", in *Proceedings of the 6th international conference on Autonomic computing - ICAC '09*, ACM Press, 2009.

[40] G. Li *et al.*, "A self-healing framework for qos-aware web service composition via case-based reasoning", in *Web Technologies and Applications*, 2013, pp. 654–661.

[41] M. L. Littman, N. Ravi, E. Fenson, and R. Howard, "Reinforcement learning for autonomic network repair", in *International Conference on Autonomic Computing, 2004. Proceedings.*, 2004, pp. 284–285.

[42] H. Ikeuchi, A. Watanabe, T. Hirao, M. Morishita, M. Nishino, Y. Matsuo, and K. Watanabe, "Recovery command generation towards automatic recovery in ict systems by seq2seq learning", in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, Budapest, Hungary: IEEE Press, 2020, pp. 1–6.

[43] Y. Dai *et al.*, "Self-healing and hybrid diagnosis in cloud computing", in *Cloud Computing*, 2009, pp. 45–56.

[44] A. Samir and C. Pahl, "Self-adaptive healing for containerized cluster architectures with hidden markov models", in *FMEC*, 2019, pp. 68–73.

[45] K. R. Joshi, M. A. Hiltunen, W. H. Sanders, and R. D. Schlichting, "Automatic model-driven recovery in distributed systems", in *SRDS'05*, 2005, pp. 25–36.

[46] K. R. Joshi *et al.*, "Automatic recovery using bounded partially observable markov decision processes", in *DSN'06*, 2006, pp. 445–456.

[47] M. Fu *et al.*, "Runtime recovery actions selection for sporadic operations on cloud", in *ASWEC*, 2015, pp. 185–194.