MicroRCA: Root Cause Localization of Performance Issues in Microservices

Li Wu*†, Johan Tordsson*‡, Erik Elmroth*‡, Odej Kao†
*Elastisys, Umeå, Sweden, Email: {li.wu, johan.tordsson, erik.elmroth}@elastisys.com
†Complex and Distributed IT-Systems Group, TU Berlin, Berlin, Germany, Email: odej.kao@tu-berlin.de
‡Department of Computing Science, Umeå University, Umeå, Sweden

***Abstract*—Software architecture is undergoing a transition from monolithic architectures to microservices to achieve resilience, agility and scalability in software development. However, with microservices it is difficult to diagnose performance issues due to technology heterogeneity, large number of microservices, and frequent updates to both software features and infrastructure. This paper presents MicroRCA, a system to locate root causes of performance issues in microservices. MicroRCA infers root causes in real time by correlating application performance symptoms with corresponding system resource utilization, without any application instrumentation. The root cause localization is based on an attributed graph that model anomaly propagation across services and machines. Our experimental evaluation where common anomalies are injected to a microservice benchmark running in a Kubernetes cluster shows that MicroRCA locates root causes well, with 89% precision and 97% mean average precision, outperforming several state-of-the-art methods.**

***Index Terms*—root cause analysis, performance degradation, microservices**

I. Introduction

More and more applications are using microservices architectures (MSA) in domains such as internet of things (IoT) [1], mobile and cloud [2], to build large-scale systems that are more resilient, robust and better adapted to dynamic customer requirements. With MSA, an application is decomposed into self-contained and independently deployable services with lightweight intercommunication [3].

To operate microservices reliably and with high uptime, performance issues must be detected quickly and their root causes pinpointed. However, it is difficult to achieve this in microservices systems due to the following challenges: *1) complex dependencies:* the number of services can often be hundreds, or thousands, (e.g., Uber has deployed 4000 microservices [4]). Consequently, the dependencies among services are much more complex than for traditional distributed systems. A performance degradation from one service can propagate widely and cause multiple alarms, making it difficult to locate the root causes; *2) numerous metrics:* the number of monitoring metrics available is very high. According to [5], Netflix exposes 2 million metrics and Uber exposes 500 million metrics. It would cause a significant overhead if all these metrics were to be used for performance issue diagnosis; *3) heterogeneous services:* technology heterogeneity [6], one key benefit of MSA, enables development teams to use different programming languages and technology stacks for their services. However, performance anomalies manifest differently for different technology stacks, making it hard to detect performance issues and locate root causes; *4) frequent updates:* microservices are frequently updated to meet customers' requirements, (e.g., Netflix updates thousands of times per day [7]). This highly dynamic environment aggravates the difficulty in root cause localization.

To date, many studies have been conducted on root cause diagnostics in distributed systems, clouds and microservices. These either require the application to be instrumented (e.g., [8], [9]) or numerous metrics to be analyzed (e.g., [5], [10]). A third class of approaches [11]–[13] avoid these limitations by building a causality graph and inferring the causes along the graph based on application-level metrics. With this approach, potential root causes are commonly ranked through the correlation between back-end services and front-end services. However, this may fail to identify faulty services that have little or no impact on front-end services.

In this paper, we propose a new system, MicroRCA[1], to locate root causes of performance issues in microservices. MicroRCA is an application-agnostic system designed for container-based microservices environments. It collects application and system levels metrics continuously and detects anomaly on SLO (Service Level Objective) metrics. Once an anomaly is detected, MicroRCA constructs an attributed graph with services and hosts to model the anomaly propagation among services. This graph does not only include the service call paths but also include services collocated on the same (virtual) machines. MicroRCA correlates anomaly symptoms of communicating services with relevant resource utilization to infer the potential abnormal services and ranks the potential root causes. With the correlation of service anomalies and resource utilization, MicroRCA can identify abnormal non-compute intensive services that have non-obvious service anomaly symptoms, and mitigate the effect of false alarms to root cause localization. We evaluate MicroRCA by injecting various anomalies to the Sock-shop[2] microservice benchmark deployed on Kubernetes running in Google Cloud Engine

[1]MicroRCA stands for **Micro**services **R**oot **C**ause **A**nalysis
[2]Sock-shop - https://microservices-demo.github.io/

978-1-7281-4973-8/20/$31.00 © 2020 IEEE

(GCE)[3]. The results show that MicroRCA achieves a good diagnosis result, with 89% in precision and 97% in mean average precision (MAP), which is higher than several state-of-the-art methods.

In summary, our contributions are threefold:

- We propose an attributed graph with service and host nodes to model the anomaly propagation in container-based microservices environments. Our approach is purely based on metrics collected at application and system levels and requires no application instrumentation.
- We provide a method to identify the anomalous services by correlating service performance symptoms with corresponding resource utilization, which adapts well to the heterogeneity of microservices.
- We evaluate MicroRCA by locating root causes from different types of faults and different kinds of faulty services. On average of 95 test scenarios, it achieves 13% precision improvement over the baseline methods.

The remainder of this paper is organized as follows. Related work is discussed in Section II. Section III gives an overview of the MicroRCA system. Root cause localization procedures are detailed in Section IV. Section V describes the experimental evaluation and Section VI concludes the paper.

II. Related Work

In recent years, many solutions have been proposed to identify root causes in distributed systems, clouds and microservices. Log-based approaches [14]–[17] build problem detection and identification models based on logs parsing. Even though log-based approaches can discovery more informational causes, they are hard to work in real time and require abnormal information to be hidden in logs. Similarly, trace-based approaches [8], [9], [18]–[23] gather information through complete tracing of the execution paths, then identify root causes through analyzing the deviation of latencies along the paths, e.g., based on machine learning. These approaches are very useful to debug distributed systems. However, it is a daunting task for developers to understand source code well enough to instrument tracing code.

In addition, there are many metrics-based approaches [5], [9]–[13], [24], [25], as well as this work. These use metrics from applications and/or additional infrastructure levels to construct a causality graph that is used to infer root causes. Seer [9] and Loud [10] use multiple-level metrics to identify root causes. Seer identifies the faulty services and which resource, like CPU overhead, that causes the service performance degradation. It is a proactive method which applies deep learning on a massive amount of data to identify root causes. This approach requires instrumentation of source code to get metrics, meanwhile, its performance may decrease when microservices are frequently updated. Loud identifies any faulty components that generate metrics. It uses the causality graph of KPI metrics from the anomaly detection system directly and locates the faulty components by means of different graph centrality algorithms. However, it requires anomaly detection to be performed on all gathered metrics, which would cause a significant overhead.

MonitorRank [13], Microscope [12] and CloudRanger [11] identify root causes based on application level metrics only. MonitorRank considers internal and external factors, and proposes a pseudo-anomaly clustering algorithm to classify external factors, then traverses the provided service call graph with a random walk algorithm to identify anomalous services. Microscope considers communicating and non-communicating dependencies between services and constructs a service causality graph to represent these two types of dependencies. Next, it traverses the constructed graph from the front-end service to find the root cause candidates and ranks them based on the metrics similarity between candidate and front-end service. CloudRanger constructs an impact graph with causal analysis and proposes a second-order random walk algorithm to locate root causes. All of these approaches achieve a good performance in identifying faulty services that impact a front-end service. However these methods commonly fail to identify root causes from backend services of the type that scarcely impact front-end services. Similar to these works, we also use a graph model, and localize root causes with an algorithm similar to random walk. However, we correlate anomalous performance symptoms with relevant resource utilization to comprehensively represent services anomaly, which improves the precision of root cause localization.

III. System Overview

In this section, we introduce MicroRCA and its components.

A. MicroRCA Overview

Figure 1 shows the overview of MicroRCA. Data collection module collects metrics from application and system levels. The application level metrics, in particular the response times of microservices, are used to detect performance issues, and metrics from both levels are used to locate root causes. Once anomalies are detected, the cause analysis engine constructs an attributed graph G with service and host nodes to represent the anomaly propagation paths. Next, the engine extracts an anomalous subgraph SG based on detected anomalies and infers which service is the most likely to cause the anomalies.

B. Data Collection

MicroRCA is designed to be application-agnostic. It collects application and system levels metrics from a service mesh [26] and monitoring system separately and stores them in a time series database. In container-based microservices, system-level metrics include container and host resource utilization, as illustrated by *container* and *host* in ”Model Overview” in Figure 1. Application level metrics include response times between two communicating services, etc.

C. Anomaly Detection

Anomaly detection is the starting point of root cause localization. MicroRCA leverages the unsupervised learning

[3]Google Cloud Engine - https://cloud.google.com/compute/

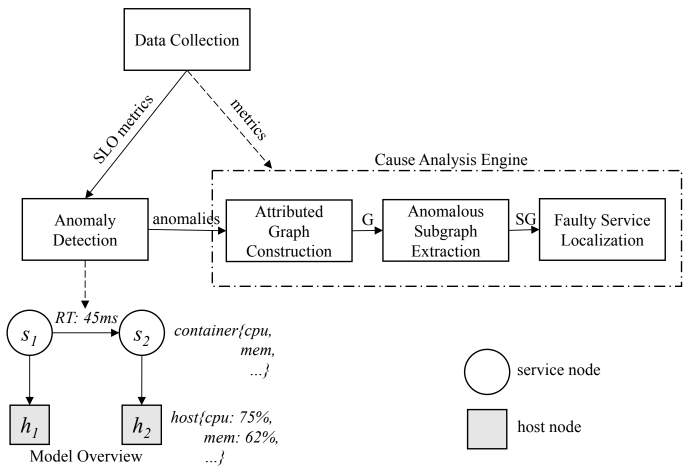

Figure 1. Overview of MicroRCA main components and root cause localization workflow.

algorithm Distance-Based online clustering BIRCH [27] as the anomaly detection method. We use the slow response time of a microservice as the definition of an anomaly.

D. Cause Analysis Engine

Once anomalies are detected, the cause analysis engine starts to locate root causes. The engine is composed of three main procedures: attributed graph construction, anomalous subgraph extraction, and faulty service localization. The engine constructs an attributed graph to represent the anomaly propagation through services and hosts. Next, it extracts the anomalous subgraph and locates the faulty services with a graph centrality algorithm named Personalized PageRank [28].

IV. Root Cause Localization

In this section, we describe the three procedures to identify root causes, namely attributed graph construction (Section IV-A), anomalous subgraph extraction (Section IV-B) and faulty services localization (Section IV-C).

A. Attributed Graph Construction

MicroRCA constructs an attributed graph to represent the anomaly propagation in microservices environments, which is based on the observation that anomalies propagate not only among services along the service call paths but also to services collocated on the same (virtual) machines [29], [30].

Our attributed graph consists of a set of services nodes $S = \{s_1, s_2, ..., s_k\}$ and host nodes $H = \{h_1, h_2, ..., h_m\}$ as shown in Figure 1. For each service node s_i, we add edges to all other service s_j it communicates with, and all hosts h_k it runs on.

We discover the graph nodes and their dynamic relationships by enumerating and parsing the metrics monitored at application and system levels. The graph nodes are interconnected as follows. When service s_i sends requests to service s_j, we add a directed edge from s_i to s_j. If a containerized service s_i is allocated in host h_j, we add a directed edge from s_i to h_j.

To construct the attributed graph, we use the metrics of a certain time frame before the anomaly was detected. We assume all microservices run reliably during this time frame and the collected metrics can precisely provide relationships among all communicating services and located hosts. Figure 2(b) shows an example of a constructed attributed graph from the initial microservices environment.

B. Anomalous Subgraph Extraction

After constructing the attributed graph, we proceed to extract the anomalous subgraph based on detected anomalies and assign anomaly-related attributes to the nodes. An anomalous subgraph SG is a connected subggraph that represents the anomaly propagation through services and hosts. Note that we use the response times between communicating services, which are the edges between service nodes in the attributed graph as the anomaly detection targets. Once the response time of an edge is determined to be an anomaly, we consider the edge as an *anomalous edge* and the origin of the edge as an *anomalous node*.

To obtain the anomalous subgraph, we first extract the anomalous service nodes. In Figure 2(a), the response times between (s_1, s_2), (s_2, s_3), (s_2, s_4) are anomalous. Therefore, we extract the anomalous edges and take the origins of the edges (s_2, s_3, s_4 in the example), as the anomalous service nodes. Figure 2(c) depicts the anomalous edges as dashed lines and anomalous services nodes as shadowed nodes. Next, we compute the average of anomalous response times for each anomalous service node, denoted as rt_a, to a node attribute. Finally, we add nodes and edges that are connected to anomalous service nodes. Figure 2(c) shows the extracted anomalous subgraph, where adjacent services nodes s_1 and s_5 and adjacent host nodes h_1 and h_2 probably are impacted.

C. Faulty Services Localization

Once an anomalous subgraph is extracted, we start to locate the most likely faulty services. We firstly compute the similarity between connected nodes by weighing the subgraph, then assign anomaly scores to anomalous service nodes, and finally locate root causes based on a graph centrality algorithm.

Anomalous Subgraph Weighing: Edge weights represent similarity between pairs of nodes. Similar to previous work [10]–[13], we use the Pearson correlation function to measure similarity, henceforth denoted as $corr(i, j)$. In particular, weights are computed in following three ways:

- *Weight of an anomalous edge* w^a. From anomalous subgraph extraction (Section IV-B), we get a list of anomalous edges. To each anomalous edge, we assign a constant value $\alpha \in (0, 1]$, which denote the anomaly detection confidence. If the real anomalies are correctly detected, we would set α higher, and vice versa. In Figure 2(d), weights $w_{(s_1,s_2)}$, $w_{(s_2,s_3)}$, $w_{(s_2,s_4)}$ are assigned α.
- *Weight between an anomalous service node and a normal service node* w^n. The weight is assigned the correlation between two response times: the response time between an anomalous service node and a normal service node, and the average anomalous response time rt_a of an anomalous service node. In Figure 2(d), the weight $w_{(s_1,s_3)}$ between the normal service node s_1 and the

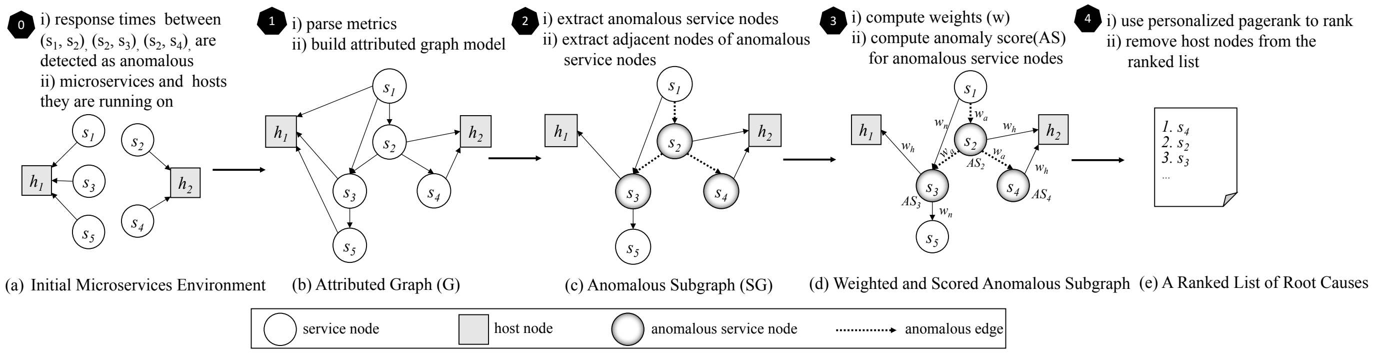
0 i) response times between (s1, s2), (s2, s3), (s2, s4), are detected as anomalous
ii) microservices and hosts they are running on
1 i) parse metrics
ii) build attributed graph model
2 i) extract anomalous service nodes
ii) extract adjacent nodes of anomalous service nodes
3 i) compute weights (w)
ii) compute anomaly score(AS) for anomalous service nodes
4 i) use personalized pagerank to rank
ii) remove host nodes from the ranked list
s1 s2 s3 s4 s5 h1 h2
wn wa wh AS2 AS3 AS4
1. s4
2. s2
3. s3
...
(a) Initial Microservices Environment
(b) Attributed Graph (G)
(c) Anomalous Subgraph (SG)
(d) Weighted and Scored Anomalous Subgraph
(e) A Ranked List of Root Causes
service node
host node
anomalous service node
anomalous edge

Figure 2. MicroRCA root cause localization procedures.

anomalous node s_3 is the correlation between the response time $rt_{(s_1,s_3)}$ between s_1 and s_3, and the response time $rt_{(s_2,s_3)}$ which is the average anomalous response time of s_3.

- *Weight between an anomalous service node and a normal host node* w^h. We use the maximum correlation coefficient between the average anomalous response time rt_a of an anomalous service node, and the host resource utilization metrics U_h, including CPU, memory, I/O, network, to represent the similarity between service anomaly symptoms and the corresponding resource utilization. Given that the response time of both normal and abnormal services have strong correlations with the host resource utilization, we take the average of in-edge weights w_{in} as a factor of the similarity. Thus, w^h_{ij} between service node i and host node j can be formulated as:

$$w^h_{ij} = \max_{k:u_k \in U_h(j)} corr(u_k, rt\_a(i)) \cdot \overline{w_I(i)} \qquad (1)$$

To summarize, the weight w_{ij} between node i and node j is given by Equation 2, where $rt\_a(i)$ denotes the average anomalous response time of anomalous service node i. Figure 2(d) shows these three types of weights along the edges.

$$w_{ij} = \begin{cases} \alpha, & \text{if } rt_{(i,j)} \text{ is anomalous,} \\ corr(rt_{(i,j)}, rt\_a(j)), & \text{if } i \in S \text{ and normal,} \\ corr(rt_{(i,j)}, rt\_a(i)), & \text{if } j \in S \text{ and normal,} \\ \text{as per Equation 1} & \text{if } j \in H \text{ and normal.} \end{cases} \qquad (2)$$

The procedure of anomalous subgraph weighing is presented in Algorithm 1. In this algorithm, we iterate over the anomalous nodes and compute the weights for in-edges in lines L1-L8 and for out-edges in lines L9-L15.

Assigning Service Anomaly Score: We calculate anomaly scores for anomalous service nodes and assign them to a node attribute, denoted as $AS \in [0, 1]$.

To quantify the anomaly, we take an average weight of service node $\overline{w(s_j)}$ that indicates the impact to linked nodes. Furthermore, In container-based microservices, we assume container resource utilization is correlated with service performance. We thus complement the service anomaly with the maximum correlation coefficient between the average anomalous response time rt_a of the anomalous service node,

Algorithm 1: Anomalous Subgraph Weighing

Input: Anomalous subgraph SG, anomalous edges, anomalous nodes, anomaly response time rt_a, host metrics U_h, response times of edges rt

Output: Weighted SG

```
1  for node $v_j$ in anomaly nodes do
2      for edge $e_{ij}$ in in-edges of $v_j$ do
3          if $rt_{(i,j)}$ in anomalous edges then
4              Assign $\alpha$ to $w_{ij}$ ;
5          else
6              Assign $corr(rt\_a(j), rt_{(i,j)})$ to $w_{ij}$
7          end
8      end
9      for edge $e_{jk}$ in out-edges of $v_j$ do
10         if node $v_k$ is service node then
11             Assign $corr(rt\_a(j), rt_{(j,k)})$ to $w_{jk}$ ;
12         else
13             Assign
                 $avg(w_{in}(j)) \times max(corr(rt\_a(j), U_h(k)))$
                 to $w_{jk}$
14         end
15     end
16 end
17 return Weighted $SG$
```

and its container resource utilization U_c. To summarize, given anomalous service node s_j, the anomaly score $AS(s_j)$ is defined as:

$$AS(s_j) = \overline{w(s_j)} \cdot \max_{k:u_k \in U_c(s_j)} corr(u_k, rt\_a(s_j)) \qquad (3)$$

Localizing Faulty Services: We locate the faulty services from the anomalous subgraph with a graph centrality algorithm - Personalized PageRank [28], which proved a good performance in capturing anomaly propagation in previous work [10], [11], [13], [29]. In Personalized PageRank, the Personalized PageRank vector (PPV) $\boldsymbol{v}$ is regarded as the root cause score for each node. To compute PPV, we first define $\boldsymbol{P}$ as the transition probability matrix, where $P_{ij} = \frac{w_{ij}}{\sum_j w_{ij}}$ if node i links to node j, and $P_{ij} = 0$ otherwise. A preference vector $\boldsymbol{u}$ denotes the preference of nodes, which we assign

Table I
HARDWARE AND SOFTWARE CONFIGURATION USED IN EXPERIMENTS.

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

the value of anomaly scores of the nodes in our method. This way, the anomaly-related service nodes are visited more frequently when the random teleportation occurs. Formally, the personalized PageRank equation [28] is defined as:

$$\boldsymbol{v} = (1-c)\boldsymbol{P}\boldsymbol{v} + c\boldsymbol{u} \tag{4}$$

where $c \in (0,1)$ is the teleportation probability, indicating that each step jumps back to a random node with probability c, and with probability $1-c$ continues forth along the graph. Typically $c = 0.15$ [28]. After ranking, we removed the host nodes from the ranked list as MicroRCA is designed to locate faulty services. As the link between service nodes in the anomalous subgraph represents the service call-callee relationship, we need to reverse the edges before running the localization algorithm. We give an example of the ranked list of root causes in Figure 2(e).

V. EXPERIMENTAL EVALUATION

In this section, we present the experimental setup, experimental results, a comparison with state-of-the-art methods, and discuss the characteristics of our approach.

A. Experimental Setup

Testbed: We evaluate MicroRCA in a testbed established in Google Cloud Engine (GCE) where we set up a Kubernetes cluster, deploy the monitoring system and service mesh, and run the benchmark named Sock-shop[2]. There are four worker nodes and one master node in the cluster, three of the worker nodes are dedicated to microservices and one for data collection. In addition, one server outside of the cluster runs the workload generator. Table I describes the detail configuration of the hardware and software in the testbed.

Benchmark: Sock-shop[2] is a microservice demo application that simulates an e-commerce website that sells socks. It is a widely used microservice benchmark designed to aid demonstration and testing of microservices and cloud-native technologies. Sock-shop consists of 13 microservices, which are implemented in heterogeneous technologies and intercommunicate using REST over HTTP. In particular, *front-end* serves as the entry point for user requests; *catalogue* provides a sock catalogue and product information; *carts* holds shopping carts; *user* provides the user authentication and store user accounts, including paymenet cards and addresses; *orders* places orders from *carts* after user log-in through the *user* service, then process the payment and shipping from the *payment* and *shipping* services separately. To each microservice,

Table II
REQUEST RATES SENT TO MICROSERVICES.

Microservices	front-end	catalogue	user	carts	orders
Concurrent users	100	100	100	100	100
Request rate(/s)	200	300	50	50	20

Table III
THE DETAIL OF INJECTED FAULTS.

Microservices	front-end	catalogue	user	carts	orders	payment	shipping
Latency(ms)	200	200	200	200	200	200	200
CPU Hog(vcpu*%)	-	2*95	2*95	2*95	2*95	2*99	2*95
Memory Leak(vm*MB)	-	2*1024	2*1024	1*2048	1*2048	2*1024	2*1024

we limit the CPU resource to 1vCPU and memory to 1GB. The replication factor of each microservice is set with 1.

Workload Generator: We develop a workload generator using Locust[4], a distributed, open-source load testing tool that simulates concurrent users in an application. The workload is selected to reflect real user behavior, e.g., more requests are sent to the entry points *front-end* and *catalogue*, and fewer to the shopping *carts*, *user* and *orders* services. We distribute requests to *front-end*, *orders*, *catalogue*, *user*, *carts* with five locust slaves, and provision 500 users that in total generate about 600 queries per second to sock-shop. The request rate of each microservice is listed in Table II.

Data Collection: We use the istio[5] service mesh, which in term uses Prometheus[6], to collect service-level and container-level metrics and node-exporter[7] to collect host-level metrics. Prometheus is configured to collect metrics every 5 seconds and sends the collected data to MicroRCA. In service-level, we collect response time between each pair of services. In both container-level and host-level, we collect CPU usage, memory usage, and the size of total sent bytes.

Faults Injection: Our method is applicable to any type of anomaly that manifests itself as increased microservice response time. In this evaluation, we inject three types of faults commonly used in the evaluation of the state-of-the-art approaches [10], [12], [31] to sock-shop microservices to simulate the performance issue. (1) Latency, we use the *tc* [8] to delay the network packets; (2) CPU hog, we use *stress-ng* [9], a tool to load and stress compute system, to exhaust CPU resources. As microservice *payment* is non-compute intensive whose CPU usage is only 50mHz, we exhaust its CPU heavily with 99% usage. (3) Memory leak: we use *stress-ng* to allocate memory continuously. As microservice *carts* and *orders* are CPU and memory intensive services, and memory leak causes CPU overhead [27], we only provision 1 virtual machine. The details of the injected faults are described in Table III.

To inject performance issues in microservices, we customize the existing sock-shop docker images with installing above faults injection tools. Each fault lasts 1 minute. To increase

[4]Locust - https://locust.io/
[5]Istio - https://istio.io/
[6]Prometheus - https://prometheus.io/
[7]Node-exporter - https://github.com/prometheus/node$_{exporter}$
[8]tc - https://linux.die.net/man/8/tc
[9]stress-ng - https://kernel.ubuntu.com/ cking/stress-ng/

the generality, we repeat the injection process 5 times for each fault. This produces a total of 95 experiment cases.

Baseline Methods: We compare MicroRCA with some baseline methods as follows:

- *Random selection(RS)*: Random selection is a way that an operating team use without specific domain knowledge of the system. Every time, the operating team randomly selects one microservice from the uninvestigated microservices to investigate until they find the root cause.
- *MonitorRank* [13]: MonitorRank uses a customized random walk algorithm to identify the root cause. It is similar with personalized PageRank[28] we adopt in MicroRCA. As we only consider the internal faults in microservices, we do not implement the pseudo-anomaly clustering algorithm to identify the external factors. To implement MonitorRank, we use their customized random walker algorithm to identify root causes in the extracted anomalous subgraph .
- *Microscope* [12]: Microscope is another graph-based method to identify faulty services in microservices environment and it also takes sock-shop as the benchmark. To implement Microscope, we construct the services causality graph with paring the service-level metrics, then use their cause inference, which traverses the causality graph with the detected anomalous services nodes, to locate root causes.

The anomaly detection methods in MonitorRank and Microscope fail to detect anomalies, especially for non-obvious anomalies, like *payment* service. As we in our experiments aim to compare the root cause localization performance, we thus use the same results from our anomaly detection module.

Evaluation Metrics: To quantify the performance of each algorithm on a set of anomalies A, we use following metrics:

- *Precision at top k* denotes the probability that the top k results given by an algorithm include the real root cause, denoted as $PR@k$. A higher $PR@k$ score, especially for small values of k, represents the algorithm correctly identifies the root cause, Let $R[i]$ be the rank of each cause and v_{rc} be the set of root causes. More formally, $PR@k$ is defined on a set of given anomalies A as:

$$PR@k = \frac{1}{|A|}\sum_{a\in A}\frac{\sum_{i<k}(R[i]\in v_{rc})}{(min(k,|v_{rc}|))} \tag{5}$$

- *Mean Average Precision* (MAP) quantifies the overall performance of an algorithm, where N is the number of microservices:

$$MAP = \frac{1}{|A|}\sum_{a\in A}\sum_{1\le k\le N} PR@k. \tag{6}$$

B. Experimental Results

Figure 3 shows the results of our proposed root cause localization method for different types of faults and sock-shop microservices. We observe that all services achieve a MAP in the range of 80%-100% and an average PR@1 over 80% except *shipping* and *payment*.

Table IV
PERFORMANCE OF MICRORCA.

microservics name	front-end	orders	catalogue	user	carts	shipping	payment	average
Latency								
PR@1	1.0	1.0	1.0	0.6	1.0	0.6	1.0	0.89
PR@3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
MAP	1.0	1.0	1.0	0.91	1.0	0.89	1.0	0.97
CPU Hog								
PR@1	-	1.0	1.0	1.0	0.8	1.0	0.6	0.9
PR@3	-	1.0	1.0	1.0	1.0	1.0	1.0	1.0
MAP	-	1.0	1.0	1.0	0.94	1.0	0.89	0.97
Memory Leak								
PR@1	-	1.0	0.8	1.0	1.0	0.8	0.8	0.9
PR@3	-	1.0	1.0	1.0	1.0	1.0	1.0	1.0
MAP	-	1.0	0.97	1.0	1.0	0.94	0.94	0.975

The average precision of *shipping* and *payment* that is lower than other services is likely for three reasons. First, *payment* is a non-compute intensive service, even though we exhaust the resource of CPU and memory, its response time is scarcely impacted. Second, there are few requests to *shipping* and *payment* and they do not request any other services, which makes the response time increase less obviously than other faulty services. Third, in order to detect anomalies in their experimental cases, we use a small threshold in anomaly detection module, which causes more false alarms.

Table IV demonstrates the performance of MicroRCA in different types of faults and microservices. It shows that MicroRCA can achieve almost 90% in terms of PR@1 and effectively locate all root causes in the top three faulty services.

C. Comparisons

To evaluate the performance of MicroRCA further, we apply it and the baseline methods on all experimental cases.

We compare the overall performance of all methods and their performances in identifying different types of faults. Table V shows the performance, in terms of PR@1, PR@3 and MAP, for all methods. We can observe that MicroRCA outperforms the baseline methods in overall. In particular, MicroRCA achieves a precision of 89% and MAP of 97%, which are at least 13% and 15% higher than the baseline methods separately. In general, Microscope performs well in CPU hog and memory leak when the anomalies are detected correctly but worse in fault latency when more false alarms are detected. However, MicroRCA performs well in all types of faults, and achieves average improvement of 24% in MAP comparing to MonitorRank and Microscope.

Next, we compare the performance of each method on different microservices. Figure 4 shows the comparison results, in terms of PR@1, PR@3 and MAP, on different services. We can see that MonitorRank performs well in identify dominating nodes which have large degrees, like microservice *orders*. However it fails to identify leaf nodes, like microservice *payment*. This is because MonitorRank calculates the similarity based on the correlation between front-end services and back-end services. The anomaly of microservice *payment* decreases the correlation during propagation and thus the root cause localization fails. On the contrary, Microscope performs better in identifying leaf nodes, such as the microservice *payment*,

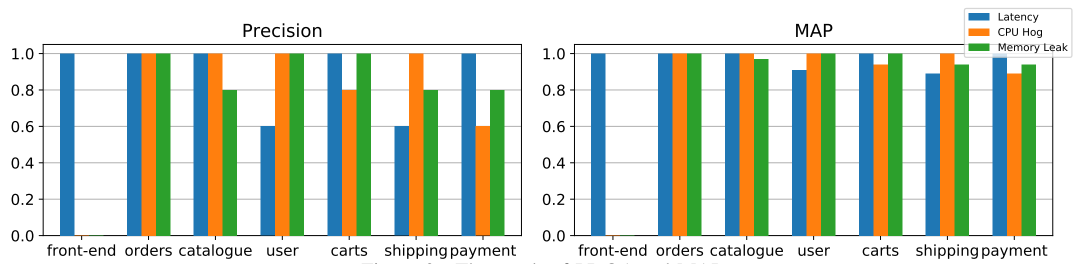

Figure 3. The result of PR@1 and MAP.

Table V
PERFORMANCE OF EACH ALGORITHM.

Metric	RS	MonitorRank	Microscope	MicroRCA	Improvement to MonitorRank (%)	Improvement to Microscope(%)
Overall						
PR@1	0.21	0.41	0.79	0.89	118	13.3
PR@3	0.46	0.65	0.86	1.0	53.2	15.9
MAP	0.58	0.73	0.85	0.97	33.7	14.7
Latency						
PR@1	0.17	0.23	0.66	0.89	287	34.8
PR@3	0.46	0.66	0.71	1.0	51.5	40.8
MAP	0.58	0.73	0.7	0.97	32.9	38.6
CPU Hog						
PR@1	0.23	0.6	0.87	0.9	50	3.5
PR@3	0.5	0.67	0.93	1.0	49.3	7.5
MAP	0.61	0.77	0.92	0.97	26	5.4
Memory Leak						
PR@1	0.23	0.43	0.87	0.9	109	3.5
PR@3	0.37	0.63	0.97	1.0	58.7	3
MAP	0.57	0.68	0.95	0.98	44.1	3.2

but worse in identify dominating node, like microservice *orders*. This is because Microscope traverses the graph based on the detected anomalies and put the anomalous child nodes into a list of potential causes, which makes it fail to identify the dominating nodes when alarms are reported from child nodes. Comparing to MonitorRank and Microscope, MicroRCA generally has a good performance for all the services, no matter dominating nodes or leaf nodes.

Finally, we compare the root cause localization performance of each method against the anomaly detection results to evaluate their sensitivity to the performance of anomaly detection. We compute the F1-score of each experimental case and count the number of different ranks of all experiment cases to different F1-scores. F1-score is computed as the harmonic mean of precision and recall: $2 \times \frac{precision \times recall}{precision + recall}$. In our experiment, we only inject one fault to one microservice, which means only one root cause in a case. Thus in each case, the recall is 1 if the root cause is in the ranked list, otherwise 0. Precision is the number of detected root causes divided by the number of detected anomalies. As RS(random selection) is not related to anomaly detection results, we analyze MonitorRank, Microscope and our MicroRCA only.

Figure 5 shows the number of different ranks against F1-score for MonotorRank, Microscope and MicroRCA separately. To show the results compactly, we set rank with 10, which is higher than the total number of microservices, when the root cause is failed to locate. We can see all of them have a good performance when the faulty services are correctly detected. However, when false alarms are frequent, Microscope cannot identify root causes in some cases; MonitorRank can identify root causes, but root causes are always low-ranked;

Table VI
THE OVERHEAD OF MICRORCA.

Modules	Cost
Data collection	0.6 vCPU and 1511MB RAM
Anomaly Detection	0.01s (8 cores)
Attributed Graph Construction	3.3s (8 cores)
Root Cause Localization	0.03 (8 cores)

MicroRCA identifies the root cause correctly both in scenarios with high and low F1-score.In particular, MicroRCA identifies 13 out of 18 faults in top 1 when F1-score is less than 0.4, which is higher than the other two methods.

D. Discussion

Here we discuss the overhead and sensitivity of our system.

Overhead: The overhead of MicroRCA on the microservices system is mostly caused by the data collection module, which collects the application-level and system-level metrics continuously. Table VI shows the overhead of data collection and the execution time of modules in MicroRCA. We can see that the execution time of MicroRCA is short enough to run in real-time given a data collection interval of 5 seconds. However the overhead of the data collection module is a little high. In the future, we would like to explore lightweight monitoring tools to reduce this overhead.

Sensitivity: To evaluate the sensitivity of MicroRCA to the anomaly detection confidence(α), which is assigned to the anomalous edges in weighing anomalous subgraph (section IV-C). We analyze the performance of MicroRCA with different values of α. Figure 6 shows the performance of MicroRCA, in terms of overall PR@k(k=1,2,3) and MAP of different types of faults over all the test cases, when α ranges from 0.15 to 1.0. We can see that the performance of MicroRCA changes with the weight assigned to the anomalous nodes. PR@1 increases to the maximum when $\alpha = 0.55$ and drops when $\alpha \geqslant 0.7$; PR@3 is always 1 when α is less than 0.55, and drops after that. MAP of all type of faults are relatively stable. and keeps over 96%. In this case, we choose 0.55 as the weight of anomalous edges where the PR@1 and PR@3 are the maximum. In practise, as different anomaly detection methods have different performances, we need to tune the anomaly detection confidence(α) according to the anomaly detection module.

Furthermore, we analyze the sensitivity to anomaly detection results with different thresholds in anomaly detection module. Figure 7 shows the MAP of MicroRCA and other two baseline methods in solid lines and F1-score in dash line

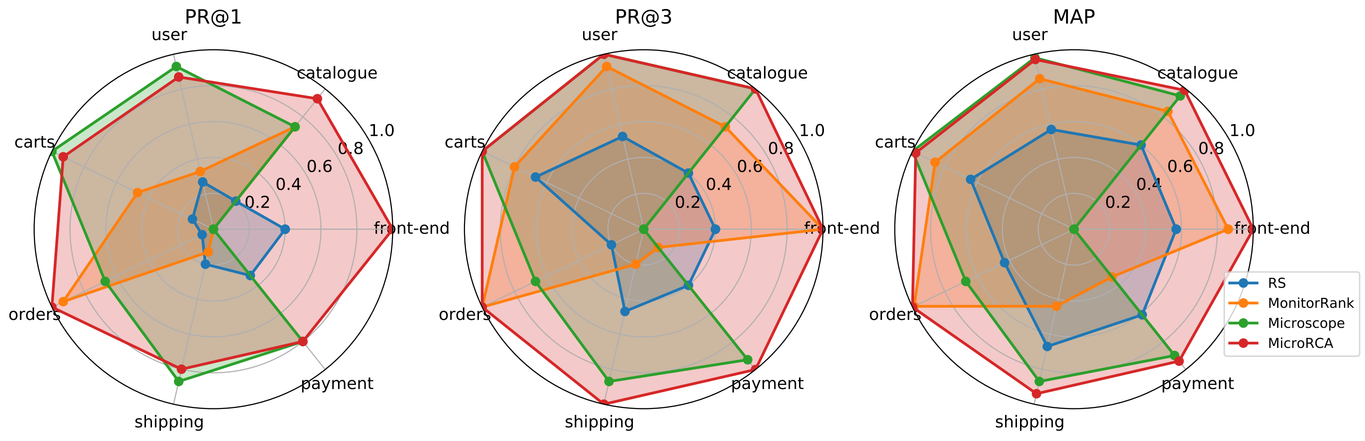

Figure 4. The comparison results of PR@1,PR@3 and MAP for different microservices.

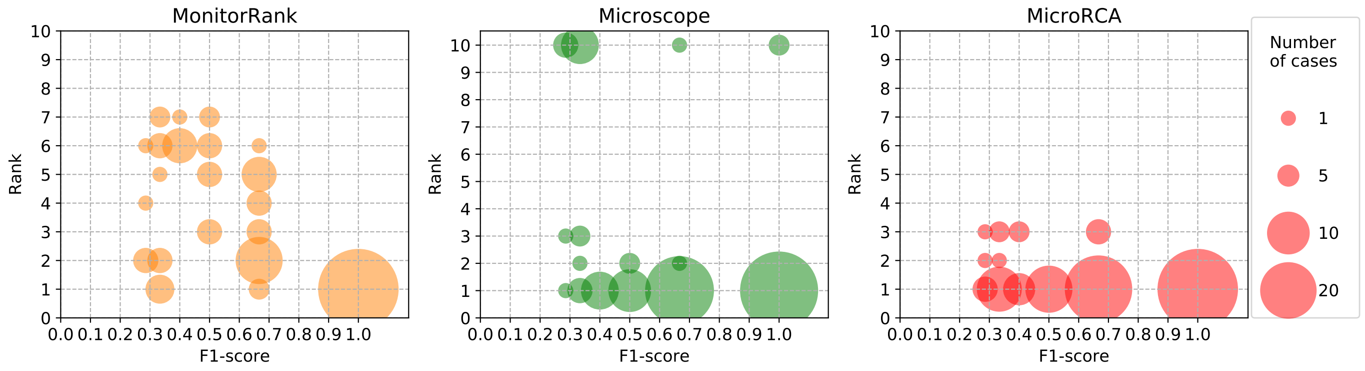

Figure 5. Root cause rank against anomaly detection F1-Score.

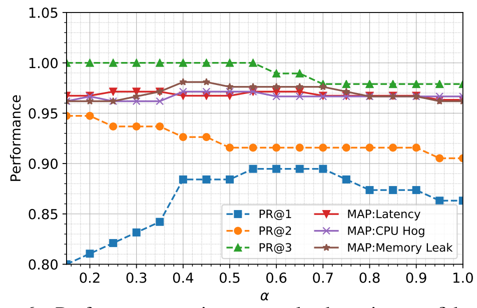

Figure 6. Performance against anomaly detection confidence(α).

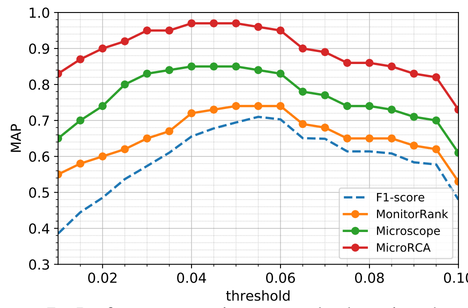

Figure 7. Performance against anomaly detection threshold.

when threshold ranges from 0.01 to 0.1. Note that we set α to the constant 0.55 when MicroRCA regularly performs well. We can see that the three methods share the similar pattern with F1-score, and MicroRCA performs well in a range from 0.02 to 0.065, which is wider than other two methods. As all three methods performs well when threshold is 0.045, we use it in the anomaly detection module for all above analysis.

VI. Conclusions

In this paper, we proposed a new application-agnostic system, MicroRCA, for root cause localization of performance anomalies in container-based microservices. It constructs an attributed graph model and correlates service anomalous performance symptoms with corresponding resource utilization to infer the anomalous microservices. Experimental evaluations with a microservice benchmark show that MicroRCA achieves 89% in precision and 97% in mean average precision (MAP). In particular, our method improves the precision in identifying root causes from both dominating services with large degrees and leaf services with non-obvious anomaly symptoms, where the state-of-the-art methods fall short.

Acknowledgment

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] B. Butzin, F. Golatowski, and D. Timmermann, "Microservices approach for the internet of things", in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, pp. 1–6.

[2] P. D. Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption", in *2017 IEEE International Conference on Software Architecture (ICSA)*, 2017, pp. 21–30.

[3] S. Newman, *Building Microservices*. 2015.

[4] *How Uber Is Monitoring 4,000 Microservices with Its Open Sourced Prometheus Platform*, (accessed: 03.07.2019).

[5] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer, "Sieve: Actionable insights from monitored metrics in distributed systems", in *Middleware '17*, 2017, pp. 14–27.

[6] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow", *Book: Present and Ulterior Software Engineering*, pp. 195–216, 2017. arXiv: 1606.04036.

[7] *Why Netflix, Amazon, and Apple Care About Microservices*, (accessed: 03.07.2019).

[8] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework", in *NSDI'07*, 2007, pp. 20–20.

[9] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices", in *ASPLOS '19*, 2019, pp. 19–33.

[10] L. Mariani, C. Monni, M. Pezzé, O. Riganelli, and R. Xin, "Localizing faults in cloud systems", in *ICST*, 2018, pp. 262–273.

[11] P. Wang, J. Xu, M. Ma, W. Lin, D. Pan, Y. Wang, and P. Chen, "Cloudranger: Root cause identification for cloud native systems", in *CCGRID*, 2018, pp. 492–502.

[12] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments", in *Service-Oriented Computing*, C. Pahl, M. Vukovic, J. Yin, and Q. Yu, Eds., 2018, pp. 3–20.

[13] M. Kim, R. Sumbaly, and S. Shah, "Root cause detection in a service-oriented architecture", *SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 1, pp. 93–104, 2013.

[14] J. Xu, P. Chen, L. Yang, F. Meng, and P. Wang, "Logdc: Problem diagnosis for declartively-deployed cloud applications with log", in *2017 IEEE 14th International Conference on e-Business Engineering (ICEBE)*, 2017, pp. 282–287.

[15] T. Jia, P. Chen, L. Yang, Y. Li, F. Meng, and J. Xu, "An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services", in *2017 IEEE International Conference on Web Services (ICWS)*, 2017, pp. 25–32.

[16] I. Weber, C. Li, L. Bass, X. Xu, and L. Zhu, "Discovering and visualizing operations processes with pod-discovery and pod-viz", in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 537–544.

[17] Jiaqi Tan, Xinghao Pan, E. Marinelli, S. Kavulya, R. Gandhi, and P. Narasimhan, "Kahuna: Problem diagnosis for mapreduce-based cloud computing environments", in *2010 IEEE Network Operations and Management Symposium - NOMS 2010*, 2010, pp. 112–119.

[18] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study", *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[19] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services", in *OSDI'14*, 2014, pp. 217–231.

[20] H. Mi, H. Wang, Y. Zhou, M. R. Lyu, and H. Cai, "Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 6, pp. 1245–1255, 2013.

[21] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer, "Failure diagnosis using decision trees", in *International Conference on Autonomic Computing, 2004. Proceedings.*, 2004, pp. 36–43.

[22] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling", in *OSDI'04*, 2004, pp. 18–18.

[23] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services", in *Proceedings International Conference on Dependable Systems and Networks*, 2002, pp. 595–604.

[24] P. Chen, Y. Qi, and D. Hou, "Causeinfer: Automated end-to-end performance diagnosis with hierarchical causality graph in cloud environment", *IEEE Transactions on Services Computing*, vol. 12, no. 2, pp. 214–230, 2019.

[25] B. Sharma, P. Jayachandran, A. Verma, and C. R. Das, "Cloudpd: Problem determination and diagnosis in shared dynamic clouds", in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–12.

[26] *What's a service mesh? And why do I need one?*, (accessed: 27.04.2019).

[27] A. Gulenko, F. Schmidt, A. Acker, M. Wallschläger, O. Kao, and F. Liu, "Detecting anomalous behavior of black-box services modeled with distance-based online clustering", in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 912–915.

[28] G. Jeh and J. Widom, "Scaling personalized web search", in *WWW '03*, 2003, pp. 271–279.

[29] J. Weng, J. H. Wang, J. Yang, and Y. Yang, "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] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification", *ACM Comput. Surv.*, vol. 48, no. 1, 4:1–4:35, 2015.

[31] H. Kang, H. Chen, and G. Jiang, "Peerwatch: A fault detection and diagnosis tool for virtualized consolidation systems", in *ICAC '10*, 2010, pp. 119–128.