MicroCBR: Case-Based Reasoning on Spatio-temporal Fault Knowledge Graph for Microservices Troubleshooting

Fengrui Liu[1,2], Yang Wang[1,2], Zhenyu Li[1,2], Rui Ren[1,2], Hongtao Guan[1,2], Xian Yu[3,4], Xiaofan Chen[4], and Gaogang Xie[2,5](✉)

[1] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
{liufengrui18z,wangyang2013,zyli,renrui2019,guanhongtao}@ict.ac.cn

[2] University of Chinese Academy of Sciences, Beijing, China
xie@cnic.cn

[3] Shenzhen Institutes of Advanced Technology, CAS, Shenzhen, China
yuxian@sangfor.com.cn

[4] Sangfor Technologies Inc., Shenzhen, China
chenxiaofan@sangfor.com.cn

[5] Computer Network Information Center, Chinese Academy of Sciences, Beijing, China

Abstract. With the growing market of cloud-native applications, microservices architectures are widely used for rapid and automated deployments, scaling, and management. However, behind the prosperity of microservices, diagnosing faults in numerous services has brought great complexities to operators. To tackle this, we present a microservices troubleshooting framework called MicroCBR, which makes use of history faults from a knowledge base to construct spatio-temporal knowledge graph offline, and then troubleshoot online through case-based reasoning. Compared to existing frameworks, MicroCBR (1) takes advantage of heterogeneous data to fingerprint the faults, (2) carefully extracts a spatio-temporal knowledge graph with only one sample for each fault, (3) can handle novel faults through hierarchical reasoning, and incrementally update it to the fault knowledge base thanks to case-based reasoning paradigm. Our framework is explainable to operators, they can easily locate the root causes and refer to historical solutions. We also conduct three different microservices architectures with fault experiments on Grid'5000 testbed, the results show that MicroCBR achieves 91% top-1 accuracy, and outperforms three state-of-the-art methods. We report success stories in a real cloud platform and the code is open-sourced.

Keywords: CBR · Knowledge graph · Microservices · Troubleshooting

1 Introduction

Cloud-native applications enable loosely coupled systems that are resilient, manageable, and observable. This technology embraces microservices, a popular architectural style for constructing modern applications in dynamic environments such as public, private, and hybrid clouds. The philosophy behind microservices is decoupling the resources from applications, where each unit has a single task and responsibility. Microservices architectures become increasingly complex with the growth of user demands, causing that fault diagnosis across numerous services exhaust the operators. Thus, an effective and automated troubleshooting method is required to offload the labor and reduce the mean time to repair (MTTR) of microservices.

We share several observations from a real cloud platform to clarify the challenges of microservices troubleshooting. First, the available data for fault diagnosis can be very different due to deployment, security, and performance requirements. Therefore, the methods [1,2] of using only homologous data are easy to fail when the required data is missing. Second, faults are rare, which brings huge obstacles for training-based methods [3,4], While these history faults are well maintained in a knowledge base, we should consider how to learn from them with only one sample for each fault. Third, we point out that the spatial topology of instances and the temporal order of anomalies can also effectively diagnose different faults. Further, in practical applications, the troubleshooting framework needs to be incrementally updated to accommodate novel fault cases. In addition, an explainable troubleshooting process and recommendation solutions for the emerging fault are important to operators.

To tackle the aforementioned challenges, we propose **MicroCBR**, a **c**ase-**b**ased **r**easoning (CBR) driven troubleshooting framework for **micro**services. In particular, MicroCBR constructs a fault spatio-temporal knowledge graph using heterogeneous data, which is embedded with temporal anomaly events sequences and spatial instances topologies. Explainable case-based reasoning is performed on the knowledge graph to diagnose emerging faults, followed by recommended solutions. The salient contributions of our work are summarized as follows:

– We fully take advantage of heterogeneous data, i.e. metrics, logs, traces, and commands. Comprehensive data entries can greatly enrich the fault fingerprints, thus expanding the scope of applicable targets and improving the troubleshooting accuracy.
– With retaining the physical and logical topologies of microservices instances, we innovatively embed anomaly events sequences into the fault knowledge graph. The analysis of spatial topology make diagnosing target instance without historical data possible, and temporal anomaly events sequences improve the troubleshooting accuracy with limited data.
– Our framework is not confined to retrieving the most similar historical case to an emerging fault, it can further assign a novel fault to a fault type by hierarchical reasoning. With CBR paradigm, we revise and retain the novel fault in the knowledge base to achieve incremental updates.
– We report experiments that compare to three state-of-the-art (SOTA) methods on Grid'5000 testbed. The results demonstrate that MicroCBR outperforms SOTA methods with 91% top-1 accuracy. Success stories from a real private cloud platform prove that our framework can troubleshoot the faults effectively and offload the labor of operators. The code[1] is also available.

[1] MicroCBR repository: https://github.com/Fengrui-Liu/MicroCBR.

2 Related Work

In this section, we study the strengths and weaknesses of existing methods that aim for microservices troubleshooting from three perspectives, i.e. data entries, graph-based analysis, and reasoning methods.

Data Entries. The most common work focus on homologous data and diagnosis faults by mining anomalies. [5,6] start from the logs. They collect anomaly scores from different log detectors and identify root causes using correlation analysis. Another great deal of efforts have been devoted to metrics analysis. [2,7] perform anomaly detection on structured time-series metrics, and further determine the faults according to the monitored targets. In addition, [1,8] turn their attention to the traces, they model the service invocations and assume that instances with abnormal latency are more likely to be root causes. However, these methods are limited to a narrow perspective with only using homologous data, and fail when required data is missing or faults affect multiple kinds of data. Recently, some work [9–11] take into account the combination of metrics and logs, showing the advantages of using heterogeneous data in industrial settings. We argue that they still have deficiencies in data entries, especially for the post-hoc commands anomalies.

Graph-Based Analysis. Graph is a popular and effective representation of entities and their relationships, researchers propose various views of graph construction to assist microservices troubleshooting. Causeinfer [12] regards physics instances as target entities while network connections as relationships, and [8,10] track invocations among different traces as a service call graph. [11,13] try to deduce event causality and build a graph to describe event relationships, with an underlying assumption that the order of anomalies can infer specific faults. Although different graph-based methods have strengths in modeling microservices, they still separate physics entities, services logic, and anomaly events sequences, which limits them to a single granularity. In our framework design, multi-level abstractions and a fusion of spatio-temporal heterogeneous data are used to construct the fault knowledge graph.

Reasoning. In recent years, the most popular faults diagnosis methods are those driven by machine learning. [4] takes advantage of variational autoencoder to detect anomalies and Seer [3] locates root causes using counterfactuals. Netrca [14] adopts an ensemble model to improve troubleshooting generalization. Most of these supervised methods suffer from labeling overhead in the training phase. They also have limited discussion on incremental model updates and the reusability when novel faults occur. In addition, unsupervised methods [1,9,13] utilize probabilistic graphical models, such as Bayesian networks. Nevertheless, from practical experience, the prior probability of a fault is hard to collect due to the bug fixes and the growth of maintenance experience. Based on this observation, probabilistic graphical models must be used with care. Case-based reasoning is different from the above. It follows a 4R paradigm which contains *Retrieve*, *Reuse*,*Revise* and *Retain*. In [15,16], authors introduce how CBR can be used for troubleshooting, and iSQUAD [17] integrates CBR to provide explanation for reasoning results.

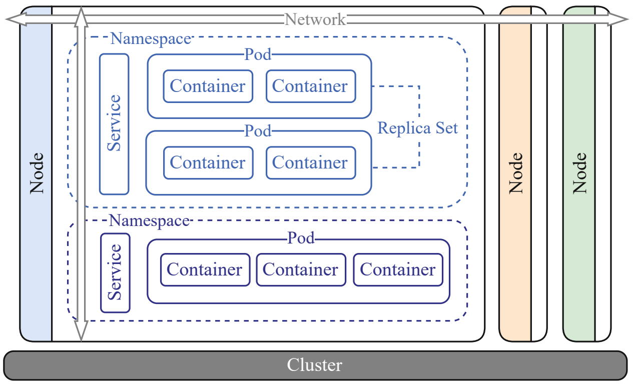


Fig. 1. Simplified microservice architecture example.

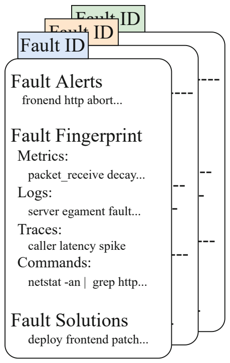


Fig. 2. Fault knowledge base.

3 Background and Motivation

This section first introduces the background of microservices troubleshooting with clarifying basic concepts. After that, we detail our motivation.

3.1 Background with Basic Concepts

Take Fig. 1 as a simplified example, we introduce the objects of microservices troubleshooting. The workloads run in **containers** at the lowest level. They are bundled together as **pods** which is the smallest deployable unit to create, schedule and manage. A **namespace** is a virtual cluster, it provides a mechanism for isolating groups of resources within a single **node**. In microservice deployment, the **replica set** is a common controller, which plays a central role in automatically orchestrating workloads. Since pods are constantly being created and destroyed, **services** provide a level of abstraction and act as an endpoint for the pods. Due to the scalability of microservices, a configuration management database (CMDB) is often used to manage their physical and logical topologies.

The **knowledge base** is a collection of all fault cases through long-term microservices maintenances. For each fault, it contains the fault alerts, a fault fingerprint, and the corresponding solutions, as shown in Fig. 2. Note that the **fault fingerprint** records all anomalies that are related to the fault. It is drawn from four types of heterogeneous data, i.e. metrics, logs, traces, and commands. **Metrics** are numeric representations of data over intervals of time. They are widely used in performance monitoring. **Logs** are timestamped records of discrete events. They tend to give more in-depth information than metrics, e.g., predefined events or program exceptions. **Traces** add critical visibility into the health of microservices end-to-end, which are introduced by caller and callee pairs. **Commands** are post-hoc supplements to the above data sources, which show great strength when required data is missing, e.g. security or performance limitations. While existing troubleshooting frameworks [1,2,5–8] turn a blind eye to commands. Note that the above four heterogeneous data sources are summarized from practical troubleshooting experiences. However, each fault fingerprint only includes the available and anomaly items, not necessarily for all.

3.2 Motivation

Microservices troubleshooting, also known as fault diagnosis, is aiming to diagnose inevitable faults from both physical and logical components. The highest priority of operators is to stabilize the system and avoid faults escalation. The motivation of this paper is to propose a troubleshooting framework for microservices, which can offload the labor by automated faults identification, minimize the losses by reducing MTTR, as well as bridge the gap between prior works and following practical constraints.

In practice, faults may recur because containers that have bugs can be deployed multiple times to different services, or because complicated scenario restrictions always lead to unexpected misconfigurations. If operators can quickly determine whether the emerging fault is similar to a previously-seen case, those known solutions may be reused to quickly fix the emerging fault. With this, several interesting observations guide us to design the troubleshooting framework.

First, we learn that the required data for diagnosing various faults can be very different [9–11]. In particular, we note that commands are widely used in practice, they should be adopted together with metrics, logs and traces to enrich the fault fingerprints. Second, the same fault data are always too scarce to train a supervised model [3,14] in reality. In our practical experiences, if an emerging fault can be solved by referring to the historical solutions, operators would hardly enrich it anymore. Supervised methods suffer from handling novel faults, and fail on incremental updates. Besides, the prior probability of each fault is difficult to determine after the bugs are fixed or the maintenance experiences growth. Thus, probabilistic graphical models [1,9,13], like the Bayesian network, should be used with care. Next, we focus on how to draw the fault fingerprints as accurately as possible in limited data. The effective answer is endowing spatio-temporal characteristics to the fingerprints.

Thanks to the above valuable practical observations, we are expected to design a microservices troubleshooting framework that can take advantage of heterogeneous data, and learn from one sample of each fault. In addition, it should be able to provide recommended solutions and handle novel faults. To tackle these challenges, we propose MicroCBR which uses case-based reasoning on spatio-temporal fault knowledge graph for microservices troubleshooting, and the details are described in the next section.

4 Troubleshooting Framework

In this section, we firstly introduce the overview workflow of our troubleshooting framework, followed by detailed descriptions of its main components.

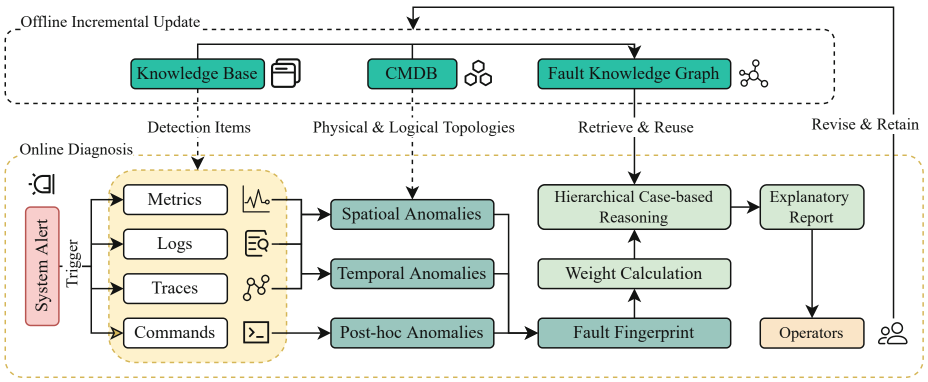


Fig. 3. Overview workflow of proposed MicroCBR troubleshooting framework.

4.1 Framework Overview

As Fig. 3 shows, our framework combines offline update with online diagnosis through case-based reasoning. For the offline part, we leverage the CMDB of target system to construct a graph that contains its physical and logical topologies. Then we embed the existing knowledge base into the graph, and enrich it into a spatio-temporal fault knowledge graph. An online troubleshooting workflow is usually triggered by a system alert. The universal set of fault fingerprints from existing knowledge base points out the objects to be detected. After various detecting methods applying on metrics, logs, traces and commands, we select those anomalies to depict a fingerprint of emerging fault. In order to distinguish the importance of different anomalies, we assign weights scores to them. After performing hierarchical case-based reasoning on constructed fault knowledge graph to realize the retrieve and reuse of existing fault cases, we provide a explanatory report with fault fingerprint and recommended solutions. The operators can revise the fault details and retain a novel fault in the knowledge base for incremental updates. Next we elaborate on each technology of the framework.

4.2 Spatio-Temporal Fault Knowledge Graph

In our framework, the fault knowledge graph is the cornerstone of fault diagnosis. The backbone of the knowledge graph is constructed from the CMDB of a target system. Its entities include logical nodes, namespaces, services topologies, and physical clusters. The spatial topologies of these entities are represented by links that indicate their affiliations, and the links of different services represent their call relation. On this foundation, we append the fingerprints from the knowledge base to the graph. For each fingerprint, its upstream is the anomaly entity, and its downstream is the corresponding fault. We further illustrate the power of temporal event sequences and spatial topologies for troubleshooting by giving a fault case respectively in Fig. 4. Note that these two analytical perspectives are not independent, and should be combined in some fault cases.

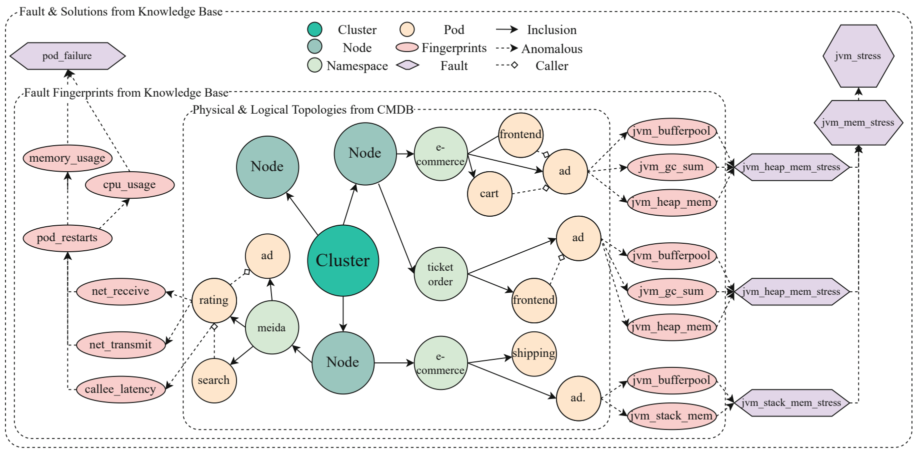


Fig. 4. A subgraph of spatio-temporal fault knowledge graph. The left pod failure fault fingerprint has temporal event sequences, while the right JVM memory stress fault shows us the spatial topologies and hierarchical reasoning for troubleshooting.

Temporal. A key insight of the fingerprints is that some anomaly events occur orderly. Having the same composition, anomaly events with different orders may point to different faults. The *rating pod* in Fig. 4 shows us a remarkable event sequence of its *pod failure*. At first, both the *net_ receive* and *net_ transmit* metrics have unexpected declines, accompanied by a significant increase in its *callee_ latency*. However, this symptom is too common in various faults to diagnose correctly. What inspires us is that there are a series of follow-ups. Its *pod_ restart* counter is incremented by one, followed by continuous increases in *memory_ usage* and *cpu_ usage*. These sequential events point out that the daemon is trying to restart the *rating pod*, and diagnose it as a *pod_failure* fault. The key challenge of embedding event sequences into the knowledge graph is mining their temporal order. For a fault fingerprint, supposing each anomaly a_i and its beginning time t_i is presented by a tuple (t_i, a_i). We ascending sort the anomalies according to their beginning time t_i. After that, an unsupervised DBSCAN [18] algorithm is performed on their time dimension for clustering, as $C_1, C_2, ..., C_m = \text{DBSCAN}([t_1, t_2, ..., t_i], n, \epsilon)$, with two important empirical parameters minimum points ($n = 1$) and neighborhood radius ($\epsilon = 15$s) in our settings. The positions m of clusters C_m on the time dimension are used as temporal order of a_i. Anomaly events in the same cluster share the same temporal order, as the *memory_ usage* and *cpu_ usage* in the above pod failure example.

Spatial. We turn to how spatial topology helps us with fault diagnosis when lacks historical data. A useful observation is that the entities with similar spatial topology can be used as references for anomaly detection. Due to the Java-based *advertising* services being widely deployed in different namespaces, we study its Java virtual machine (JVM) stress faults in Fig. 4. Owing to resources limitations, the pod under *ticket-order* namespace has a JVM stress fault during the startup phase, it is impossible to detect anomalies on related time series metrics without historical data. Similar *advertising* pods that work well in other namespaces can serve as normal references. It turns out that the target pod has a higher ratio of bufferpool, memory, and garbage collection usage than those normals. To determine those references, for a target instance i, we retrieve its one-hop neighbors n and form a subgraph $subg(i)$. Thanks to CMDB provides us the physical and logical topologies of microservices, we search the whole topology graph for reference instances r, *s.t.* $sim(subg(i), subg(r)) > \theta$, empirically findings that $sim(\cdot)$ can be calculated by Jaccard distance and $\theta = 0.7$.

Hierarchical. In some certain circumstances, like a novel fault that has never been seen before occurring in the target system, we are unable to retrieve a sufficiently similar case from the existing fault knowledge graph to reuse its solutions. Thus, we introduce a hierarchical abstraction of all known faults. Take the JVM stress fault in Figure 4 as an example. At the finest granularity, the knowledge base has only collected heap and stack memory stress faults. Once there occurs another kind of memory stress fault, metaspace overflow, similarity-based retrieving (introduced in Sect. 4.3) fails to assign it to any known faults with a predefined threshold. An ideal way is to classify this novel fault to a higher level, such as a JVM memory stress or a JVM stress fault. Although this is a coarse-grained diagnosis, it can also effectively reduce the scope of troubleshooting. After revising the fault fingerprint and solutions by operators, the novel fault is added to the knowledge base and finishes the incremental updates to the fault knowledge graph.

4.3 Fingerprinting the Fault

When an alarm triggers the troubleshooting workflow as Fig. 3 shows, the first step is to detect anomalies for the target system. All the items to be detected are from the fingerprints of the existing knowledge base. Here we list a series of anomaly detection methods that are used in practice. For metrics data, we detect their anomalies in a time series manner [19,20]. Log anomalies [21,22] can be reported from their template sequences and specific events. As for traces, according to [1,8], their anomalies are reflected in call latency. Thus we select their one-hop caller and callee latency to monitor. The commands are post-hoc detections for faults, they should be sent to different targets and collect returns to automate the troubleshooting workflow. Predefined rules are the best practice for commands owing to security and customization requirements.

The above methods usually represent the state of detected items in a binary way, a.k.a normal and anomaly. We introduce an algorithm to evaluate the weights of anomalies in a fault fingerprint. Suppose a knowledge base that contains k known fault fingerprints F. For each fingerprint F, it contains a collection of anomalous items f. The first consideration is frequency. It reflects the belief that the lower the frequency, the more important the word is, as

$$W_{freq}(f) = \frac{\sum_i^k \mathbb{1}_{[f \in i]}}{k} \tag{1}$$

Another factor to consider is the relatedness of anomalous items, which reflects in their degrees. The more numbers of different terms that co-occur with the candidate detection item, the more meaningless the item is likely to be. The length of co-occur list of f is $C_l(f) = \sum_i^k \sum_j^i \mathbb{1}_{[j, f \in i]}$, and the length of co-occur set of f is $C_s(f) = \sum_i^k \sum_j^i \mathbb{1}_{\{j, f \in i\}}$ which only counts the unique detection items. We define the relatedness weight of f as

$$W_{rel}(f) = 1 + \frac{C_s(f)}{C_l(f)} + \frac{C_s(f)}{max(W_{freq}(\cdot))} \tag{2}$$

We heuristically combine the two important factors into a single measure as

$$W(f) = \frac{W_{rel}(f)}{\frac{W_{freq}(f)}{W_{rel}(f)} + C_l(f)} \tag{3}$$

The motivation of this equation is to assign high weights to anomalous items that appear infrequently as long as the item is relevant. Those anomalies with high weights have great contributions to fault discrimination.

4.4 Case-Based Reasoning

After fingerprinting the emerging fault, we show how to use the paradigm of case-based reasoning to complete once fault diagnosis.

Retrieve. The key idea of this step is to select the most similar known fault F from the knowledge base to the emerging fault F'. As we have discussed the temporal characteristic of fault fingerprints in Sect. 4.2, it can be solved by converting to a weighted longest common subsequence (WLCS) problem [23]. The similarity can be defined in a weighted bitwise way:

$$Sim(F', F) = \frac{\sum_f^{F'} W(f) \mathbb{1}_{[f \in F \cap WLCS(F, F')]}}{max(\sum_f^{F'} W(f) \mathbb{1}_{[f \in F']}, \sum_f^{F} W(f) \mathbb{1}_{[f \in F]})} \tag{4}$$

Reuse. For each fault F in the knowledge base, we can get its similarity score to emerging fault F'. The solutions of a fault F that has the highest similarity score can be used as references for troubleshooting. Furthermore, to ensure the recall rate for similar faults, a common practice is to sort their scores in descending and select top-k for recommendations. At the same time, we should avoid recommending too many options under the consideration of labor cost. Thus, we argue the power of top-2 in experiments.

Revise & Retain. The weighted detection items, similar faults, and recommended solutions together form the explanatory report. The good benefits are that the operators can understand the root causes of an emerging fault. Some anomalies that hardly distinguish various faults can be eliminated from the fingerprint by operators. After that, the revised fault fingerprint is retained within the knowledge base.

5 Evaluation

This section presents the detailed results of extensive experiments to evaluate our troubleshooting framework by answering the following questions:

Q1. What is the accuracy of MicroCBR compared to SOTA baselines?
Q2. How much accuracy is affected by each component of the framework?
Q3. How MicroCBR is affected by the data scale?
Q4. Does it offload the labor in practical applications?

5.1 Evaluation Setup

We configure a testbed in Grid'5000 [24] and deploy three open-source microservices architectures to evaluate our framework. Besides, three selected SOTA methods for comparison are introduced in this section.

Testbed & Microservices. We prepare 3 private nodes in the Nancy site of Grid'5000, with 16GB RAM and 4 cores Xeon E5-2650 CPU for each node. Online-Boutique[2](OB), Sock-Shop[3](SS), and Train-Ticket[4](TT) are deployed respectively, which have different microservice architectures. Each service of them is deployed with multiple instances. We continuously run a workload generator to simulate the real-world user access behaviors.

Fault injection & Data collection. To explore the faults of different microservices architectures, we select 31 faults at both pods and containers from Chaos-Mesh[5]. Besides, 10 misconfiguration faults have been prepared manually. These faults can be categorized in a high hierarchy with 6 types, including network faults, stress scenarios, etc. For different faults, we carefully analyze their anomalies and collect them to form a knowledge base. Then, we successfully inject them into different instances and finish 3572 experiments. For each injection, we collect the anomalies from metrics, logs, traces, and commands within 10 min.

Baselines. We select three SOTA methods, Bayesian-based CloudRCA [9], case-based CloodCBR [15] and graph-based GraphRCA [10] for comparison. To make these methods suitable for our experimental settings, we assume that CloudRCA treats all faults with the same prior probability, CloodCBR adopts equal matching as the similarity calculation strategy, and GraphRCA needs no additional assumptions. Another thing to note is that all of them ignore the traces and commands, and we comply with their original frameworks.

5.2 Q1. Comparative Experiments

For the four troubleshooting methods applied on different microservices, we use *top-k* (A@k), specially A@1 and A@3, to evaluate their accuracy. It refers to the probability that the groundtruth fault is included in the *top-k* results.

[2] Online-Boutique: https://github.com/GoogleCloudPlatform/microservices-demo.
[3] Sock-Shop: https://github.com/microservices-demo/microservices-demo.
[4] Train-Ticket: https://github.com/FudanSELab/train-ticket.
[5] Chaos-Mesh: https://chaos-mesh.org/.

Table 1. Overall accuracy comparison of microservices troubleshooting.

Fault type	Fault (#)	Micro-services	Instance (#)	MicroCBR		CloodCBR		CloudRCA		GraphRCA	
				A@1	A@3	A@1	A@3	A@1	A@3	A@1	A@3
Net	17	OB	187	***.91***	**1.0**	.47	.59	.72	.82	.81	**1.0**
		SS	238	***.94***	**1.0**	.39	.54	.61	.78	.80	**1.0**
		TT	1139	***.92***	**1.0**	.49	.61	.70	.81	.79	**1.0**
Pod	3	OB	33	***1.0***	**1.0**	.84	.84	.82	.82	.91	.97
		SS	42	***1.0***	**1.0**	.88	.88	.83	.83	.88	.93
		TT	201	***1.0***	**1.0**	.81	.81	.88	.88	.81	.93
Stress	2	OB	22	.95	**1.0**	***1.0***	**1.0**	.95	**1.0**	.73	.95
		SS	28	***.96***	**1.0**	.89	**1.0**	.89	**1.0**	.57	.93
		TT	134	***.97***	**1.0**	.90	**1.0**	.94	**1.0**	.66	.96
JVM	4	OB	4	***1.0***	**1.0**	.91	**1.0**	.84	**1.0**	.67	**1.0**
		SS	16	***.93***	**1.0**	.69	**1.0**	.57	**1.0**	.74	.98
		TT	148	***.94***	**1.0**	.78	**1.0**	.80	**1.0**	.64	.94
I/O	5	OB	55	***.96***	**1.0**	.62	.75	.87	**1.0**	.84	**1.0**
		SS	70	***.98***	**1.0**	.66	.84	.84	**1.0**	.87	**1.0**
		TT	335	***.90***	**1.0**	.70	.80	.87	**1.0**	.83	**1.0**
Config	10	OB	110	***.84***	**1.0**	.00	.00	.00	.00	.00	.00
		SS	140	***.81***	**1.0**	.00	.00	.00	.00	.00	.00
		TT	670	***.82***	**1.0**	.00	.00	.00	.00	.00	.00
All (of 3572 instances)				***.91***	**1.0**	.45	.54	.57	.66	.59	.74
Improvement of our method				(ours)		102%	85%	60%	51%	54%	35%
Var. across 3 microservices				***1.3e−4***	**0.0**	7.6e−4	4.7e−5	1.1e−3	4.9e−6	3.2e−4	8.8e−5

Table 1 details the experiment results. The A@1 of our method outperforms the compared SOTA methods by 54% to 102%, and A@3 prove that our method can always provide the correct answer within three options. Furthermore, the pod faults catch our attention, which contain plenty of temporal anomaly events collected by the knowledge base. The design of our method that can handle temporal characteristics ensures our accuracy advantages. Besides, those configuration faults heavily depend on the post-hoc commands. Compared to SOTA methods that leak commands information, our method benefits from adequate heterogeneous data and achieves higher accuracy. We also study the accuracy variance of these methods across different microservices, while our method has the lowest variance. This indicates that MicroCBR is more robust than others and can be used widely on various microservices.

Next, with other SOTA methods ignoring the fault type analysis, we report A@1 of MicroCBR hierarchical reasoning in Table 2 on its own. Different from the first experiment, we study the fault type accuracy of novel faults that have never been seen before. By analyzing the results, we find out that although novel faults are not included in the knowledge base, whether their fingerprints are similar to known faults matters. For the novel stress and JVM faults, parts of their fingerprints overlap with known same type faults, resulting in high accuracy. In

Table 2. A@1 of MicroCBR hierarchical reasoning for novel faults.

	Net	Pod	Stress	JVM	I/O	Config	All (of 6 types)
OB	0.81	0.76	0.95	0.93	0.83	0.79	0.82
SS	0.83	0.74	0.93	0.95	0.76	0.72	0.80
TT	0.85	0.75	0.97	0.93	0.80	0.78	0.84
All (of 3 systems)	0.96	0.75	1.00	0.94	0.80	0.77	0.83

a comprehensive view across 3 microservices with 6 fault types, the hierarchical reasoning of MicroCBR can diagnose the fault type with 83% accuracy.

5.3 Q2. Ablation Experiment

For ablation experiments, we first study the A@1 with different ablation rates and further talk about the effect of spatio-temporal knowledge graph on troubleshooting accuracy.

Anomaly detectors play the entry role of MicroCBR, by drawing the fingerprints of emerging faults. However, due to various limitations, such as parameter selections or data bias, they fail to give a perfect fingerprint. We study their impact by customizing the ablation rates of fingerprints, e.g. 0.2 ablation rate means that only 80% of the complete fingerprint can be collected. One additional setting is that the size of each fault fingerprint is no less than one. Figure 5 reports the A@1 of MicroCBR with different ablation rates. With the increase of ablation rate, the accuracy of stress and JVM faults decrease significantly. While the pod faults benefit from temporal anomaly events orders, our framework is still able to locate root causes correctly. Besides, limited to the size of fingerprints, those configuration faults that have small fingerprints change slightly on their A@1. However, it is particularly noteworthy that the accuracy of I/O faults increases when the ablation rate changes from 0.2 to 0.4. This is because this change affects fingerprint importance weight, as expressed in Eq. 3. This experiment illustrates that the anomaly detectors do affect the accuracy of

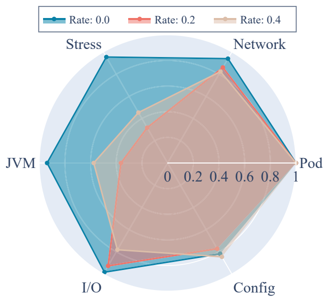


Fig. 5. A@1 of MicroCBR with different fingerprint ablation rates.

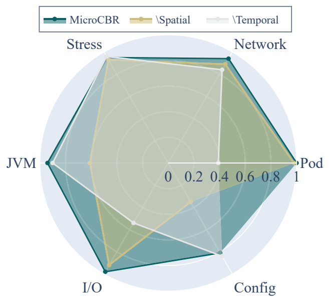


Fig. 6. A@1 of MicroCBR with spatial and temporal ablations.

MicroCBR. This conclusion guides us to select anomaly detectors with care in practical applications, ensuring high-quality fault fingerprints.

Next is the ablation study on spatio-temporal knowledge graph. We separately eliminate the spatial and temporal characteristics from our framework, represented by \ Spatial and \ Temporal. Figure 6 shows that these two characteristics in varying degrees impact A@1. The JVM and configuration faults need to compare similar instances to detect anomalies, while pod and I/O faults contain anomaly events sequences. The results show that the design of spatio-temporal knowledge graph improves the troubleshooting accuracy of MicroCBR.

5.4 Q3. Efficiency Experiments

Retain new cases of CBR make the incremental update possible, accompanied by the accumulation of knowledge base. To ensure the long-term usability of our method, we study how the scale of knowledge base and the size of fault fingerprints affect the troubleshooting efficiency.

Fingerprinting the emerging fault is the first step after a system alert. In our framework, it requires interaction with the microservice systems, which includes requesting data from databases and executing commands post-hoc. From Fig. 7 we can see that the troubleshooting mean time of MicroCBR far outstrips others because collecting commands results takes plenty of time. Fortunately, the

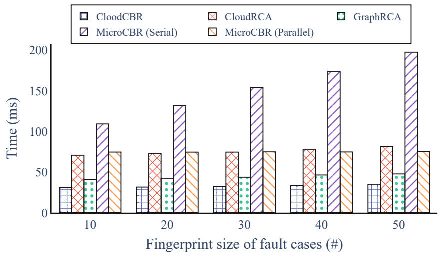


Fig. 7. Troubleshooting mean time with different fingerprint sizes (knowledge base size = 3000).

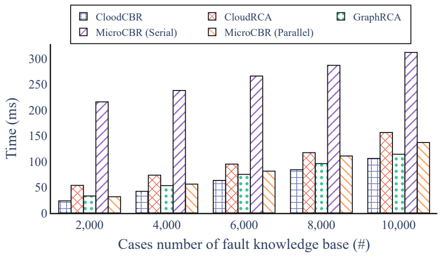
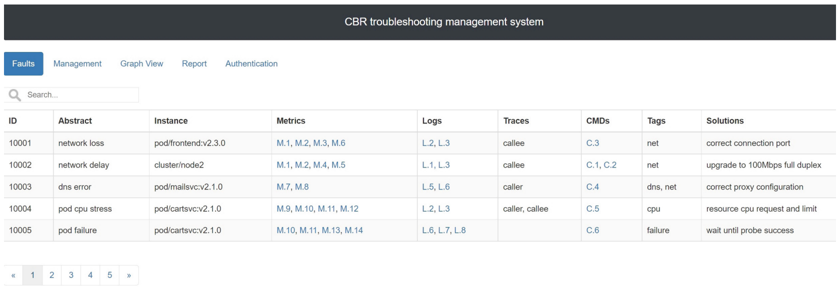


Fig. 8. Troubleshooting mean time with different knowledge base sizes (fingerprint size = 30).



ID	Abstract	Instance	Metrics	Logs	Traces	CMDs	Tags	Solutions
10001	network loss	pod/frontend:v2.3.0	M.1, M.2, M.3, M.6	L.2, L.3	callee	C.3	net	correct connection port
10002	network delay	cluster/node2	M.1, M.2, M.4, M.5	L.1, L.3	callee	C.1, C.2	net	upgrade to 100Mbps full duplex
10003	dns error	pod/mailsvc:v2.1.0	M.7, M.8	L.5, L.6	caller	C.4	dns, net	correct proxy configuration
10004	pod cpu stress	pod/cartsvc:v2.1.0	M.9, M.10, M.11, M.12	L.2, L.3	caller, callee	C.5	cpu	resource cpu request and limit
10005	pod failure	pod/cartsvc:v2.1.0	M.10, M.11, M.13, M.14	L.6, L.7, L.8		C.6	failure	wait until probe success

Fig. 9. Prototype of MicroCBR management system.

overhead of time can be optimized through parallelism. Figure 8 shows us the troubleshooting mean time with different knowledge base sizes. The mean troubleshooting time of our MicroCBR and other compared methods are increase linearly with the knowledge base accumulation, which is consistent with the theoretical analysis from Eq. 4. In particular, linear time complexity also ensures the usability of our algorithm in practical scenarios. Although the cost of our framework in time comparison is not the best, we argue that its performance of once troubleshooting at a second level is still acceptable.

5.5 Q4. Case Studies and Learned Lessons

Our framework has been widely tested in a real private cloud platform, which has more than 100,000 users from the field of education, health care and finance. During the long-term maintenance of this platform, we collect 710 known faults as a knowledge base. To evaluate the usability of our method in practical applications, we apply MicroCBR on this platform. Figure 9 shows a prototype that provides an easy access for operators to use MicroCBR. We collect the user experience feedback and select two typical cases for study.

The first case is a misconfiguration fault. We successfully detect a disk mount error after once container upgrade. It is noteworthy that most of the metrics fail to be sampled during the upgrade, thus anomaly detection cannot be performed from the time dimension when a new pod start. Thanks to our framework having designed a strategy of referencing similar instances, it finds out a misconfiguration by comparing pods that have not been upgraded. The whole diagnosis is completed in 60 s, which shortens the time by 90% compared with experiential 10-minutes manual troubleshooting. This case shows the power of our framework in reducing MTTR and improving the quality of service.

Another case is related to the mining virus. The system monitors that the CPU utilization of a *cnrig* process on a cluster is as high as 3181% and continues to be abnormal. Since no mining-related cases have been included in the knowledge base before. MicroCBR reports it as a stress fault at a high level. Operators revise this report by adding more mining virus names and traffic features of mining pools, with adding it to the knowledge base. We successfully detect another mining virus called *syst3md* after one week. This case proves the ability of our framework to deal with novel faults.

An important lesson learned from real deployment is that anomalies with high weighted scores always indicate root causes. Meanwhile, these anomalies provide sufficient discrimination for fault fingerprints, which is the basis for similarity-based retrieval.

6 Conclusion

This paper presents our framework named MicroCBR for microservices troubleshooting. Heterogeneous data from metrics, logs, traces, and commands are integrated into a spatio-temporal fault knowledge graph. Hierarchical case-based reasoning is performed to recommend historical similar cases and solutions to operators. The extensive experiments show that MicroCBR outperforms the SOTA methods on troubleshooting accuracy. We also share case studies and learned lessons from our deployment in a real private cloud.

Acknowledgement. This work was supported in part by Austrian-Chinese Cooperative RTD Projects: 171111KYSB20200001, and the National Natural Science Foundation of China No. U20A20180 and No. 61802366.

References

1. Liu, P., Xu, H., Ouyang, Q., et al.: Unsupervised detection of microservice trace anomalies through service-level deep Bayesian networks. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering, pp. 48–58. IEEE (2020)
2. Xu, H., Chen, W., Zhao, N., et al.: Unsupervised anomaly detection via variational auto-encoder for seasonal KPIs in web applications. In: Proceedings of the 2018 World Wide Web Conference, pp. 187–196 (2018)
3. Gan, Y., Zhang, Y., Hu, K., et al.: Seer: leveraging big data to navigate the complexity of performance debugging in cloud microservices. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 19–33 (2019)
4. Wu, L., Bogatinovski, J., Nedelkoski, S., Tordsson, J., Kao, O.: Performance diagnosis in cloud microservices using deep learning. In: Hacid, H., et al. (eds.) ICSOC 2020. LNCS, vol. 12632, pp. 85–96. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-76352-7_13
5. Zhao, N., Wang, H., Li, Z., et al.: An empirical investigation of practical log anomaly detection for online service systems. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1404–1415 (2021)
6. Zhou, P., Wang, Y., Li, Z., et al.: Logchain: cloud workflow reconstruction & troubleshooting with unstructured logs. Comput. Netw. **175**, 107279 (2020)
7. Luo, C., Lou, J.-G., Lin, Q., et al.: Correlating events with time series for incident diagnosis. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1583–1592 (2014)
8. Li, Z., Chen, J., Jiao, R., et al.: Practical root cause localization for microservice systems via trace analysis. In: 2021 IEEE/ACM 29th International Symposium on Quality of Service, pp. 1–10. IEEE (2021)
9. Zhang, Y., Guan, Z., Qian, H., et al.: CloudRCA: a root cause analysis framework for cloud computing platforms. In: Proceedings of the 30th ACM International Conference on Information & Knowledge Management, pp. 4373–4382 (2021)
10. Brandón, Á., Solé, M., Huélamo, A., et al.: Graph-based root cause analysis for service-oriented and microservice architectures. J. Syst. Softw. **159**, 110432 (2020)
11. Wang, H., Wu, Z., Jiang, H., et al.: Groot: an event-graph-based approach for root cause analysis in industrial settings. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering, pp. 419–429. IEEE (2021)
12. Chen, P., Qi, Y., Zheng, P., Hou, D.: CauseInfer: automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. In: IEEE INFOCOM Conference on Computer Communications. IEEE (2014)

13. Qiu, J., Du, Q., Yin, K., et al.: A causality mining and knowledge graph based method of root cause diagnosis for performance anomaly in cloud applications. Appl. Sci. **10**(6), 2166 (2020)
14. Zhang, C., Zhou, Z., Zhang, Y., et al.: Netrca: an effective network fault cause localization algorithm. arXiv preprint arXiv:2202.11269 (2022)
15. Nkisi-Orji, I., Wiratunga, N., Palihawadana, C., Recio-García, J.A., Corsar, D.: Clood CBR: towards microservices oriented case-based reasoning. In: Watson, I., Weber, R. (eds.) ICCBR 2020. LNCS (LNAI), vol. 12311, pp. 129–143. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58342-2_9
16. Bennacer, L., Amirat, Y., Chibani, A., et al.: Self-diagnosis technique for virtual private networks combining Bayesian networks and case-based reasoning. IEEE Trans. Autom. Sci. Eng. **12**(1), 354–366 (2014)
17. Ma, M., Yin, Z., Zhang, S., et al.: Diagnosing root causes of intermittent slow queries in cloud databases. Proc. VLDB Endow. **13**(8), 1176–1189 (2020)
18. Ester, M., Kriegel, H.-P., Sander, J., et al.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: KDD, vol. 96, pp. 226–231 (1996)
19. Ren, H., Xu, B., Wang, Y., et al.: Time-series anomaly detection service at Microsoft. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 3009–3017 (2019)
20. Blázquez-García, A., Conde, A., Mori, U., Lozano, J.A.: A review on outlier/anomaly detection in time series data. ACM Comput. Surv. **54**, 1–33 (2021)
21. Du, M., Li, F., Zheng, G., Srikumar, V.: DeepLog: anomaly detection and diagnosis from system logs through deep learning. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (2017)
22. He, S., Zhu, J., He, P., Lyu, M.R.: Experience report: system log analysis for anomaly detection. In: 2016 IEEE 27th International Symposium on Software Reliability Engineering, pp. 207–218. IEEE (2016)
23. Amir, A., Gotthilf, Z., Shalom, B.R.: Weighted LCS. J. Discret. Algorithms **8**(3), 273–281 (2010)
24. Balouek, D., et al.: Adding virtualization capabilities to the grid'5000 testbed. In: Ivanov, I.I., van Sinderen, M., Leymann, F., Shan, T. (eds.) CLOSER 2012. CCIS, vol. 367, pp. 3–20. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-04519-1_1