Received: 16 July 2021 | Revised: 4 November 2021 | Accepted: 18 November 2021

DOI: 10.1002/smr.2413

SPECIAL ISSUE - TECHNOLOGY PAPER

Journal of Software: Evolution and Process WILEY

TraceRank: Abnormal service localization with dis-aggregated end-to-end tracing data in cloud native systems

Guangba Yu | Zicheng Huang | Pengfei Chen

School of Computer and Engineering, Sun Yat-sen University, Guangzhou, China

Correspondence
Pengfei Chen, School of Computer and Engineering, Sun Yat-sen University, Guangzhou, China.
Email: chenpf7@mail.sysu.edu.cn

Funding information
Key-Area Research and Development Program of Guangdong Province, Grant/Award Number: 2020B010165002; National Natural Science Foundation of China, Grant/Award Numbers: 61802448, U1811462; Natural Science Foundation of Guangdong Province, Grant/Award Number: 2019A1515012229; Wechat Rhino-Bird Joint Research Program, Grant/Award Number: JR-WXG-2021621; Basic and Applied Basic Research of Guangzhou, Grant/Award Number: 202002030328

Abstract

Modern cloud native applications are generally built with a microservice architecture. To tackle various performance problems among a large number of services and machines, an end-to-end tracing tool is always equipped in these systems to track the execution path of every single request. However, it is nontrivial to conduct root cause analysis of anomalies with such a large volume of tracing data. This paper proposes a novel system named *TraceRank* to identify and locate abnormal services causing performance problems with dis-aggregated end-to-end traces. *TraceRank* mainly includes an anomaly detection module and a root cause analysis module. The root cause analysis procedure is triggered when an anomaly is detected. To fully leverage the information provided by the tracing data, both the spectrum analysis and the PageRank-based random walk methods are introduced to pinpoint abnormal services. The experiments in TrainTicket and Bookinfo microservice benchmarks and a real-world system show that *TraceRank* can locate root causes with 90% in Precision and 86% in Recall. *TraceRank* has up to 10% improvement compared with several state-of-the-art approaches in both Precision and Recall. Finally, *TraceRank* has good scalability and a low overhead to adapt to large-scale microservice systems.

KEYWORDS
end-to-end tracing, microservice, random walk, root cause analysis, spectrum

1 | INTRODUCTION

Recently, microservice has become a popular architecture to construct cloud native systems. In a microservice system, applications are decomposed into small, autonomous, single responsible, and loosely coupled services communicating via network messages. The microservice architecture has advantages of a diversity of technologies, such as resilience, scalability, reliability, reusability, and agility.[1–3] On the other hand, lots of unexpected problems may occur with the increase of service scale and complexity of service dependencies, which may have a strong impact on user experience. According to one report,[4] the WeChat system of Tencent contains 3000 microservices distributed on 20,000 virtual servers. It is extraordinarily difficult to find a tiny root cause from such a large-scale system. Moreover, by the reason of agile development and the usage of DevOps tools, code submission and version updates can be up to dozens of times within a single day,[5–7] which highly increases the probability of potential faults. Hence, it is a difficult task for operators to identify the root cause.

Numerous studies have been done on root cause analysis (RCA). FChain[8] pinpoints faults based on the changed propagation patterns of anomalies. CauseInfer[9] and Sieve[10] build causality graphs of system components and then conduct RCA. Besides, lots of efforts take advantage of log data and metrics to locate root causes of system problems.[11–14] However, RCA is still a difficult problem due to the complex service dependencies and the dynamic run-time environment of microservice systems.

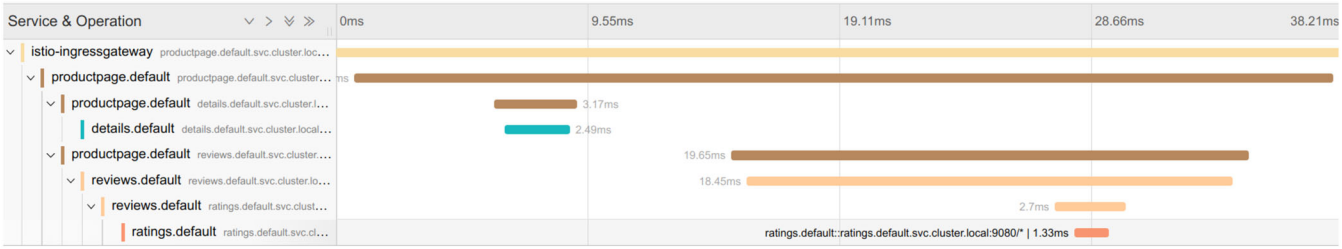

FIGURE 1 A trace of Bookinfo from Jaeger. Each of the time range is known as a span, which represent the starting/ending time of services being performed. The latency of each service can be obtained by *ending time − staring time*. The process time represents the amount of time where a span is definitely not waiting on a child span to finish

Recently, much work has been done on resolving the fault localization problem in microservice systems by using distributed tracing. Distributed tracing (e.g., Opentracing* and Jaeger†) is being integrated into modern microservice systems as a fundamental tool for understanding and solving program bugs.[15] A trace captures the execution path and process time about how a single request traverses the system.[16] Figure 1 illustrates a trace of a request traversing BookInfo microservice system.‡ In this paper, we define dis-aggregated tracing data as the separate trace providing an in-depth view into the execution of a single request. We define aggregated tracing data as the analysis data (e.g., success rate and average latency per minute of a service) collected and aggregated from a set of traces.

Current tracing systems are primarily designed to present detailed information about a single trace like Figure 1. Engineers need to manually look through a large number of traces to localize the root causes. To accelerate the process of root cause localization in microservice systems, MicroRCA,[7] Automap,[17] MS-rank,[18] MEPFL,[19] and Microscope[20] have proposed interesting graph, statistic, or deep learning based approaches. However, most of the above approaches conduct RCA with the aggregated end-to-end tracing data. They usually extract service dependency graph from traces and then localize the root causes of microservice systems based on the overall change of some metrics, which extracted from the traces (e.g., average latency per minute). The above methods are efficient and scalable because they only use coarse-grained telemetric information from the aggregated tracing data. But they are ineffective in localizing partial and intermittent faults. In such scenarios, only part of the traffic is affected,[21] and the anomalies are concealed in the aggregated data. For example, the average latency of a service may be normal even when some error requests have covered it.

If we mine the fine-grained information from each trace in a dis-aggregated manner, we can discover the anomalous behavior of the partial affected requests distinctly. The challenges on leveraging traces in a dis-aggregated manner to conduct RCA include the following: (i) the complex service topology makes the process of fault propagation unpredictable, (ii) the root cause is difficult to localize due to fault propagation, and (iii) the traces are generated in large volumes. Moreover, analyzing traces manually is of low efficiency and depends heavily on domain knowledge.

The most relevant work to ours is TraceAnomaly.[22] TraceAnomaly proposed an unsupervised deep learning algorithm which can learn the complex trace patterns in a service and accurately detect trace anomalies. However, TraceAnomaly is not scalable in the large complex microservice systems composed of tens of thousands of services with several terabytes of tracing data generated per day.[4] The high-volume and high-dimensional input data make the deep learning approach extremely time-consuming. Moreover, the rapid updates of microservice systems incur more retraining cost and weaken the applicability of TraceAnomaly. To overcome the aforementioned challenges and drawbacks of existing work, this paper introduces *TraceRank*, a novel approach that localizes root causes with dis-aggregated traces. *TraceRank* combines a random walk algorithm with a spectrum-based method to analyze traces. The main idea of *TraceRank* is to leverage the clues provided by a collection of normal and anomalous traces and the service call graph to locate the root causes. To achieve that, *TraceRank* queries traces from the trace database (e.g., Elasticsearch§) and detects anomalies by monitoring the latency of services in real time. An analyzer module with a personalized PageRank-based random walk algorithm[23] gives an initial ranked root cause list. Then, a spectrum-based method[24] is used to calibrate the initial ranked list in order to locate root causes precisely. The experiments in TrainTicket and Bookinfo microservice benchmarks and a real-world application show that *TraceRank* can locate root causes with 90% in Precision and 86% in Recall. *TraceRank* has up to 10% improvement compared with several state-of-the-art approaches in both Precision and Recall. Finally, *TraceRank* has good scalability and a low overhead to adapt to large-scale microservice systems.

The contributions of this paper are threefold:

- We mine the abundant insightful information that is buried in the dis-aggregated tracing data and leverage the clues provided by normal and abnormal traces to conduct RCA.
- We propose a novel RCA approach in microservice environments by combining the PageRank-based random walk and spectrum analysis. This approach can always locate the root cause service at the first rank.

- We design and implement a prototype, namely, *TraceRank*, to infer the root causes in microservice systems with high precision and recall as well as good scalability and low overhead.

The rest of this paper is organized as follows. In Section 2, we present the background on related techniques. Section 3 elaborates the details of *TraceRank*. In Section 4, we evaluate *TraceRank* with two widely used microservice benchmarks and a real-world application. We discuss the threats to validity of *TraceRank* facing more complicated workloads in Section 5. Section 6 presents the related work, and Section 7 concludes this paper.

2 | BACKGROUND

In this section, we discuss the relevant technologies ranging from end-to-end tracing, spectrum-based fault localization (SBFL), and PageRank algorithm, which comprise our core algorithm of RCA in microservice environments.

2.1 | End-to-end tracing

Due to the complex dependencies of microservice systems, end-to-end tracing is always enabled to help people understand and diagnose problems. Dapper[25] proposed by Google is a seminal work on end-to-end tracing. It defines a tracepoint as a span and tag timestamps for four crucial activities on each span, namely, *server send, client receive, client send, and server receive* (*SS*,*CR*,*CS*,and*SR*), which is shown in Figure 2. A unique ID is assigned to each request and span, respectively. Moreover, the newly spawned span has a parent span assigned a parent ID. With these IDs, we can correlate the spans into an end-to-end tracing path. An end-to-end tracing system comprises an instrumented client that collects and sends spans, a span collector gathering span data, a back-end storage as the persistent data storage, and APIs and UI dashboard for users to query traces. Moreover, a number of implementations of end-to-end tracing including X-Trace,[26] Dapper,[25] and Stardust[27] have been proved that they are low overhead and can be used continuously in production systems.

Figure 2 shows the tracing process of a request from a client to a server. In this figure, when a client service instance sends a request to the server service instance (i.e., cs), we will get a new record in the trace with a new endpoint whose value is "cs" and the timestamp is T_{cs}. When the server service instance receives this request (i.e., sr), the request is processed at the server end. After that, the server service instance will send the response back to the client service instance (i.e., ss), then the span will get two new endpoints whose value are "sr" and "ss" with timestamps T_{sr} and T_{ss}, respectively. When the client service instance receives the response from the server service instance (i.e., cr), the span will get another endpoint whose value is "cr" with a timestamp T_{cr}. Therefore, we can get the latency of each service by calculating $T_{cs} - T_{cr}$. Besides, the processing time of a service can be measured by subtracting the latency of its direct callee from its latency.

Although end-to-end tracing can provide essential information to find anomalies in microservice systems, it is still extraordinarily difficult to precisely locate root cause services. The primary reason is that the complex-dependent relationships among microservice systems all bury the real

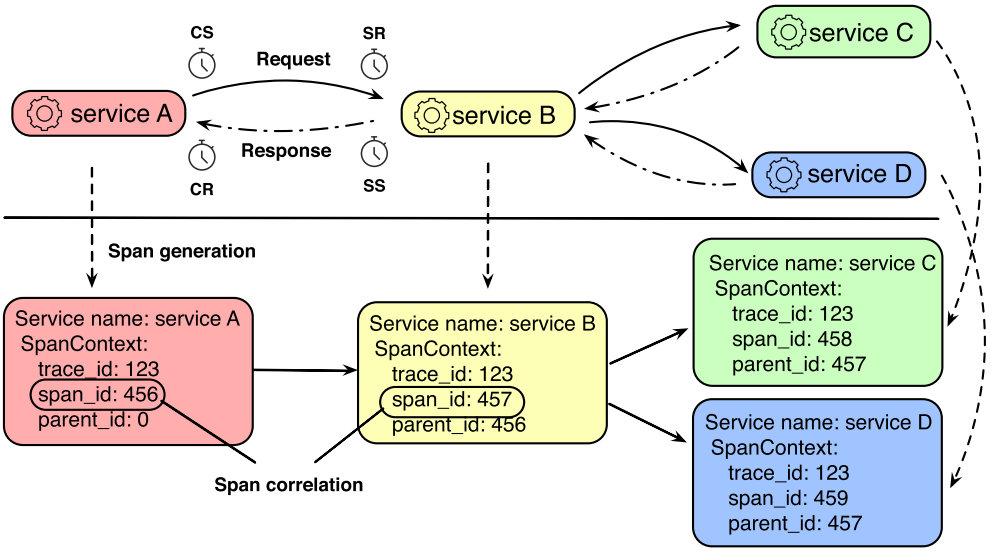

FIGURE 2 An example of end-to-end tracing between services

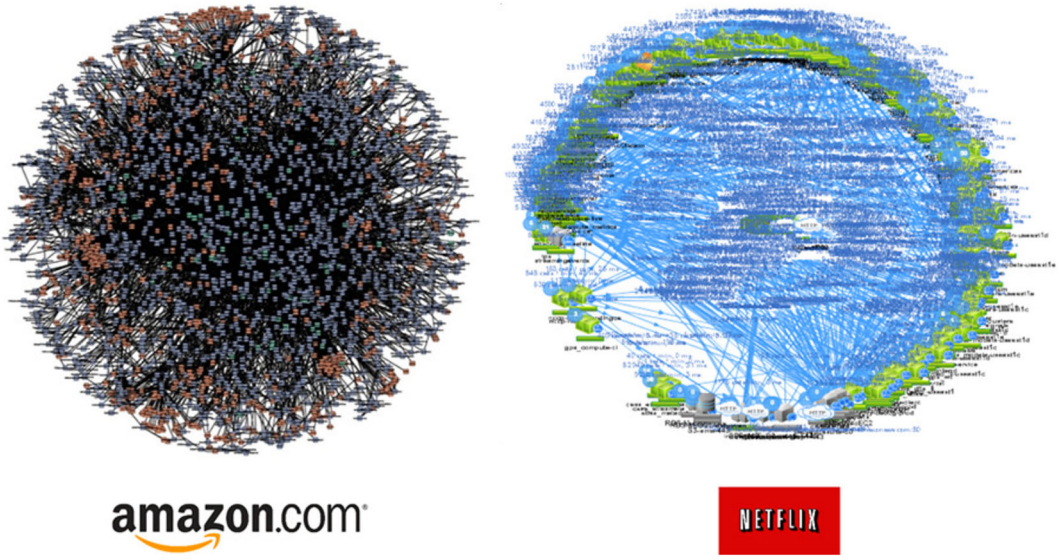

FIGURE 3 Service dependencies of Amazon and Netflix[28]

root causes. Figure 3 shows the complex service dependencies of Amazon and Netflix, respectively. There is no doubt that it is difficult to find the root causes of performance problems in such dependencies.

2.2 | Spectrum-based fault localization

SBFL is a widely used technique in the software testing domain.[24,29–32] When given a program and some failing and passing test cases, SBFL collects test coverage information for each program element e first. The SBFL technique mainly takes four critical statistics: e_f, e_p, n_f, n_p, where e_f and e_p are the total numbers of failing and passing tests covering the program entity e, while n_f and n_p are the numbers of failing and passing tests that do not cover the program entity e, respectively. Then, it calculates a list of suspicious scores for all program entity based on the coverage information using various SBFL formulae (e.g., Ochiai).[29] The suspicious score indicates how likely a program entity is to be faulty.

2.3 | PageRank algorithm

PageRank[23] is a famous algorithm to analyze web links, which originally targets improving the quality and speed of web search. The intuition behind PageRank is to rank the importance of each website based on its internal links. The transition matrix describes the links among websites, and the additional teleportation vector is designed in case that some websites have no outbound links. The PageRank algorithm can be described as follows:

$$\vec{x} = d \cdot P\vec{x} + (1-d) \cdot \vec{v} \tag{1}$$

where $\vec{x}$ presents the ranking score, P is the transition matrix weighted by the damping parameter d, and $\vec{v}$ denotes the additional teleportation vector. Beyond ranking websites, PageRank is adopted in RCA of IT systems. For example, Kim et al[33] proposed a PageRank-based approach to find root causes of anomalies in service-oriented architectures due to the similarity between the service dependencies and website links.

2.4 | Motivation

In this section, we present a motivating example. Assume there is a simple microservice system whose service dependency graph is shown in Figure 4, with the average request latency of each service. The latency of each service visited by different requests is also presented in Table 1.

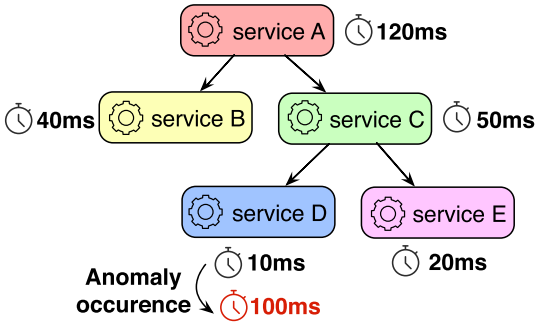

FIGURE 4 Service dependency graph of microservice system in the example of motivation

TABLE 1 Visited services and latency of each trace

		Service Latency				
Trace	**Anomaly**	***A* (ms)**	***B* (ms)**	***C* (ms)**	***D* (ms)**	***E* (ms)**
1	False	114	38	–	–	–
2	False	121	–	55	–	23
3	True	240	50	150	100	–
4	True	270	45	170	90	40
5	True	230	–	200	120	50

Service *D* is the root cause in this example. A bug in service *D* caused delays to increase by 90 ms. In this example, due to the fault propagation, the anomaly of service *D* propagates to service *C* and service *A*. Considering the interference due to co-location, service *E* is also influenced with a probability, whereas service *B* is normal.

According to the aggregated end-to-end tracing data, the average latency of services *A*, *C*, *D*, and *E* all increased. The current RCA approaches based on aggregated data may be misled by these spikes. They draw a wrong conclusion that service *E* is the culprit. However, from the perspective of dis-aggregated traces, we can find that the latency of services *A*, *C*, and *E* increase only when the requests pass through service *D*, but not otherwise. Therefore, we propose to use dis-aggregated traces to localize root causes in microservice systems.

The spectrum-based approach uses test coverage as input to localize program faults. The idea of the spectrum-based approach can be transformed and applied in microservice systems with a concept mapping. The concept of "program entity" in the software testing domain could be replaced by "service," and the concept of "test case" could be replaced by "trace". Thus, we can conclude some inferences based on the trace coverage as follows:

1. {*trace 1*} → {*services A and B are normal*}
2. {*trace 2*} Δ {*services A, C, and E are normal*}
3. {*trace 1, trace 2, trace 4*} Δ {*service D is abnormal*}
4. {*trace 1, trace 3, trace 4*} Δ {*services C and D may be abnormal*}

where "Δ" means "infer". The services involved in the anomalous traces are the candidates of root causes. With the normal traces, we filter out candidates involved and narrow down the scope of investigation. As shown in inference 3, by analyzing the coverage of traces 1, 2, and 4, the spectrum-based approach can easily find the root cause service *D*. This inference procedure benefits from the clues provided by the normal and abnormal traces.

However, if we replace trace 2 with trace 3 like inference 4, we find that services *C* and *D* get the same suspicious score in the spectrum-based approach. This is because services *C* and *D* have the same coverage information in inference 4. In such a situation, we need other approaches like PageRank to calibrate the results obtained by the spectrum-based approaches. If a suspicious service is confirmed by these two methods, we regard it as the root cause. Therefore, we propose a novel approach by combining the SBFL and randown walk algorithm, which will be stated in detail in the following sections.

3 | SYSTEM DESIGN

3.1 | System overview

In this section, we show an overview of *TraceRank*. As demonstrated in Figure 5, *TraceRank* requires end-to-end tracing data as input. *TraceRank* continuously queries traces within a sliding time window (set by anomaly detection module) from the trace database (e.g., Elasticsearch) and feeds them to the anomaly detection module. The anomaly detection module triggers the root cause localization module as soon as it detects an anomaly. *TraceRank* extracts the dependency graph, latency, and process time of each service from the traces. The root cause localization module first calculates a suspicious score of spectrum analysis for each service instance. Then, another suspicious score is obtained for each service instance using the personalized PageRank based on the dependency graph and correlation scores. Finally, *TraceRank* combines these two scores to generate a ranked list as the result. Please note that the granularity of *TraceRank* in locating root causes is service instance.

3.2 | Anomaly detection and data preparation

To detect system anomalies, *TraceRank* continuously monitors the latency of each trace within a sliding time window. The time window is $\beta * \Delta t$ ($\beta = 3$ in this paper) in length and moves Δt forward each time, where Δt denotes the time unit ($\Delta t = 1$ min in this paper). Some requests may not be completed within a time window, resulting in incomplete traces. Therefore, we set $\beta = 3$ to ensure that requests that were not completed in the previous window can continue to be analyzed in the next window. We use Δt to control the volume of traces to be analyzed at one time. The Δt can be adjusted according to the workload of microservice systems.

SLOs provide a quantitative approach to define the level of service that customers expect. The SLO may be composed of one or more service-level indicators (SLI) to produce the SLO achievement value. Engineers need to try their best to keep the SLIs under the SLOs to provide an acceptable quality of service. Thus, once a SLO violation is detected, the root cause inference should be triggered. Because most microservice faults manifest themselves in terms of latency increase or request time-out, latency is the most widely used SLI to profile the running applications.[34–36] Hence, we use the request latency as an indication of anomaly. As shown in Figure 1, we can extract the latency of each service from traces.

Figure 6 shows that different types of traces have different execution paths and their latency varies significantly. The latency variation of the same kind of traces is a potential indicator of the anomaly. The traces that belong to the same type of trace are similar in structure. It is intuitive to think of dividing traces with the same structure into a group, and then doing anomaly detection with *K*-means within the group. However, a large-scale microservice system may have thousands of types of traces due to the explosion of services. It is ineffective and expensive to maintain a *K*-means model for each type of trace.

Fortunately, we found that the different types of traces that have similar structure may have a similar normal latency. For example, types 15 and 16 in Figure 6 have similar latency. This is because most services covered by them are similar. Therefore, it is possible to divide the traces with structural similarity into the same groups to reduce the number of *K*-means models. We use the hierarchical clustering method[37] to get

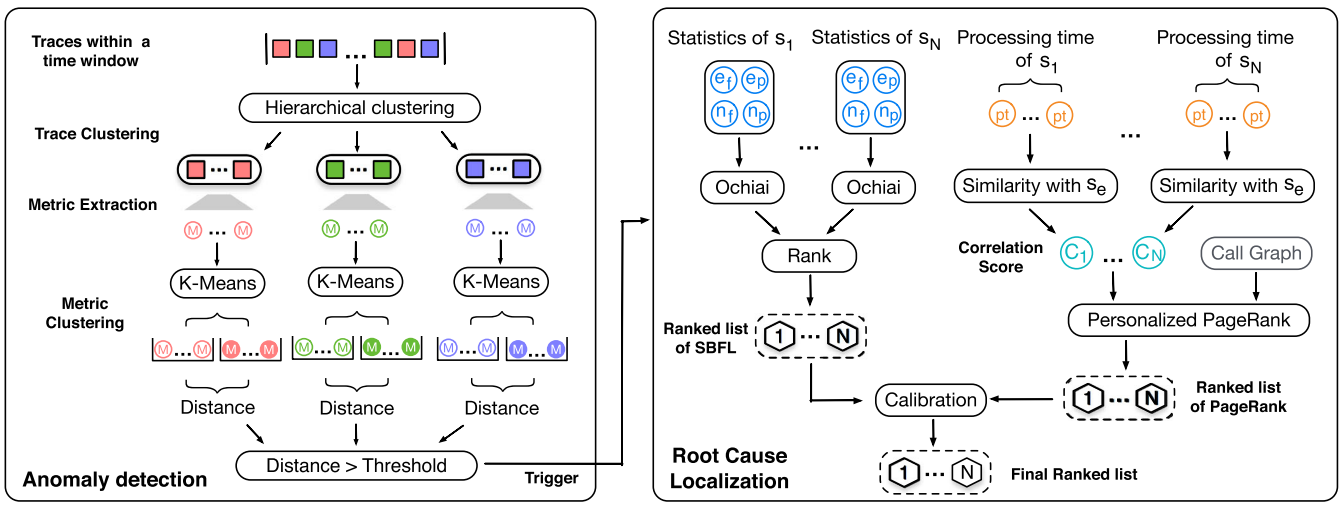

FIGURE 5 The workflow and main modules of *TraceRank*

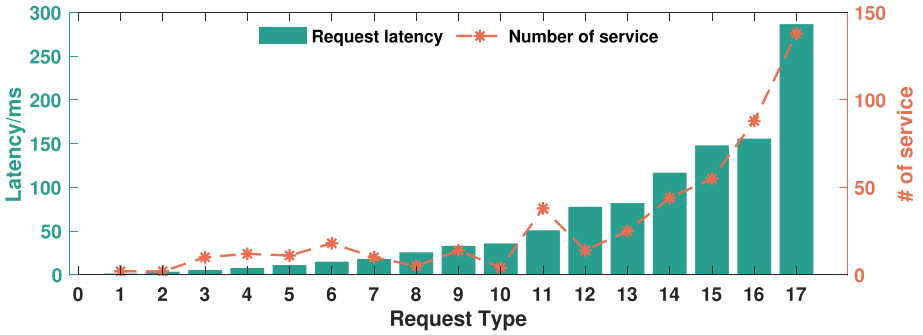

FIGURE 6 The number of visited services and the latency of different kind of traces

different groups of traces. To measure the structural similarity between traces, each trace is encoded into a vector v that embedding its structural information. The value v_i in index i is the number of times that service i is accessed in the trace. For example, the trace 1 in Table 1 can be encoded as [1, 1, 0, 0, 0]. A clustering tree is built by inserting each vector next to its nearest neighbor in the tree. The distance between two vectors is measured by the Euclidean distance. The number of groups mostly depends on a distance threshold t, which is the maximum intercluster distance allowed. The value of t ($t = 2$ is this paper) is a trade-off between the efficiency and accuracy of the anomaly detection module. If the threshold t is set less than 1, the anomaly detection module needs to maintain a K-means model for each type of trace. If the threshold t is set too large, it will allow for all traces to be merged together.

We apply the K-means based anomaly detection method to each group. In *TraceRank*, we adopt an unsupervised K-means clustering method to conduct anomaly detection. We set k as 2 and determine the distance between two centroids of corresponding clusters to detect anomalies. The idea behind our approach is that, when an anomaly happens in the time window, the pattern of abnormal latency is supposed to be different from normal latency. Once the distance between two centroids of these two clusters exceeds a certain level, an anomaly likely occurs. A parameter σ that usually ranges from 2 to 3 is adopted as the threshold. The pseudocode of anomaly detection module is shown in Algorithm 1. To avoid the influence of the outliers which are not caused by anomalies but by random fluctuation, we drop the smaller cluster and perform K-means clustering again if the number of elements in the smaller cluster is less than 1% of the number of elements in the group.

As for data preparation, the overall service dependency graph G, which is generated based on the invocation relationships recorded in traces, is prepared for the random walk algorithm. Table 2 shows the definition of the spectrum statistics. The statistics e_f, e_p, n_f, and n_p of each service can be extracted based on the coverage of normal and anomalous traces. Moreover, the mean and standard deviation of latency of each service extracted from the normal traces and mean latency of each service in anomalous traces are provided for Algorithm 2.

Algorithm 1 The anomaly detection algorithm

Require: List of traces within the sliding window, T;
Ensure: The anomaly alarm variable, *detect*;

```
detect ⇐ False
G ⇐ HClustering(T)                          ▷ Group traces via the hierarchical clustering method
for group in G do
    latency ⇐ ExtractLatency(group)         ▷ Obtain the latency from the traces of the group
    while True do
        cen, clusterS, clusterL ⇐ KMeans(latency, 2)   ▷ Clustering to two categories, obtain the centroids and the two clusters
        if clusterS.size() < group.size() * 1% then    ▷ Evaluate the size of the smaller cluster clusterS
            group.drop(clusterS)                       ▷ Outlier found, drop them from the group
        else
            break
        end if
    end while
    if |cen[0] − cen[1]| > σ * min(cen[0], cen[1]) then   ▷ Evaluate the distance between two centroids
        detect ⇐ True
        break
    end if
end for
return detect
```

TABLE 2 Definition of statistics for a specific service *svc*

Statistic	Definition
e_f	Number of anomalous traces that visit *svc*
e_p	Number of normal traces that visit *svc*
n_f	Number of anomalous traces that do not visit *svc*
n_p	Number of normal traces that do not visit *svc*

TABLE 3 Suspicious scores of reference 3 in Section 2.4

Container	e_f	e_p	n_f	n_p	Suspicious score
A	1	2	0	0	0.58
B	1	1	0	1	0.71
C	1	1	0	1	0.71
D	1	0	0	2	1
E	1	1	0	1	0.71

TABLE 4 Suspicious scores of reference 4 in Section 2.4

Container	e_f	e_p	n_f	n_p	Suspicious score
A	2	1	0	0	0.82
B	1	2	0	0	0.58
C	2	0	0	1	1
D	2	0	0	1	1
E	1	0	1	1	0.71

3.3 | Ranking with spectrum

We leverage the spectrum analysis to calculate the score of each service based on normal and anomalous traces. Intuitively, all the services in the normal traces are normal. But there is at least one anomalous service in anomalous traces. The four spectrum statistics e_f, e_p, n_f, and n_p have been prepared in the previous step. Our algorithm chooses Ochiai[38] SBFL technique because prior work have proved its effectiveness,[38] which is defined as:

$$\frac{e_f}{\sqrt{(e_f+e_p)*(e_f+n_f))}} \tag{2}$$

The spectrum analysis will provide a suspicious value for each service based on trace coverage. The service with the largest suspicious value will be identified as root cause. We use the example in Section 2.4 to show how spectrum analysis calculates suspicious scores. In the reference 3 in Section 2.4, as shown in Table 3, the spectrum analysis delivers the largest suspicious value for the service *D*. This example shows the effectiveness of spectrum analysis. However, when some services have the same coverage information like the reference 4 in Section 2.4, the spectrum analysis delivers the same suspicious score for them. As shown in Table 4, services *C* and *D* have the same suspicious score. The spectrum analysis cannot distinguish which service is the real root cause. Therefore, we introduce the PageRank algorithm to enhance spectrum analysis in the next section.

3.4 | Ranking with random walk

As shown in Section 3.3 the spectrum analysis cannot pinpoint the real culprits when multiple services with closely dependent relationships co-occur in anomalous requests because it does not consider the service dependency relationships. The spectrum analysis only considers successful and failed requests, whereas the latency of services is ignored. Therefore, we introduce the PageRank-based random walk algorithm to leverage service dependency topology and latency of services to further refine RCA.

Microservice systems typically have one or more front-end services that interact with users. The front-end services are the entrance to the user traffic and the beginning of the traces. Inspired by the Microscaler,[2] the latency of front-end services and abnormal services observe similar patterns in latency when an anomaly is. In *TraceRank*, we adopt the processing time rather than latency of each service to calculate the correlation score between each service and the front-end service. Compared with the access latency, the processing time can eliminate the propagated anomalies from dependent services, which can show the health status of service more precisely. To address the issue of multiple front-end services, *TraceRank* records all anomalous traces through them. Then, we construct service call graphs for these front-end services. *TraceRank* conducts random walk on these service call graphs. For each service s_i, the correlation score C_i, with respect to its front-end services s_f, is described as follows:

$$C_i = Sim(s_i, s_f) \tag{3}$$

$Sim(\cdot\,,\cdot)$ is the Pearson correlation coefficient (PCC)[39] between the processing time of s_i and s_f. This correlation score C_i signifies the relevance of the service s_i to the given s_f. PCC compares the trend of the processing time of s_i and s_f without considering their absolute values. *TraceRank* does not use the absolute values because the values of different services are various. For example, the processing time of one service is 100 ms, whereas another service may be 10 ms. It is not fair to consider the increment of the absolute value of the above two services. Moreover, we find that considering the trend of the processing time is enough to root cause localization.

Inspired by *MonitorRank*,[33] we adopt a random walk algorithm to find root causes. The basic idea of our approach is to do a random walk over the service call graph weighted by the correlation score. Intuitively, the longer staying on a certain service during the random walk, the more suspicious this service is. *TraceRank* adopts the personalized PageRank algorithm[40] with an additional teleportation probability (i.e., preference vector) to rank each service. If there are multiple front-end services, all of them will be analyzed simultaneously by our algorithm.

3.4.1 | Forward transition

The random walk algorithm starts from the front-end services and walks along the service call graph. Let the whole service call graph be $G = <V, E>$, where each node in V indicates a service instance and each edge e_{ij} is set to 1 when node n_i calls node n_j, excluding any self edge (e.g., $e_{ii} \notin E$).

The correlation score of each node C_i in the service call graph is another input to the algorithm. Let n_1 be the front-end, $\{n_2, \dots, n_k\}$ are the nodes visited by n_1. The correlation scores of all nodes in the trace of n_1 are denoted by $C = [C_1, \dots, C_k] \in (0,1]^k$. For the random walk, each node n_i is accessed according to its correlation score C_i, and the strength of each edge e_{ij} indicating node n_i calls node n_j is assigned to C_j. The transition probability matrix P of this part is defined as follows:

$$P_{ij} = \frac{A_{ij} C_j}{\sum_j A_{ij} C_j} \tag{4}$$

while $i, j = 1, 2, \dots, |V|$ and A is the adjacency matrix of the service call graph G.

3.4.2 | Backward transition

If there are only forward transitions along the service call graph, the random walker can not take other actions when the current service or its parent service shows a high correlation with the front-end while all other neighboring nodes do not. In other words, when the random walker falls into services that look less relevant to the given abnormal service, there is no way to escape until the next teleportation. Accordingly, we set up the *backward* transition to help the walker find more routes and make its random walk more heuristic. For each pair of nodes n_i and n_j, e_{ij} indicates that node n_i calls n_j, whereas the strength of e_{ij} is equal to C_j, the strength of e_{ji} is set as ρC_i to transit backward, where $\rho \in [0, 1)$. A higher value of ρ allows more flexibility to search some new paths.

3.4.3 | Selfward transition

In this section, we add a self edge to every service to avoid forcibly moving to another service. The random walker is encouraged to stay longer on the service if none of its in and out-neighbor services are of high correlation score. Specifically, for each node n_i, the strength of the corresponding self edge e_{ii} is equal to the correlation score C_i subtracted by the maximum correlation score of child nodes, namely, $max_{j:e_{ij} \in E} C_j$. The entry-point node is an exception as we do not add a selfward transition on it, namely, $e_{11} = 0$.

To combine both backward and selfward transitions, we define a new adjacency matrix A' with the service call graph G and the correlation score C as follows:

$$A'_{ij} = \begin{cases} \max(C_j) & \text{if } e_{ij} \in E \\ \max(\rho C_i) & \text{if } e_{ji} \in E, e_{ij} \notin E \\ \max(\max(0, C_i - \max_{k:e_{ik} \in E} C_k)) & \text{if } i = j, i > 1 \end{cases} \tag{5}$$

Notice that a node may be visited by different entry points, we take the max value in each execution path. With A', the new transition probability matrix P is defined as follows:

$$P_{ij} = \frac{A'_{ij}}{\sum_j A'_{ij}} \quad i,j = 1,2,...,|V| \tag{6}$$

As for the teleportation vector $\vec{v}$ of the random walk algorithm, the personalized teleportation probability for node n_i is set as C_i except for the entry-point node. That is, $v_i = C_i$ for $i = 2,3,...,|V|$. Hence, the random walker is more likely to jump to anomaly related nodes when random teleportation occurs. Staying in the entry-point node is meaningless for the random walker. Therefore, the value in the preference vector corresponding to the entry-point node n_1 is assigned 0. Above all, the personalized PageRank vector $\vec{x}$ can be computed as follows:

$$\vec{x} = d \cdot P\vec{x} + (1-d) \cdot \vec{v} \tag{7}$$

where d is the damping parameter in [0, 1], indicating the probability to move in the transition matrix P. Consider an extreme case when $d = 0$, the random walker is equivalent to using only the correlation score.

3.5 | Result calibration

TraceRank uses the scores obtained from the spectrum analysis to calibrate the ranked list of the random walk algorithm. Although the random walk algorithm gives a relatively accurate ranked list, it does not consider any information of a single trace and the latency of each node. As for the spectrum method, it may rank some services with the same rank scores, because those tightly coupled services may share the same spectrum score. Thus, inspecting the latency of one node to check whether it deviates from the normal state is necessary. Considering the relatively accurate ranked list that the random walk algorithm provides, we use the spectrum score and the anomaly degree to correct the ranked list. The algorithm is summarized in Algorithm 2, and the parameter γ usually ranges from 2 to 3. For example, let S_a, S_b, and S_c be the *Top*@3 services in the ranked list obtained by the random walk algorithm. Although S_a is not in the *Top*@3 obtained by the spectrum method, it is put behind. We assume that the latency of anomalous traces on node S_c does not exceed the range given by the latency of normal traces and parameter γ. Therefore, the final ranked list is S_b, S_a.

Algorithm 2 The ranked list correction algorithm

Require: The *Top*@*k* list obtained by the spectrum method, *LS*; the *Top*@*k* list obtained by the random walk algorithm, *LR*; the mean and standard deviation of latency of each service in the normal requests, D_n; the mean latency of each service in the anomalous requests, D_a;

Ensure: The final ranked list, *L*;

```
L, temp ⇐ list()
for svc in LR do
    if D_a[svc] > D_n[svc].mean + γ * d_n[svc].std then        ▷ Evaluate whether the latency falls outside the normal range
        if svc in LS then
            L.append(svc)                                      ▷ Confirmed by the spectrum method
        else
            temp.append(svc)                                   ▷ Not confirmed by the spectrum method, descend the ranking
        end if
    end if
end for
L ⇐ L.append(temp)                                             ▷ Splice the temp behind L
return L
```

4 | EXPERIMENTAL EVALUATION

Platform setting. *TraceRank* is evaluated in two microservice benchmarks and an open-source trace dataset. The platform is deployed on a cluster with 10 virtual machines on 5 physical nodes. Each virtual machine has a 2-core 1.7 GHz CPU, 8 GB of RAM, and the operating system is Ubuntu 16.04. The network between the physical nodes is a Gigabit network through the switch. We evaluate *TraceRank* on the Kubernetes platform,[¶] an open-source system for automating deployment, scaling, and management of containerized applications. The parameter σ in Section 3.5 is set as 3 in all experiments. All experimental codes are written in Python 3.6.

Benchmark. (I) TrainTicket[#] is a train ticket seller application implemented by a microservice architecture, including more than 40 kinds of microservices. Due to the limitation of resource, we set one instance for each service. The function of tracing has been enabled in TrainTicket by default. (II) BookInfo is a sample microservice application that displays information about a book, similar to a single catalog entry of an online book store, containing six kinds of microservices. To get the tracing data, we enable tracing by Istio.[‖] (III) To mimic actual requests to the application, a load generator is built to simulate a number of clients keeping the query per second (QPS) about 20 for both benchmarks. The experimental time window is set to 3 min and the time window slides forward per minute. The faults are injected at the last minute. TrainTicket contains more services than other microservice benchmarks, for example, Sock-shop[**] and Hipster Shop.[††] BookInfo is simple and stable, which enables us to test the scalability of *TraceRank*.

Fault injection. According to Occam's razor theory, a complex situation is of low probability. Thus, we inject our system with two simultaneous root causes at most. Figure 7 illustrates the effect of fault injection. Our work focuses on locating the root cause service instances, which are pods in Kubernetes. To mimic real performance problems, we inject faults into containers belonging to specific pods randomly. In microservice environments, the service-level agreement (SLA) has the highest priority. To mimic SLA violations, we inject three kinds of faults such as CPU exhausting, network delay, and I/O burn with ChaosBlade,[‡‡] which is a chaos testing tool for dockers. We inject each kind of fault 20 times in one root cause scenario. We random select two of three kinds of fault 20 times in two root causes scenarios. In order to reduce randomness, we repeat each injection 20 times and calculate the average result as the final result. Each injection would cause the latency of benchmark to violate the corresponding SLO for 5 min.

Real-world microservice system. We also use a trace dataset released by the 2020 AIOps Challenge Event[§§] to evaluate *TraceRank*. This dataset is generated by a real-world production microservice system in China Mobile Zhejiang, which is the biggest communications provider in China. In particular, the workload of the microservice system is a replica of the real-world workload. Note that because this event does not only focus on microservice applications, in this paper, we only selected those faults related to microservices. Only one fault is injected one time and every fault lasts for 5 min in this dataset.

4.1 | Evaluation metric

We use the *Top@K* score to evaluate *TraceRank*. If all the root causes are included by the *Top@K* ranking list, we say the inference result is correct. In other words, even only one root cause is not included in the *Top@K* list, the result is considered to be wrong. To quantify overall performance, we use the standard Precision and Recall metrics. Let N_{tp}, N_{fn}, and N_{fp} denote the number of true positives, false negatives, and false positives, respectively. We calculate the Precision and Recall metrics in the standard way as follows:

$$Precision = \frac{N_{tp}}{N_{tp} + N_{fp}}, Recall = \frac{N_{tp}}{N_{tp} + N_{fn}}. \tag{8}$$

4.2 | Effectiveness evaluation

To verify the effectiveness of *TraceRank*, we leverage *TraceRank* to detect and localize the injected faults.

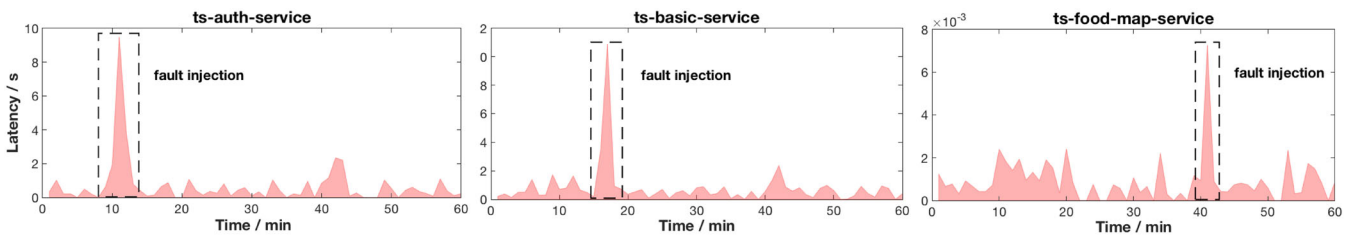

FIGURE 7 The effect of fault injection on different services

TABLE 5 The results of hierarchical clustering with different thresholds in the TrainTicket benchmark

Threshold t	Kind of requests	Groups	Precision of detection (%)
0	21	25	99
1	21	25	99
2	21	17	96
3	21	15	90
4	21	15	90
5	21	14	86

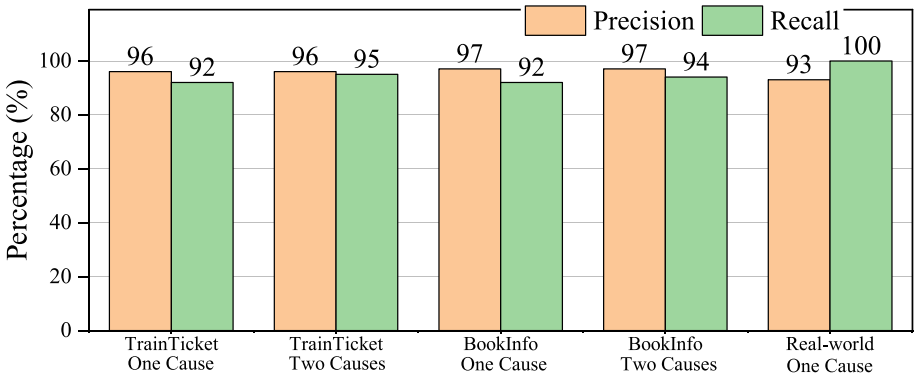

FIGURE 8 The Precision and Recall of anomaly detection module in *TraceRank*

4.2.1 | Effectiveness of anomaly detection module

Table 5 shows the results of the hierarchical clustering with different thresholds in the TrainTicket benchmark. We find that there are 21 kinds of requests in TrainTicket after analyzing the traces of TrainTicket. From Table 5, we can find that when t is less than 1, the same kind of requests are clustered into multiple groups. This is because some kinds of requests may have multiple structures. For example, one kind of request in TrainTicket may have 144 or 145 or 146 spans. In addition, the number of groups in the hierarchical clustering decreases with increasing t, and the accuracy of anomaly detection decreases as well. In this paper, we select $t = 2$ because compared with $t = 0, t = 2$ reduces the number of K-means models by 32% at a cost of only 3% decrease in accuracy of anomaly detection.

Figure 8 presents the results of the anomaly detection module in TrainTicket, BookInfo benchmark, and a real-world system. We find that *TraceRank* has high Precision and Recall in all five scenarios. *TraceRank* has a higher Recall when we inject two faults at one time. This is because injecting two faults at one time makes the system's anomalous behavior more severe. The Recall in real-world system is higher than TrainTicket and BookInfo because the anomalies in the real-world system are more severe than our injected faults.

4.2.2 | Effectiveness of root cause localization module

Table 6 shows the fault localization results of specific services in TrainTicket and BookInfo when one fault is injected at a time. From this table, we first observe that *TraceRank* can find the root cause services with high Precision and Recall no matter whether in TrainTicket or BookInfo. We can also observe that the localization results are quite different from service to service. For example, *TraceRank* can achieve 100% in Precision and Recall in order service when a CPU fault is injected, but get a low Precision and Recall in notify service. The result is relevant to the service dependency and the anomaly degree when a fault occurs. Another observation is that the localization result is better on BookInfo than on TrainTicket. The possible reason is that the topology of BookInfo is much simpler than TrainTicket. It is easier to localize root causes in BookInfo. We do not show the result of the real-world system because the its faults are rare and widely distributed. We cannot achieve the same statistics in this system just as the other two systems. Therefore, we show the overall results of real-world system in Figure 9 separately.

TraceRank relies on the service call graph to conduct a random walk algorithm, which can be acquired from traces. For the TrainTicket application, *TraceRank* achieves a high recall in *Top@{1,2,3}* score with one root cause as presented in Figure 9 due to the relatively simple impact caused by a single root cause. In the two root causes case, the result decreases. However, *TraceRank* gives a relatively good result when tolerating one or

TABLE 6 The fault localization results of specific services in TrainTicket and BookInfo when only one fault is injected at each time

	TrainTicket										BookInfo		
	Travel	Order	Route	Station	Train	Pay	Notify	User	Seat	Basic	Product	Details	Ratings
CPU exhausting													
PR@1	86	100	78	92	88	76	64	80	96	86	90	90	92
PR@2	94	100	80	92	90	80	68	80	100	88	90	94	94
Recall@1	85	100	75	90	90	65	65	75	80	85	85	85	90
Recall@2	95	100	75	90	100	70	75	85	85	90	90	85	90
Network delay													
PR@1	100	100	88	60	100	80	75	84	90	92	88	92	86
PR@2	100	100	88	60	100	86	80	88	90	92	92	96	90
Recall@1	100	100	90	50	100	90	80	85	85	90	90	85	85
Recall@2	100	100	90	55	100	95	90	85	90	95	95	85	85
IO burn													
PR@1	90	95	86	74	90	82	78	86	90	96	88	90	90
PR@2	100	100	90	77	96	90	82	90	90	100	92	92	93
Recall@1	85	100	90	65	75	90	75	80	90	90	85	80	90
Recall@2	95	100	90	65	90	100	85	85	100	90	85	90	95

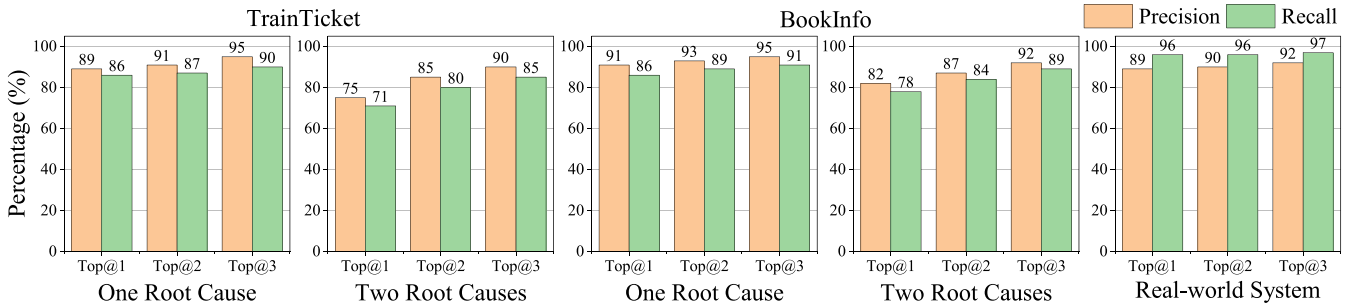

FIGURE 9 The Precision and Recall of RCA in TrainTicket, BookInfo benchmark, and real-world application

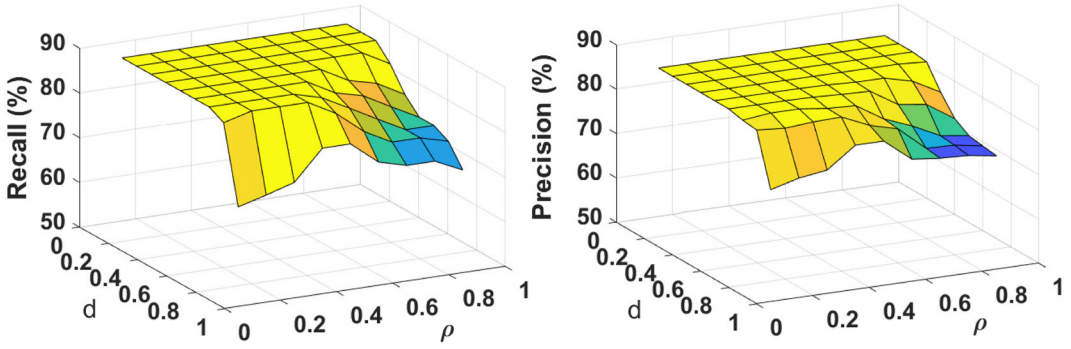

FIGURE 10 Impact of parameter d and ρ on Precision and Recall of *Top*@1 score in the Trainticket benchmark when one root cause is injected

two mistakes in *Top*@3 and *Top*@4. The reason for the decrease in *Top*@2 is that two different services are abnormal simultaneously affecting a number of other services and adding complexity to the fault propagation. Moreover, we also observe some cases where the anomaly detection module is mis-triggered. These cases result in a decrease in Precision.

As for the BookInfo benchmark, we set five replicas for each service. However, the simple service dependencies in the BookInfo do not lead to significant growth of Recall and Precision. That is because the services in BookInfo application simply return plain values without conducting any business logic, which results in a high correlation score between each service and the front-end service. Nevertheless, the anomaly detection module works better in BookInfo than in TrainTicket due to the simplicity of the BookInfo application.

Figure 10 demonstrates the impact of parameters d and ρ on Precision and Recall in *Top@1* score when a single root cause is injected in the TrainTicket application. Intuitively, *TraceRank* achieves the best Precision and Recall when the damping parameter d is low, indicating the random walker takes teleportation frequently. It shows that the correlation score that comprises the teleportation vector is of high precision. However, it does not mean that the random walk method is useless. To our observation, the random walk method provides crucial information when multiple root causes co-occur or the root causes appear in intermediate services along the execution path. Figure 10 also presents that *TraceRank* achieves the highest when ρ equals 4 and 5 on high d, showing that the balanced flexibility and restriction on the service call graph improves the random walk algorithm.

4.3 | Comparisons

To validate the effectiveness of *TraceRank* thoroughly, we compare it with several state-of-the-art methods including MicroRCA,[7] Automap,[17] MS-rank,[18] MicroHECL,[41] GMTA,[42] TraceLingo,[43] TraceAnomaly,[22] Roots,[44] CauseInfer,[9] MonitorRank,[33] Microscope,[20] and T-Rank.[45]

- To compare with MicroRCA, we first construct the attributed graph (i.e., system dependencies) with CPU and memory utilization metrics by the method provided by MicroRCA.[7] Then, we use a personalized PageRank approach to help locate the root causes.
- To compare with Automap[17] and MS-rank,[18] we leverage PC-algorithm[9] to construct the dependencies of service instances with the service latency and throughput metrics. Then, a second-order random walk approach is applied to pinpoint root cause service instances.
- To compare with MicroHECL,[41] we use the end-to-end tracing data to construct the dependency graph of service instances. Then, we localize the root cause by calculating the correlation between the target abnormal services and the downstream service instances in the dependency graph.
- To compare with GMTA,[42] the end-to-end tracing data are adopted to construct the execution paths of requests. The root causes are identified by comparing the execution paths under normal and abnormal situations, which is also used in GMTA.[42]
- To compare with TraceLingo,[43] we directly leverage the provided source code to localize root causes.[¶¶]
- To compare with TraceAnomaly,[22] we directly leverage the provided source code to localize root causes.[##]
- To compare with Roots,[44] we implement the four root cause identification approaches mentioned in Roots.
- To compare with Microscope,[20] we leverage the method presented in Microscope to locate root cause services.
- To compare with CauseInfer,[9] we capture the network packets and leverage lag correlation to find service dependencies. Then, we leverage a depth-first search (DFS) based traversal approach to infer root causes.
- To compare with MonitorRank,[33] we use a random walk approach to find the root cause services, which has been implemented in our previous work.[46]
- To compare T-Rank,[45] we use the Ochiai spectrum-based approach to localize root cause services.

Figures 11 and 12 show the comparisons between different systems in one root cause situation and two root causes situation in the TrainTicket, respectively. From these two figures, we observe that *TraceRank* achieves a consistent better result in Precision and Recall no matter when one fault is injected or two faults are injected simultaneously. *TraceRank* outperforms Microscope by 6% higher in Precision and 7% in Recall in one root cause situation and by 5% higher in Precision in the situation of two root causes. Compared with Roots, we have over 20%

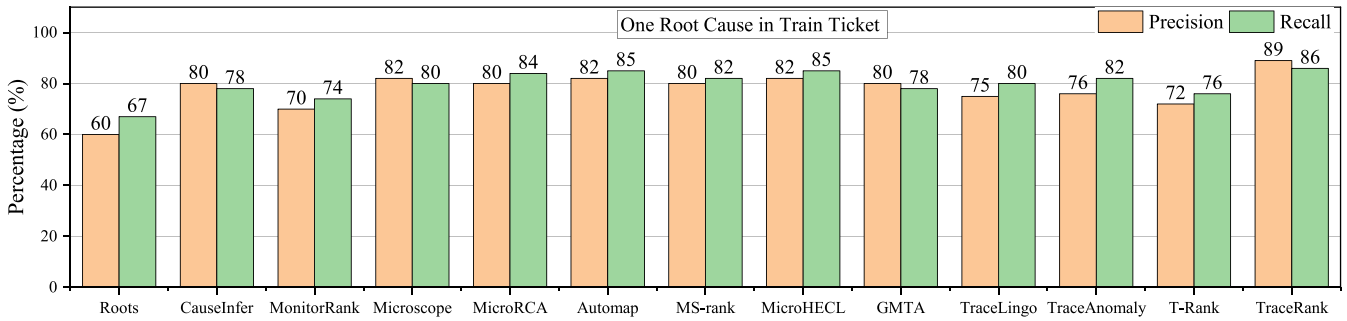

FIGURE 11 The comparison results of different systems with *Top@1* when one root cause is injected at a time in the Trainticket benchmark

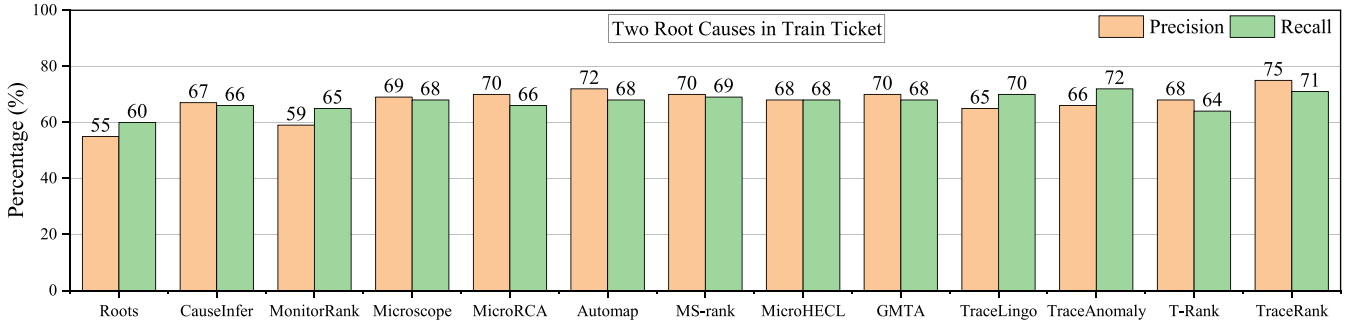

FIGURE 12 The comparison results of different systems with *Top@2* when two root causes are injected at a time in the Trainticket benchmark

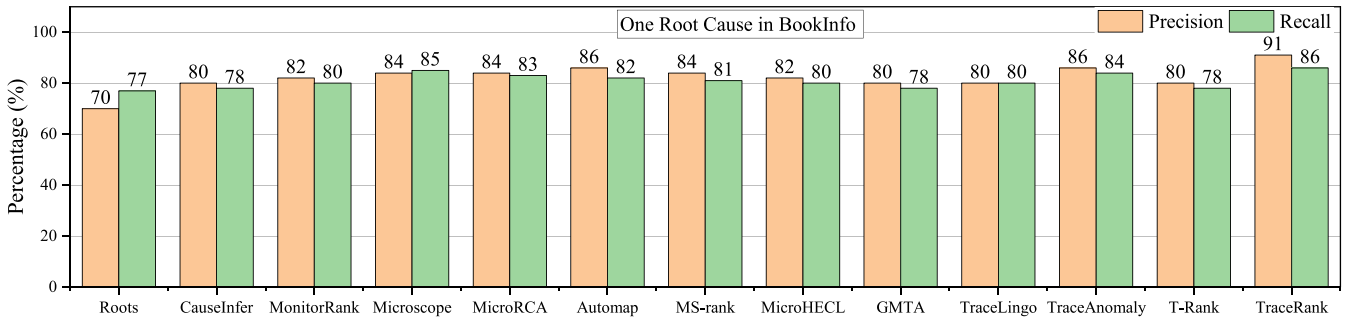

FIGURE 13 The comparison results of different systems with *Top@1* when one root cause is injected at a time in the BookInfo benchmark

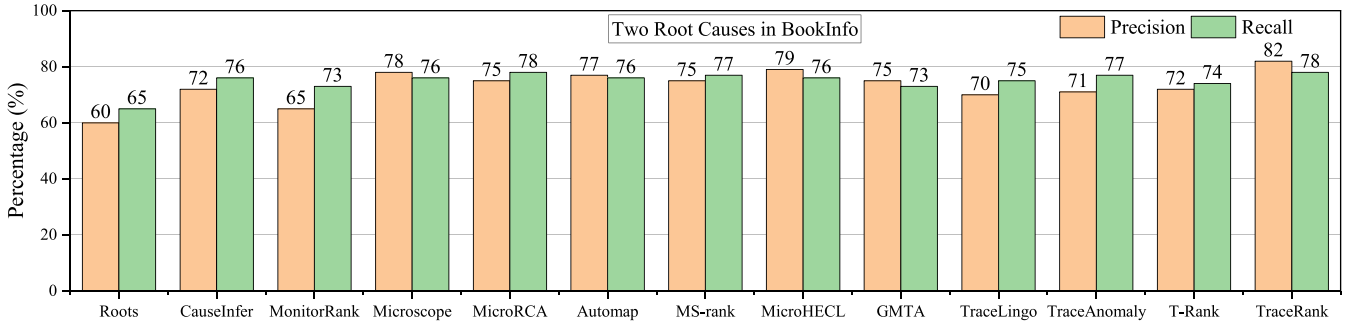

FIGURE 14 The comparison results of different systems with *Top@2* when two root causes are injected at a time in the BookInfo benchmark

higher in Precision. Roots identifies the service which contributes to the most variance of the abnormal service as the root cause. However, in our experiments, we observe that Roots always finds the first upstream service as the culprit rather than the real root causes. That is why its result is not so promising. Similarly, MonitorRank also puts the first upstream service in the first rank. Compared with MicroRCA, MS-Rank, AutoMap, MicroHECL, and GMTA, *TraceRank* not only leverages the service call graph but also the dis-aggregated end-to-end request tracing data. The fine-grained tracing data can provide more direct hints to locate root causes. That is a key point to help *TraceRank* outperform other approaches. Compared with our work, T-Rank[45] only uses the spectrum analysis to find root causes. It cannot distinguish the suspiciousness of the services with the same coverage information. Figure 11 shows that *TraceRank* outperforms T-Rank by more than 10% in Precision when injecting one fault. These results prove that the random walk algorithm can enhance the spectrum analysis. It is necessary to combine the spectrum analysis and the PageRank-based random walk methods.

Figures 13 and 14 show the comparisons between different systems in one root cause situation and two root causes situation in the BookInfo, respectively. *TraceRank* achieves a better result because the BookInfo is simpler than TrainTicket. Overall, *TraceRank* achieves a consistent better result in Precision and Recall than other approaches. Figure 15 demonstrates that comparison results of different systems with *Top@1* in

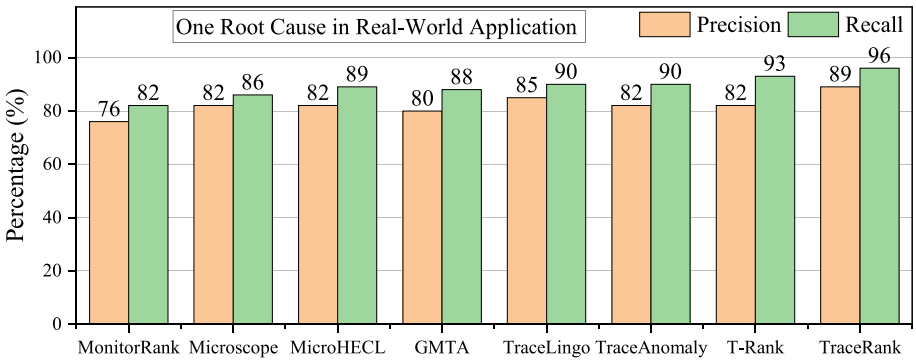

FIGURE 15 The comparison results of different systems with *Top@1* in the real-world application

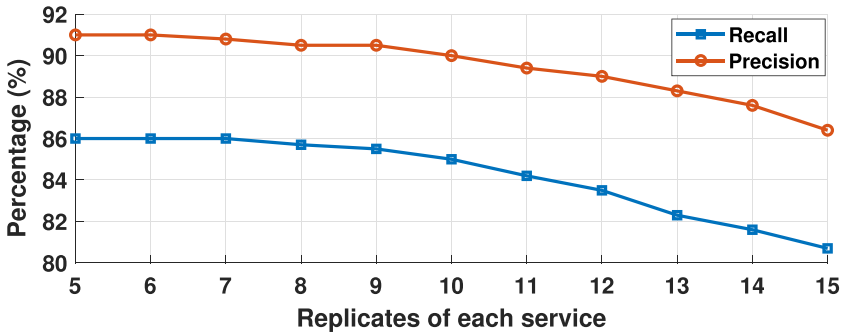

FIGURE 16 The impact of service replicas on Precision and Recall in two root causes case

TABLE 7 Overhead statistics for different modules in *TraceRank*

System module	Overhead (single node)
Client instrumentation	2% ± 1% CPU utilization
Latency extraction	4% ± 1% CPU utilization
Anomaly detection	2.5% ± 1% CPU utilization
Data preparation	2.8 s
Root cause analysis	500 ms

the real-world system. We do not compare *TraceRank* with the approaches (e.g., CauseInfer and Microscope) that require other information that the dataset do not have. From Figure 15, we find that most approaches have better performance in the real-world system than in benchmarks. This is because the faults in the real-world system are simpler and more obvious than in benchmarks. Moreover, *TraceRank* achieves a consistent better result in Precision and Recall than other approaches in the real-world system.

4.4 | Scalability

TraceRank is easy to scale when new services are added in large microservice systems. We have discussed the scalability of *TraceRank* when the system scales out with higher complexity in Section 4.2. In this experiment, we use the BookInfo benchmark and scale all the services replicas to 15 due to limited resources in our platform. The results are presented in Figure 16. We do not find a significant descending trend on both in Recall and Precision: about 5% decrease when replicas are scaled out from 5 to 15. Scaling out a system does add a burden to our random walk method due to more connected nodes are added to the service call graph. However, more replicas can improve the spectrum method by adding different request paths (e.g., some requests visit replica *A*, whereas other requests visit replica *B*). It is obvious that scaling out service replicas only adds up the node complexity of the service call graph. In short, due to the detailed information provided by the end-to-end tracing, increasing the number of services slightly affects the Precision and Recall of *TraceRank*.

4.5 | Overhead

Table 7 shows the overhead of *TraceRank*. It takes about 4% CPU utilization to extract latency and process time from traces. When the RCA is triggered, *TraceRank* computes the spectrum and *Personalized* PageRank score then combines these scores, which consumes 2.5% CPU utilization. The process consumes about 3 s to format tracing data and 500 ms to compute the final score. Obviously, the computation process of *TraceRank* is relatively simple. The overhead of *TraceRank* can be controlled when the request volume is large in production systems.

5 | THREATS TO VALIDITY

In this part, we conduct additional discussions with *TraceRank* from different angles. The threat to validity mainly refers to the generalizability of our approach.

Mount of traces. In Section 3.2, we use a time window Δt to control the volume of traces to be analyzed at one time. The Δt can be adjusted according to the workload of microservice systems. The principle of tuning Δt is to allow as many types of traces as possible to be fed into the model. Moreover, we can also input the trace data that we want to analyze into *TraceRank* rather than using the traces in a time window.

Type of requests. In Section 3.2, we divide traces with a similar structure into a group and do anomaly detection with *K*-means within the group. The core assumption behind this idea is that the traces with a similar structure have similar latency. However, the latency of requests may depend not only on its path but also on the parameters of the requests. When facing such complicated workloads, we need to add more features (e.g., parameters of requests) to the hierarchical clustering model to distinguish this problem.

Number of clusters. In Section 3.2, we use a threshold t to control the number of trace clusters. The value of t is a trade-off between the efficiency and accuracy of the anomaly detection module. If the threshold t is set less than 1, the anomaly detection module needs to maintain a *K*-means model for each type of trace. The results of anomaly detection are accurate, but it is expensive to maintain thousands of models. If the threshold t is set too large, it will allow for all traces to be merged together. This will decrease the accuracy of the anomaly detection module.

6 | RELATED WORK

Anomaly detection and RCA in large distributed systems are vibrant but challenging topics. Extensive methods have been proposed to resolve the two problems. Here, we present the related work as follows.

6.1 | Anomaly detection

Liu et al[47] proposed an approach based on a spatiotemporal feature extraction scheme built on the concept of symbolic dynamics for representing causal interactions. Then, a restricted Boltzmann machine (RBM) is used to learn system-wide patterns to detect anomalies. Nedelkoski et al[48] integrate gated recurrent unit with variational autoencoder to learn a prediction model, which detects anomaly by determining the reconstruction error of the input response time series exceeds an adaptive threshold. In another literature,[49] they propose a method to merge two single-modality LSTM networks, which make use of different modalities of tracing data (i.e., the event sequence and the response time series) to detect anomalies, into one multimodal LSTM model. The multimodal architecture enables the detection of structural and temporal anomalies simultaneously. TraceAnomaly[22] utilizes a service trace vector (STV) to unify the invocation pattern and the response time pattern in a trace and designs a deep Bayesian network with posterior flows to learn the distribution of the normal STV. Anomalous traces are detected if their STVs do not follow the distribution. Scheinert et al[50] present a neural graph method to detect and localize anomalies. It models the components in the distributed cloud application as nodes and their dependencies as edges. Then, they learn the feature vector of each node by applying the graph convolutional neural network. The calculated feature vectors are used to train a classification model. The topology of a distributed system is helpful for anomaly detection. TopoMAD[51] uses the topological information and metrics collected from different components of the cloud system to build an unsupervised multivariate time series anomaly detector. It takes a stochastic seq2seq autoencoder model to evaluate the anomaly score of a component according to the reconstruction error. These anomaly detection approaches can substitute our clustering based method in different scenarios.

6.2 | Root cause analysis

There are numerous RCA methods in the literature. We introduce two main categories of them: the tracing-based methods and the metric-based methods.

6.2.1 | Tracing-based work

Various tools and systems[26,52–54] for end-to-end tracing have been proposed. They build tracing systems to instrument the source code and collect traces. Due to sufficient information tracing data provides, Spectroscope[55] and TraceCompare[56] pinpoint performance problems by comparing requests flows. However, performance problems may be common in a microservice environment due to agile development, auto scaling, and so forth. That fails these approaches to diagnose root causes in such an environment. Magpie[52] constructs causal paths based on OS-level event tracing and clusters those by a string-edit-distance algorithm to help understand complex system behavior. By annotating applications or platforms, Pinpoint[53] uses a probabilistic, context-free grammar to detect anomalies on a per event basis rather than considering whole paths. GMTA[42] is built for microservice trace analysis, supporting streaming data processing, flexible data access, and efficient data storage in the industrial-scale microservice system. It can help site reliability engineering (SRE) teams narrow down the root cause scope of a production problem and retrieve and visualize the error propagation chain. TraceRCA[57] performs RCA with tracing data based on the insight that a microservice with more abnormal and less normal traces passing through it is more likely to be the root cause, which is just like the intuition behind *TraceRank*. MicroHECL[41] constructs service graph dynamically based on traces. It analyzes possible anomaly propagation chains with a pruning strategy to get the candidate root causes, which are ranked based on the service graph and correlation analysis. T-Rank[45] proposes a lightweight spectrum-based performance diagnosis tool to find root causes. However, it cannot handle the services with similar coverage information. Compared with these works, *TraceRank* automatically monitors the microservice system and pinpoints the root causes timely with a good performance when more than one root cause emerge simultaneously.

6.2.2 | Metric-based work

Approaches in this category usually build a dependency graph of services in distributed systems with different approaches. Besides, these works commonly utilize monitoring metrics to detect anomalies, then diagnose problems with the combination of metrics and the dependency graph. CloudRanger[46] takes a heuristic investigation algorithm based on a second-order random walk through the causal relationship of services. NetMedic[58] constructs the component dependencies of programs with predefined templates. Then, it does random walk along with the dependency graph to infers the root causes. Microscope[20] intercepts system calls on network sockets and builds the dependency graph with network information and then finds root cause candidates by comparing the similarity between SLO metrics and the abnormal services. MicroRCA[7] constructs an attribute graph, which including service dependency and performance metrics, to represent the anomaly propagation. Based on the detected anomalies, an anomalous subgraph is extracted from the attribute graph. The personalized PageRank[40] method is used on the anomalous subgraph to locate the root causes. Ma et al. propose MS-Rank[18] and AutoMap[17] to conduct RCA. The main steps of these two methods include generating a causality graph between services dynamically, selecting the appropriate diagnosis metric automatically and identifying the root cause by random walking. MicroDiag[59] constructs a component dependency graph that describes the relationship between entities at different levels and derives a metric causality graph. It weighs the metrics causality graph and ranks culprit metrics. This kind of work does not need to instrument the source code. However, they are not able to provide fine-grained data for analysis compared with end-to-end tracing-based work and that is part of the reason why *TraceRank* performs better.

7 | CONCLUSION AND FUTURE WORK

This paper designs and implements *TraceRank*, a novel system to pinpoint root causes in microservice environments by analyzing end-to-end tracing data. *TraceRank* extracts the latency and processing time for each service from tracing data to conduct the anomaly detection procedure. By combining the personalized PageRank-based random walk algorithm and spectrum analysis, the suspicious root cause services are ranked with higher scores. The experimental evaluation result shows that *TraceRank* achieves a promising precision and recall for RCA. It also can scale out readily in large-scale microservice systems. With the help of *TraceRank*, the operation of microservice systems can be more efficient and effective. As part of future work, we plan to extend *TraceRank* to pinpoint different types of faults by combining more types of metrics. Moreover, other anomaly detection approaches are also planned in future work.

ACKNOWLEDGMENTS

The research is supported by the Key-Area Research and Development Program of Guangdong Province (2020B010165002), the National Natural Science Foundation of China (61802448 and U1811462), the Basic and Applied Basic Research of Guangzhou (202002030328), the Wechat Rhino-Bird Joint Research Program, and the Natural Science Foundation of Guangdong Province (2019A1515012229). The corresponding author is Pengfei Chen.

DATA AVAILABILITY STATEMENT

Data available on request from the authors.

ORCID

Pengfei Chen https://orcid.org/0000-0003-0972-6900

ENDNOTES

* https://opentracing.io

† https://www.jaegertracing.io/

‡ https://github.com/istio/istio/tree/master/samples/Bookinfo

§ https://www.elastic.co/

¶ https://kubernetes.io/

https://github.com/FudanSELab/train-ticket

‖ https://istio.io

** https://github.com/microservices-demo/microservices-demo

†† https://github.com/GoogleCloudPlatform/microservices-demo

‡‡ https://github.com/chaosblade-io/chaosblade

§§ https://iops.ai/competition_detail/?competition_id=15&flag=1

¶¶ https://github.com/serina-hku/TraceLingo

https://github.com/NetManAIOps/TraceAnomaly

REFERENCES

1. Balalaie A, Heydarnoori A, Jamshidi P. Microservices architecture enables devops: migration to a cloud-native architecture. *IEEE Software*. 2016;33(3):42-52.
2. Yu G, Chen P, Zheng Z. Microscaler: automatic scaling for microservices with an online learning approach. In: Proceedings of the 2019 IEEE international conference on web servics. IEEE; 2019:68-75.
3. Yu G, Chen P, Zheng Z. Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach. IEEE Transactions on Cloud Computing; 2020:1-1.
4. Zhou H, Chen M, Lin Q, Wang Y, She X, Liu S, Gu R, Ooi BC, Yang J. Overload control for scaling Wechat microservices. In: Proceedings of the 2018 acm symposium on cloud computing. Association for Computing Machinery; 2018:149-161.
5. Dunn E, Richard E, Alramli N. How we deploy 300 times a day. Accessed: 2021-10-06; 2013.
6. Heorhiadi V, Rajagopalan S, Jamjoom H, Reiter MK, Sekar V. Gremlin: systematic resilience testing of microservices. In: Proceedings of the 36th IEEE international conference on distributed computing systems. IEEE Computer Society; 2016:57-66.
7. Wu L, Tordsson J, Elmroth E, Kao O. Microrca: root cause localization of performance issues in microservices. In: Proceedings of the 2020 IEEE/IFIP network operations and management symposium; 2020:1-9.
8. Nguyen H, Shen Z, Tan Y, Gu X. Fchain: Toward black-box online fault localization for cloud systems. In: 2013 IEEE 33rd International Conference on Distributed Computing Systems IEEE; 2013:21-30.
9. Chen P, Qi Y, Zheng P, Hou D. Causeinfer: automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. In: Proceedings of infocom 2014 IEEE; 2014:1887-1895.
10. Thalheim J, Rodrigues A, Akkus IE, Bhatotia P, Chen R, Viswanath B, Jiao L, Fetzer C. Sieve: actionable insights from monitored metrics in distributed systems. In: Proceedings of the 18th acm/ifip/usenix middleware conference ACM; 2017:14-27.
11. Nagaraj K, Killian CE, Neville J. Structured comparative analysis of systems logs to diagnose performance problems. In: Proceedings of the 9th USENIX symposium on networked systems design and implementation. USENIX Association; 2012:353-366.
12. Farshchi M, Schneider J-G, Weber I, Grundy JC. Experience report: anomaly detection of cloud application operations using log and cloud metric correlation analysis. In: Proceedings of the 26th IEEE international symposium on software reliability engineering. IEEE Computer Society; 2015:24-34.
13. Wang L, Zhao N, Chen J, Li P, Zhang W, Sui K. Root-cause metric location for microservice systems via log anomaly detection. In: Proceedings of the 2020 IEEE international conference on web services. IEEE; 2020:142-150.
14. Ma M, Yin Z, Zhang S, et al. Diagnosing root causes of intermittent slow queries in large-scale cloud databases. *Proc VLDB Endowment*. 2020;13(8):1176-1189.
15. Mace J. End-to-end tracing: adoption and use cases. Survey, Brown University; 2017.
16. Yu G, Chen P, Chen H, et al. Microrank: end-to-end latency issue localization with extended spectrum analysis in microservice environments. In: Proceedings of the the web conference 2021. ACM / IW3C2; 2021:3087-3098.
17. Ma M, Xu J, Wang Y, Chen P, Zhang Z, Wang P. Automap: diagnose your microservice-based web applications automatically. In: Proceedings of the web conference 2020. Association for Computing Machinery; 2020:246-258.
18. Ma M, Lin W, Pan D, Wang P. Ms-rank: multi-metric and self-adaptive root cause diagnosis for microservice applications. In: Proceedings of the 2019 ieee international conference on web services. IEEE; 2019:60-67.

19. Zhou X, Peng X, Xie T, Sun J, Ji C, Liu D, Xiang Q, He C. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM; 2019:683-694.
20. Lin J, Chen P, Zheng Z. Microscope: pinpoint performance issues with causal graphs in micro-service environments. In: Proceedings of the 2018 International Conference on Service-oriented Computing Springer; 2018:3-20.
21. Roy A, Das R, Zeng H, Bagga J, Snoeren AC. Understanding the limits of passive realtime datacenter fault detection and localization. *IEEE/ACM Trans Netw*. 2019;27(5):2001-2014.
22. Liu P, Xu H, Ouyang Q, et al. Unsupervised detection of microservice trace anomalies through service-level deep Bayesian networks. In: Proceedings of the 2020 IEEE 31st International Symposium on Software Reliability Engineering. IEEE; 2020:48-58.
23. Page L, Brin S, Motwani R, Winograd T. The pagerank citation ranking: bringing order to the web. *Tech. report*, Stanford InfoLab; 1999.
24. Zhang M, Li X, Zhang L, Khurshid S. Boosting spectrum-based fault localization using pagerank. In: Proceedings of the 26th ACM Sigsoft International Symposium on Software Testing and Analysis ACM; 2017:261-272.
25. Sigelman BH, Barroso LA, Burrows M, et al. Dapper, a large-scale distributed systems tracing infrastructure, Google Technical report, Google; 2010.
26. Fonseca R, Porter G, Katz RH, Shenker S, Stoica I. X-trace: a pervasive network tracing framework. In: Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation USENIX Association; 2007:20-20.
27. Thereska E, Salmon B, Strunk J, Wachs M, Abd-El-Malek M, Lopez J, Ganger GR. Stardust: tracking activity in a distributed storage system. In: Acm sigmetrics performance evaluation review ACM; 2006:3-14.
28. Gan Y, Zhang Y, Hu K, Cheng D, He Y, Pancholi M, Delimitrou C. 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. Association for Computing Machinery; 2019:19-33.
29. Jones JA, Harrold MJ, Stasko JT. Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering. ACM; 2002:467-477.
30. Jones JA, Harrold MJ. Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of 20th IEEE/ACM International Conference on Automated Software Engineering. IEEE / ACM; 2005:273-282.
31. Xiong Y, Wang J, Yan R, Zhang J, Han S, Huang G, Zhang L. Precise condition synthesis for program repair. In: Proceedings of the 39th International Conference on Software Engineering. IEEE / ACM; 2017:416-426.
32. Jiang J, Wang R, Xiong Y, Chen X, Zhang L. Combining spectrum-based fault localization and statistical debugging: an empirical study. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering. IEEE / ACM; 2019:502-514.
33. Kim M, Sumbaly R, Shah S. Root cause detection in a service-oriented architecture. In: Proceedings of the 2013 International Conference on Measurement and Modeling of Computer Systems ACM; 2013:93-104.
34. Baset SA. Cloud SLAs: present and future. *ACM SIGOPS Oper Syst Rev*. 2012;46(2):57-66.
35. Michlmayr A, Rosenberg F, Leitner P, Dustdar S. Comprehensive QoS monitoring of web services and event-based SLA violation detection. In: Proceedings of the 4th International Workshop on Middleware for Service Oriented Computing. Association for Computing Machinery; 2009:1-6.
36. Shan H, Chen Y, Liu H, Zhang Y, Xiao X, He X, Li M, Ding W. Diagnosis: unsupervised and real-time diagnosis of small- window long-tail latency in large-scale microservice platforms. In: Proceedings of the the World wide Web Conference. ACM; 2019:3215-3222.
37. Las-Casas P, Mace J, Guedes D, Fonseca R. Weighted sampling of execution traces: capturing more needles and less hay. In: Proceedings of the 2018 ACM Symposium on Cloud Computing. Association for Computing Machinery; 2018:326-332.
38. Rui A, Zoeteweij P, Gemund AJCV. On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques - Mutation, 2007. taicpart-mutation; 2007:89-98.
39. Phillips T, GauthierDickey C, Thurimella R. Using transitivity to increase the accuracy of sample-based Pearson correlation coefficients. In: Proceedings of the 12th International Conference of Data Warehousing and Knowledge Discovery, Vol. 6263. Springer; 2010:157-171.
40. Jeh G, Widom J. Scaling personalized web search. In: Proceedings of the 12th International Conference on World Wide Web ACM; 2003:271-279.
41. Liu D, He C, Peng X, et al. Microhecl: high-efficient root cause localization in large-scale microservice systems. In: Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice. IEEE; 2021:338-347.
42. Guo X, Peng X, Wang H, Li W, Jiang H, Ding D, Xie T, Su L. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery; 2020:1387-1397.
43. Xu Y, Zhu Y, Qiao B, et al. Tracelingo: trace representation and learning for performance issue diagnosis in cloud services. In: Proceedings of the 2021 IEEE/ACM International Workshop on Cloud Intelligence. IEEE; 2021:37-40.
44. Jayathilaka H, Krintz C, Wolski R. Performance monitoring and root cause analysis for cloud-hosted web applications. In: Proceedings of the 26th International Conference on World Wide Web International World Wide Web Conferences Steering Committee; 2017:469-478.
45. Ye Z, Chen P, Yu G. T-rank: a lightweight spectrum based fault localization approach for microservice systems. In: Proceedings of the 21st IEEE/ACM international symposium on cluster, cloud and internet computing. IEEE; 2021:416-425.
46. Wang P, Xu J, Ma M, Lin W, Pan D, Wang Y, Chen P. Cloudranger: root cause identification for cloud native systems. In: 2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID) IEEE; 2018:492-502.
47. Liu C, Ghosal S, Jiang Z, Sarkar S. An unsupervised spatiotemporal graphical modeling approach to anomaly detection in distributed CPS. In: Proceedings of the Cyber-Physical Systems (ICCPS), 2016 ACM/IEEE 7th International Conference on IEEE; 2016:1-10.
48. Nedelkoski S, Cardoso J, Kao O. Anomaly detection and classification using distributed tracing and deep learning. In: Proceedings of the 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. IEEE/ACM; 2019:241-250.
49. Nedelkoski S, Cardoso JS, Kao O. Anomaly detection from system tracing data using multimodal deep learning. In: Proceedings of the 12th IEEE International Conference on Cloud Computing. IEEE; 2019:179-186.
50. Scheinert D, Acker A, Thamsen L, Geldenhuys MK, Kao O. Learning dependencies in distributed cloud applications to identify and localize anomalies. In: Proceedings of the 2021 IEEE/ACM International Workshop on Cloud Intelligence; 2021:7-12.

51. He Z, Chen P, Li X, Wang Y, Yu G, Chen C, Li X, Zheng Z. A spatiotemporal deep learning approach for unsupervised anomaly detection in cloud systems. IEEE Transactions on Neural Networks and Learning Systems; 2020:1-15.
52. Barham P, Donnelly A, Isaacs R, Mortier R. Using magpie for request extraction and workload modelling. In: Proceedings of the 6th Symposium on Operating System Design and Implementation. USENIX Association; 2004:259-272.
53. Chen Y-YM, Accardi AJ, Kiciman E, Patterson DA, Fox A, Brewer EA. Path-based failure and evolution management. In: Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation. USENIX; 2004:1-14.
54. Kaldor J, Mace J, Bejda M, et al. Canopy: an end-to-end performance tracing and analysis system. In: Proceedings of the 26th Symposium on Operating Systems Principles ACM; 2017:34-50.
55. Sambasivan RR, Zheng AX, De Rosa M, et al. Diagnosing performance changes by comparing request flows. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation USENIX; 2011:43-56.
56. Doray F, Dagenais M. Diagnosing performance variations by comparing multi-level execution traces. *IEEE Trans Parallel Distrib Syst*. 2017;28(2): 462-474.
57. Li Z, Chen J, Jiao R, et al. Practical root cause localization for microservice systems via trace analysis. In: Proceedings of the 2021 IEEE/ACM 29th International Symposium on Quality of Service IEEE; 2021:1-10.
58. Kandula S, Mahajan R, Verkaik P, Agarwal S, Padhye J, Bahl P. Detailed diagnosis in enterprise networks. *ACM SIGCOMM Comput Commun Rev*. 2009; 39(4):243-254.
59. Wu L, Tordsson J, Bogatinovski J, Elmroth E, Kao O. Microdiag: fine-grained performance diagnosis for microservice systems. In: Proceedings of the 2021 IEEE/ACM International Workshop on Cloud Intelligence. IEEE; 2021:31-36.

How to cite this article: Yu G, Huang Z, Chen P. TraceRank: Abnormal service localization with dis-aggregated end-to-end tracing data in cloud native systems. *J Softw Evol Proc*. 2021;e2413. doi:10.1002/smr.2413