MicroRank: End-to-End Latency Issue Localization with Extended Spectrum Analysis in Microservice Environments

Guangba Yu
Sun Yat-Sen University, China
yugb5@mail2.sysu.edu.cn

Pengfei Chen*
Sun Yat-Sen University, China
chenpf7@mail.sysu.edu.cn

Hongyang Chen
Sun Yat-Sen University, China
chenhy95@mail2.sysu.edu.cn

Zijie Guan†
Tencent, China
byteguan@tencent.com

Zicheng Huang
Sun Yat-Sen University, China
huangzch8@mail2.sysu.edu.cn

Linxiao Jing
Sun Yat-Sen University,China
jinglx3@mail2.sysu.edu.cn

Tianjun Weng
Sun Yat-Sen University, China
wengtj3@mail2.sysu.edu.cn

Xinmeng Sun
Sun Yat-Sen University, China
sunxm25@mail2.sysu.edu.cn

Xiaoyun Li
Sun Yat-Sen University, China
lixy223@mail2.sysu.edu.cn

ABSTRACT

With the advantages of flexible scalability and fast delivery, microservice has become a popular software architecture in the modern IT industry. However, the explosion in the number of service instances and complex dependencies make the troubleshooting extremely challenging in microservice environments. To help understand and troubleshoot a microservice system, the end-to-end tracing technology has been widely applied to capture the execution path of each request. Nevertheless, the tracing data are not fully leveraged by cloud and application providers when conducting latency issue localization in the microservice environment.

This paper proposes a novel system, named *MicroRank*, which analyzes clues provided by normal and abnormal traces to locate root causes of latency issues. Once a latency issue is detected by the *Anomaly Detector* in *MicroRank*, the cause localization procedure is triggered. *MicroRank* first distinguishes which traces are abnormal. Then, *MicroRank*'s *PageRank Scorer* module uses the abnormal and normal trace information as its input and differentials the importance of different traces to extended spectrum techniques . Finally, the spectrum techniques can calculate the ranking list based on the weighted spectrum information from *PageRank Scorer* to locate root causes more effectively. The experimental evaluations on a widely-used open-source system and a production system show that *MicroRank* achieves excellent results not only in one root cause situation but also in two issues that happen at the same time. Moreover, *MicroRank* makes 6% to 22% improvement in recall in localizing root causes compared to current state-of-the-art methods.

*Pengfei Chen is the corresponding author.
†Work was done while the author was a master student at Sun Yat-sen University

CCS CONCEPTS

• **Software and its engineering → Cloud computing**; • **General and reference → Reliability**; **Performance**.

KEYWORDS

Microservice, end-to-end tracing, root cause localization, PageRank, spectrum analysis

ACM Reference Format:
Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. 2021. *MicroRank*: End-to-End Latency Issue Localization with Extended Spectrum Analysis in Microservice Environments. In *Proceedings of the Web Conference 2021 (WWW '21), April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3442381.3449905

1 INTRODUCTION

By adopting microservice architectures, modern enterprises reap many benefits, from better scalability of components, higher developer productivity and smaller programming language restrictions [8, 19, 37, 40]. Each service in the microservice system runs as a set of instances on one or multiple machines and communicates with other microservices through message passing [40]. This leads to an explosion in the number of processes that application providers need to manage and a larger challenge of root cause localization. The application providers have to handle the complex distributed environment to meet availability constraints and strict Service Level Objective (SLO).

Peng et.al. [41] has divided microservice faults into four categories such as *configuration faults* and *resource faults*. Most of them manifest themselves in terms of service latency increase or request time-out , thus latency is the main consideration of SLO [3, 23, 30]. Naturally, it is necessary to quickly detect and diagnose latency issues to keep the systems running in high performance. Nowadays, some of the monitoring tools (e.g., Prometheus[1]) can observe what generally happens in a microservice instance (e.g., the mean latency per minute). These metrics of a single instance are definitely helpful, but they tell us very little about the relations among different instances. For example, the application operators

[1]Prometheus, https://prometheus.io/

do see a latency spike when there was a latency issue in microservice system, but they do not have enough fine-grained context to explain the problem.

Therefore, the application operators wish to have a detailed view of what exactly happened in microservice systems. The end-to-end tracing takes a request-centric view [31] in the microservice environment. It captures the detailed execution of causally-related operations performed by the service instances of a microservice system, and reports them to tracing collector (e.g., Jaeger[2]). Then, the tracing collector constructs the whole paths of request as a graph of events and causal edges between them, which we call a "trace". When a latency issue emerged, application providers can reason about how the microservice system processed the requests by extracting information from the tracing data.

The information provided by the tracing data is rich, which allows application operators to detect latency issues at request level and locate root causes at service instance level. However, analyzing tracing data manually is inefficient and depends heavily on expert knowledge due to the following challenges.

- **Complex tracing paths.** With the microservice architecture, an application is decoupled into multiple loosely distributed fine-grained services with complex interactions [19]. Furthermore, each microservice may have multiple instances to serve requests, which further increases the complexity of trace paths. The complex structure of a microservice system makes it difficult to track the process of fault propagation at service instance level.
- **Multiple cause candidates.** Because upstream services' performance is always dependent on downstream services, a downstream service anomaly may result in multiple anomalies in other upstream services. It means that we need to find root causes in a huge potential cause set.
- **Dynamic runtime environment.** A microservice system usually has a rapid update. New features may be continuously integrated and deployed into each of microservice over time. In addition, microservices usually run in the dynamic container-based environment where their states change frequently, i.e., from running to stop. The anomaly detection and root cause diagnosis methods should adapt to these dynamic environments with a low cost.

Over the years, many approaches have been proposed to locate root causes in large distributed systems [5, 9, 19, 25, 30, 33]. CauseInfer [5] and Sieve [33] build causality graphs of system components, then they pinpoint root causes with these graphs, but their methods cannot get the exact direction of dependency between two service instances. Roots [9] automatically identifies the root causes of performance anomalies in web applications deployed in PaaS clouds, but it is not natively designed for microservice systems. MicroScope [19] maintains a service dependency graph via PC-algorithm. However, MicroScope only considers abnormal traces and leaves out the information provided by normal traces.

To address the drawbacks of existing work and new challenges in microservice systems, this paper introduces a novel approach, named *MicroRank*, which is based on extended spectrum analysis to identify the latency issues in microservice systems. It primarily comprises four procedures including *Anomaly Detector*, *Data Preparator*, *PageRank Scorer* and *Weighted Spectrum Ranker*. Our method is compatible with traces of multiple widely-used microservice trace standards (e.g., OpenTracing [3], OpenCensus [4] and OpenTelemetry [5]). Once a latency issue is detected by the *Anomaly Detector* in *MicroRank*, the cause localization procedure is triggered. *Data Preparator* first distinguishes which traces are abnormal and mind the relations of service instances and traces. Then, *MicroRank*'s *PageRank Scorer* module uses the abnormal and normal trace information as its input and differentiates the importance of different traces to extended spectrum techniques. Finally, the *Wighted Spectrum Ranker* outputs a ranked list of potential root causes based on the weighted spectrum information from *PageRank Scorer*. This ranked list can decrease the overall search space for the application operators significantly. The results in a widely-used open-source system and a production system show that *MicroRank* can locate the root causes effectively and efficiently. Compared to previous cause localization methods such as CauseInfer [5] and Sieve [33], our approach is more effective.

Overall, the contributions of this paper are four-fold:

- To the best of our knowledge, *MicroRank* is the first approach to weave the clues provided by normal and anomalous traces to conduct root cause localization for microservice applications.
- We propose a novel root cause location approach in microservice environments based on the extended spectrum analysis. PageRank is introduced to strengthen the effectiveness of spectrum analysis by considering the importance of different traces.
- We instrument the OpenTelemetry tracing API into Hipster-Shop[6] microservice benchmark provided by Google Cloud. Our modifications give this benchmark[7] the ability of end-to-end tracing that it did not have before.
- We design and implement a prototype, namely *MicroRank*, to locate root causes of latency issues in microservice systems. We conduct extensive experiments based on a widely-used open-source microservice system and a production microservice system with 157 faults in total. Experimental results demonstrate the effectiveness of *MicroRank* over state-of-the-art methods.

2 BACKGROUND

2.1 Microservice End-to-end Tracing

As stated in the Dapper[32], modern Internet services are often implemented as complex, large-scale distributed systems, for example, using the popular microservice architectural style. Unlike the traditional host-level tracing tools, such as DTrace[8], end-to-end tracing is primarily focused on profiling requests as they move across many service instances, usually running on many different hosts. The

[2]Jaeger, https://www.jaegertracing.io/

[3]OpenTracing, https://opentracing.io/
[4]OpenCensus, https://opencensus.io/
[5]OpenTelemetry, https://opentelemetry.io
[6]Hipster-Shop, https://github.com/GoogleCloudPlatform/microservices-demo
[7]Hipster-shop with OpenTelemetry, https://github.com/yuxiaoba/Hipster-Shop
[8]DTrace, http://dtrace.org/

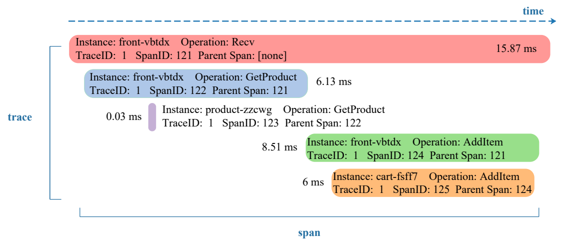
time
trace
Instance: front-vbtdx Operation: Recv
TraceID: 1 SpanID: 121 Parent Span: [none]
15.87 ms
Instance: front-vbtdx Operation: GetProduct
TraceID: 1 SpanID: 122 Parent Span: 121
6.13 ms
0.03 ms
Instance: product-zzcwg Operation: GetProduct
TraceID: 1 SpanID: 123 Parent Span: 122
8.51 ms
Instance: front-vbtdx Operation: AddItem
TraceID: 1 SpanID: 124 Parent Span: 121
6 ms
Instance: cart-fsff7 Operation: AddItem
TraceID: 1 SpanID: 125 Parent Span: 124
span

Figure 1: An example of trace in Hipster-Shop

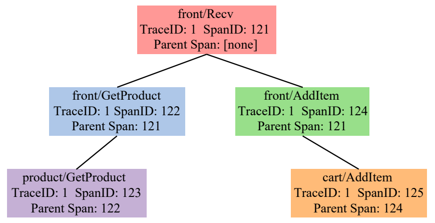

Figure 2: The trace coverage tree of Fig. 1

tracing data provides lots of invaluable information in understanding system behavior, helping with debugging and reasoning about root causes in such complex systems.

In the microservice scenario, programmers usually need to instrument the trace interfaces into each microservice code. The trace interfaces record causality and profiling information about the request with an unique identifier (i.e., trace ID) and report them to the tracing collector (e.g., Jaeger) through calls to a Tracing API like OpenTelemetry. The tracing collector receives the tracing data, normalizes it to a common trace model representation and puts it to the persistent storage (e.g., Elasticsearch [9]).

Before the birth of OpenTelemetry, two incompatible open-source projects OpenTracing and OpenCensus, each with its own standard, have dominated the microservice end-to-end tracing landscape. OpenTelemetry converges best features of both OpenTracing and OpenCensus into a single standard and provides a complete solution to solve the problem of end-to-end tracing in microservice systems. This brings maturity to the OpenTelemetry standard as both previous projects were already mature and production tested. Therefore, we instrument the OpenTelemetry tracing codes into Hipster-Shop to provide a complete tracing view. To the best of our knowledge, it is the first complete microservice benchmark equipped with OpenTelemetry.

Fig. 1 shows a series of spans and their relationships within a trace in Hipster-Shop. As shown in Fig. 1, each span is a named and timed operation and they share a unique traceID in the same trace. The root span (like *front-vbdtx/Recv.* in Fig. 1) is the first span in a trace, while the other spans are related through parent-child relationships. In addition, we attach additional information like the service instance name for each span. Therefore, we can locate root causes at service instance level based on the tracing information.

3 LATENCY ISSUE LOCALIZATION

In this section, we formally describe the problem of latency issue localization at service instance level in a microservice system. Then, we introduce the motivation of applying spectrum-based method to find root causes and its limitations.

3.1 Problem formulation

Before formulating the latency issue localization problem, we summarize the information that we can extract from tracing data.

[9]Elasticsearch, https://www.elastic.co/

- **Latency and Handling Time.** As shown in Fig. 1, the tracing data comprise the end-to-end latency of each trace and the processing time of each operation in one trace.
- **Trace Coverage Graph.** Because each span in one trace comprises the *parent-child* relationship information and service instance information, we can construct the coverage tree for each trace (e.g., Fig. 2 shows the coverage tree of Fig. 1). A coverage tree is a tree where each node represents a service instance operation, and an edge from node v_i to node v_j indicates the calling relation. We can get the complete trace coverage graph (i.e., the connections between service instances operations and the traces) when we aggregate all the coverage trees together.
- **Call Graph.** If we remove the duplicate edges in a trace coverage graph and shift the attention to nodes, we can get the call graph among different service operations.

Based on the above information, the formal description of the latency issue locating problem in microservice systems is as follows. Given a collection of traces in a time window, namely $\mathbf{T} = \{T_1, \cdots, T_n\}$, where $T_i = \{O_1^i, \cdots, O_m^i\}$, and O_j^i demotes one operation in trace i, we get the end-to-end latency of these traces, namely $\mathbf{L} = \{L_1, \cdots, L_n\}$ and the coverage graph $G = (V, E)$, where V is the collection of service instance operations, namely $O_j^i \in V$ and E is the collection of calling relations. The problem is to find out normal traces $\mathbf{T}^n = \{T_1^n, \cdots, T_k^n\}$ and abnormal traces $\mathbf{T}^a = \{T_1^a, \cdots, T_{n-k}^a\}$. Moreover, based on $\mathbf{T}^n$, $\mathbf{T}^a$, and G, finding the root cause service instances related to O_j^i. Our target is to rank service instances directly relevant to the root cause higher than those unrelated to it.

3.2 Motivation

The Spectrum-based fault localization (SBFL) technique is one of the most popular approaches used in program debugging because of its simplicity and efficiency [1, 11, 13, 24]. When given a faulty program P and a set of test cases in which at least one test failed, a typical SBFL approach collects test coverage information for each program element $O \in P$ while running test cases firstly. Next various statistics, e.g., tuple $(O_{ef}, O_{ep}, O_{nf}, O_{np})$, can be extracted based on the test coverage information. Here, O_{ef} denotes the number of failed test cases that cover program element O, O_{ep} denotes the number of passed test cases that cover program element O. O_{nf} denotes the number of failed test cases that do not cover program element O, O_{np} denotes the number of passed test cases that do not cover program element O. Based on the above tuples, several risk evaluation formulae, e.g. Tarantula, have been proposed

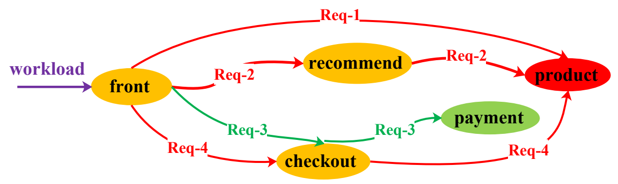

Figure 3: First limitation in spectrum-based method

Table 1: Original Spectrum and *MicroRank* score in Fig. 3

	Initial Spectrum(Tarantula)					MicroRank (Tarantula)				
Service	O_{ef}	O_{ep}	O_{nf}	O_{np}	score	O_{ef}	O_{ep}	O_{nf}	O_{np}	score
front	3	1	0	0	0.5	2.97	0.990	0.0	0.0	0.5
checkout	1	1	2	0	0.25	0.328	0.328	0.656	0.198	0.25
recommend	1	0	2	1	1	0.328	0.0	0.686	0.0	0.4
product	3	0	0	1	1	3.0	0.0	0.0	0.0	0.666
payment	0	1	3	0	0	0.0	1.32	0.0	0.0	0.333

to compute suspicious score that indicates how likely it is to be faulty for each program element.

Given the problem of analyzing end-to-end tracing data, this problem is similar as the software debugging. A request visits several service instances in microservice system while a test case covers several program entities in software testing. The service instances executed by more anomalous requests and less normal requests is more likely to be root cause. Hence we believe it is possible to apply the spectrum-based method to locate the root causes in microservice system. However, when we apply the spectrum-based method directly to find how likely the service instance is the root cause, we find some limitations of this approach.

The first limitation of spectrum-based method. The basic spectrum-based method only focuses on how many normal and anomalous requests cover the service instances but ignores the differences among different requests. To simplify the examples, we consider the microservices as the microservice instance's operations in this section. Figure. 3 shows part of the service dependency graph of Hipster-Shop. The red lines in Fig. 3 denote the anomalous requests in systems and the green line denotes the normal request. The ellipses in Fig. 3 represent the different microservices and the edges show dependencies between services. In this example, we inject 100 ms latency for *product* service to simulate network jam. Due to the fault propagation, the anomaly of *product* propagates to *recommend*, *checkout* and *front*, while *payment* is normal because it does not call *product*. In other words, even if there is only a single anomalous service instance, the anomalous instance may lead to many other instances perform anomalously, which generates a huge potential cause set. Owing to the unpredictability of fault propagation, it is non-trivial to exclude anomalous instances that are caused by fault propagation manually.

When we apply the Tarantula spectrum formula [13] to Fig. 3, we can find that *recommend* and *product* share the same result (details shown in the left half of Table 1). Actually from the operator perspective, the anomalous request *Req-1* in Fig. 3 only covers *front* and *product*. In other words, the anomalous instances are either *front* or *product* and the *recommend* would not be considered first. In addition, because the normal request *Req-4* passes through the *front*, the *product* seems more like the anomalous instance. This example inspires us that if the requests can be weighted based on their ability in localizing root causes, the spectrum-based method in localizing microservice root causes will be more precise.

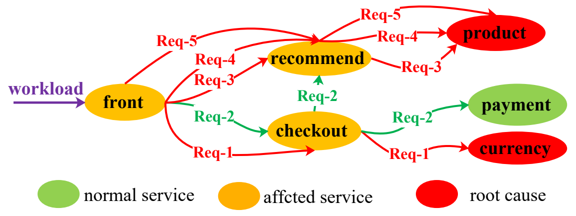

Figure 4: Second limitation in spectrum-based method

The second limitation of spectrum-based method. Unlike test cases which are well-designed by testing engineers, the traces' coverage is relatively unbalanced. In other words, some kinds of requests that cover the same service instances may appear frequently (like *Req-3*, *Req-4* and *Req-5* in Fig. 4) because the user preference (e.g., view products many times). While some kinds of requests (like Req-1 in Fig. 4) may emerge a few times because users access them less often (e.g., only checkout the selected products once). In Fig. 4, we have injected network jam for both *product* and *currency* to stimulate the scenario where two errors occur at the same time. When we apply the Jaccard spectrum formula (shown in Table 1) to Fig. 4, we can find that *recommend* get higher score result than the root cause *currency*. The season is that the spectrum-based methods assume that the test cases are well-designed and they would not cover the same instances many times. This example inspires us that we should give more consideration to the kinds of requests, which appear fewer times to balance the occurrence number of different kinds of requests.

4 SYSTEM DESIGN

4.1 System overview

In this paper, we proposed an extended spectrum method with PageRank, named *MicroRank*, to find service instances that potentially contribute to the latency issues. To rank the service instance autonomously, *MicroRank*, illustrated in Fig. 5, consists of four modules, namely *Anomaly Detector*, *Data Preparator*, *PageRank Scorer* and *Weighted Spectrum Ranker.* We briefly introduce each module in this subsection and later explain them in detail in the following subsections.

First, *Anomaly Detector* module which is presented in Section 4.2 continually monitors the system by comparing the expected latency and real latency of each trace within a sliding time window (30 seconds in this paper). If the real latency of one trace is larger than its expected latency, *Anomaly Detector* determines that the system is at an abnormal state and triggers the root causes localization procedure. Second, *Data Preparator* presented in Section 4.3 distinguishes whether the traces in that time window is abnormal or not and construct both the call graph and operation-trace graph for the *PageRank Scorer* . Third, the *PageRank Scorer* presented in Section 4.4 can generate anomalous scores and normal scores for each operations in parallel based on the anomalous and normal tracing information from *Data Preparator* . In the final *Weighted Spectrum Ranker* presented in Section 4.5 takes the scores from *PageRank Scorer* as input to update weighted spectra and applies ranking formulae to output an ordered list for operators to examine. In the best case, the first service instance presented in the ranked output of *MicroRank* is the exact root cause of that anomaly.

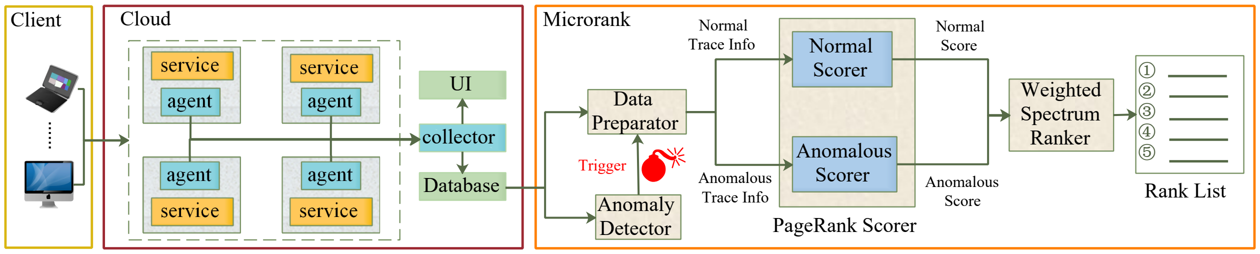

Figure 5: The framework of *MicroRank*

4.2 Anomaly Detector

Before localizing root causes, we need to distinguish whether the target microservice system has latency issues first. Previous approaches (e.g., [9, 19, 34, 37]) on anomaly detection only detect SLO (Service Level Objective, e.g., average latency per minute) deviations of the front-end service under the condition of the mixture of all kinds of traces. However, traces contain a variable number of operations and each operation has different handing time as shown in Section 2.1. For example, *trace1* covers 7 service instance operations and it has low latency 20 ms, *trace2* covers 40 service instance operations and it has high latency 100 ms because it has handled more tasks. Actually both *trace1* and *trace2* are normal traces. But the *trace2* may be determined as abnormal trace by the methods that mix all kinds of requests because these methods do not consider trace-level details.

In this study, we propose an unsupervised and lightweight trace-level anomaly detection method, which utilizes relations between the operation handling time and the number of operations covered by traces. First *Anomaly Detector* calculates the average handling time μ_o and its standard deviation σ_o of each operation from a period time (e.g., one hour) of normal tracing data offline. For example, $\{"checkout/PlaceOrder" : \{"\mu_o" : 50ms, "\sigma_o" : 20ms\}\}$ shows that the mean handling time of *chekout's PlaceOrder* operation is 50ms. To be specific, we do not need to update the average handling time and its variation unless a series of false positives have occurred.

When *Anomaly Detector* monitors traces in one short sliding time windows, it only needs to mine which operations and how many times these operations covered by one trace. This approach is more lightweight than the methods [41] that encode traces into fixed-length vectors. Inspired by the setting of SLO in Roots[9], the excepted latency $L_{excepted}$ of one trace can be calculated as:

$$L_{excepted} = \sum count_o * (\mu_o + n * \sigma_o), \qquad (1)$$

where $count_o$ denotes the number of times which operation o has been covered by that trace. n is used to adjust upper bound values, we set $n = 1.5$ in this paper. If the excepted latency $L_{excepted}$ is less than the real latency of that trace, the trace would be determined as anomalous trace. Once *Anomaly Detector* detects one anomalous trace, it triggers the root cause localization phase. To avoid detecting the same anomalous state multiple times, we flush the detection window (5 minutes in this paper) after each trigger.

4.3 Data Preparator

In anomaly detection phase, *MicroRank* makes a rough analysis of the tracing data to reduce the cost . For example, *Anomaly Detector*

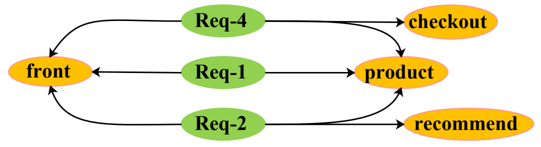

Figure 6: The anomalous operation-trace graph of Fig. 3

only determines whether the operations occur but ignores the relationships among operations. Only when the root cause localization phase is triggered, the *Data Preparator* module in *MicroRank* will extract detailed trace information about the operations' relationships.

Because the anomalous and normal traces may cover different sets of service instances, *Data Preparator* first divides the traces in the last time windows into anomalous trace list and normal trace list based an the Equation 1. For the traceID in anomalous trace list or normal trace list, *Data Preparator* queries its trace information through the unique traceID from persistent storage. Each trace information records the parent-child relation and the service instance operations that the trace has covered. *Data Preparator* can utilize the above information to construct coverage tree of each trace. In this paper, we identify the traces that have the same coverage tree as the same kind of trace. *Data Preparator* also records the number of occurrences of each kind of trace.

In addition, we can get the complete anomalous or normal trace coverage graph when we aggregate the coverage trees together. Then, we can construct the operation-trace graph based on trace coverage graph when we regard traces as part of nodes. For example, Fig. 3 shows the anomalous request-operation graph of Fig. 1. Finally, if we remove duplicate edges in the coverage graph and put the attention to nodes, we can get the real-time call graph among different service instance operations. The anomalous and normal clues serve as the input of *Anomalous PageRank Scorer* and *Normal PageRank Scorer*, respectively.

4.4 PageRank Scorer

As discussed in section 3.2, if all traces are treated equally, using the spectrum-based method to microservice root cause localization directly may lead to inferior performance. Therefore, we add an additional module, named *PageRank Scorer*, before the *Weighted Spectrum Ranker*. *PageRank Scorer* is composed of *Normal Scorer* and *Anomalous Scorer* to deal with the anomalous and normal clues in parallel. *PageRank Scorer* can calculate the weighted spectrum information for *Weighted Spectrum Ranker*.

The PageRank method is used to give each page a relative score of importance by evaluating the quality and quantity of its links. If one webpage contains a hyperlink pointing to another webpage, there is a link between them. Each link from one webpage to another casts a so-called vote, the weight of which depends on the weight of the webpages that link to it.

The basic idea of our approach is to give each operation a weight by evaluating the importance of traces that cover it based on the operation-trace graph(e.g., Fig 6). The importance of traces is based on traces' ability in finding root causes. The key insights of *PageRank Scorer* are that ① if an operation covered by more anomalous traces, it should also be more likely the root cause, and ② if an anomalous trace covers fewer operations the trace should be considered more important because it has a smaller scope to infer real cause, and ③ if a kind of trace appears fewer times the kind of trace should be given more consider to prevent the diversion from the kinds of trace that occur many times.

4.4.1 ***Personalised PageRank Algorithm.***

The standard PageRank is applied to the graph whose nodes are all homogeneous[26, 36]. Therefore, we chose Personalised PageRank [10] because it is a method to analyse the graph of heterogeneous nodes (operations and traces in this paper). Given a digraph $G = \langle V, E\rangle$ that contains n nodes and m edges, and E contains a directed edge $\langle s, t\rangle$ if node s to node t. Let A be the transition matrix of n by n elements, all of the A_{st} will combine into the complete A. The A_{st} is defined as the probability that a random walk starting from s terminates at t, which reflects the importance of t with respect to s. The value of A_{st} can be calculated by:

$$A_{st} = \begin{cases} \frac{1}{|O(s)|}, & t \in O(s) \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

where $O(s)$ denotes the out-neighbors of s. For a given preference vector $\boldsymbol{u}$, the personalized PageRank equation can be written as [10]:

$$\boldsymbol{v} = (1-d)\boldsymbol{A}\boldsymbol{v} + d\cdot \boldsymbol{u}, \tag{3}$$

where d is the damping factor ($0 \le d < 1$), the solution $\boldsymbol{v}$ is the personalized PageRank vector (PPV) for preference vector $\boldsymbol{u}$. If $\boldsymbol{u}$ is the uniform distribution vector $\boldsymbol{u} = [\frac{1}{n}, \ldots, \frac{1}{n}]$, it means that PageRank gives no preference to any pages.

The work in [26] introduced an iterative algorithm to get the approximate solution of Equation(3). The equation of the qth iteration is defined as:

$$\boldsymbol{v}^{(q)} = d\cdot \boldsymbol{A}\boldsymbol{v}^{(q-1)} + (1-d)\cdot \boldsymbol{u}. \tag{4}$$

Each time we run the calculation, we are getting a closer estimate of the final value. The outcome vector denotes the ranked scores of all nodes.

4.4.2 ***Transition Matrix .***

Given the operation-trace graph and call graph information, the transition matrices in our approach can be partitioned as:

$$\boldsymbol{A} = \left[\begin{array}{c|c} \overbrace{\boldsymbol{A}_{oo}}^{operations} & \overbrace{\boldsymbol{A}_{ot}}^{traces} \\ \hline \boldsymbol{A}_{to} & 0 \end{array}\right], \tag{5}$$

where $\boldsymbol{A}_{oo}$ denotes the transition matrix among operations based on the call graph, $\boldsymbol{A}_{ot}$ and $\boldsymbol{A}_{to}$ denote the transition matrices between operations and traces based on the operation-trace graph. *PageRank Scorer* can utilize the $\boldsymbol{A}_{oo}$ to differentiate the service operations covered by the same anomalous and normal traces. For example, after calculating the anomalous trace *Req-1, Req-2, Req-3* in Fig. 3, we can get the anomalous transition matrix $\boldsymbol{A}$ with a vector $[front, recommend, checkout, product, req1, req2, req3]$ below:

$$A = \left[\begin{array}{cccc|ccc} 0 & 0 & 0 & 0 & 1/2 & 1/3 & 1/3 \\ 1/3 & 0 & 0 & 0 & 0 & 1/3 & 0 \\ 1/3 & 0 & 0 & 0 & 0 & 0 & 1/3 \\ 1/3 & 1 & 1 & 0 & 1/2 & 1/3 & 1/3 \\ \hline 1/3 & 0 & 0 & 1/3 & 0 & 0 & 0 \\ 1/3 & 0 & 1 & 1/3 & 0 & 0 & 0 \\ 1/3 & 1 & 0 & 1/3 & 0 & 0 & 0 \end{array}\right].$$

Compared with the continuously changing trace coverage graph, the call graph is relatively stable unless a program change occurs. Therefore, we believe that the operation-trace graph is more important than the call graph in root cause localization. Our approach uses parameter ω ($0 \le \omega \le 1$) to tune the weight of call graph as below:

$$\boldsymbol{A} = \left[\begin{array}{c|c} \omega\, \boldsymbol{A}_{oo} & \boldsymbol{A}_{ot} \\ \hline \boldsymbol{A}_{to} & 0 \end{array}\right]. \tag{6}$$

The matrix $\boldsymbol{A}$ can present the difference among operations and meet the key insight ①.

4.4.3 ***Preference Vector .***

MicroRank uses the preference vector $\boldsymbol{u}$ to demonstrate the impact of trace scope and kind of traces. Preference vector $\boldsymbol{u}$ consists of two sub-vectors: $\boldsymbol{u} = \left[\boldsymbol{u}_o^T, \boldsymbol{u}_t^T\right]^T$, where $\boldsymbol{u}_o$ and $\boldsymbol{u}_t$ denote the preference vector of operations and traces, respectively. Our approach sets the $\boldsymbol{u}_o$ as $\vec{0}$ because we do not consider preference of operations in this module.

For the *Anomalous Scorer*, $\boldsymbol{u}_t$ is set as $[\theta_1, \theta_2, \ldots, \theta_m]^T$. Here,

$$\theta_i = \varphi \cdot \frac{n_i^{-1}}{\sum n_j^{-1}} + (1-\varphi)\cdot \frac{k_i^{-1}}{\sum k_j^{-1}}, \tag{7}$$

where n_i denotes the number of operations covered by anomalous trace i, k_i denotes the number of anomalous traces of trace i's kind, and φ ($0 \le \varphi \le 1$, default $\varphi = 0.5$ in this paper) presents the tradeoff between trace scope and trace kind. The n_i considers the anomalous trace scope based on key insight ② and k_i puts attention to the kind of anomalous traces based on key insight ③. For anomalous traces in Fig. 3, we obtain the preference vector $\boldsymbol{u} = \left[0, 0, 0, 0, \frac{8}{21}, \frac{13}{42}, \frac{13}{42}\right]$.

For the *Normal Scorer*, we only consider the kind of traces for the preference vector because the trace scope does not reflect the importance of normal traces. Therefore, the θ_i is $\frac{n_i^{-1}}{\sum n_j^{-1}}$, where n_i denotes the number of normal traces belong to the kind of trace i.

4.4.4 ***PageRank Score.***

Once we determine the transition matrix $\boldsymbol{A}$ and the preference vector $\boldsymbol{u}$, we set the initial PPV $\boldsymbol{v}^{(0)} = \left[\boldsymbol{v}_o^T, \boldsymbol{v}_t^T\right]^T$. In this paper, $\boldsymbol{v}_o = \left[\frac{1}{N_o}, \frac{1}{N_o}, \ldots, \frac{1}{N_o}\right]$, where N_o is the number of operations in the operation-trace graph. And $\boldsymbol{v}_t = \left[\frac{1}{N_t}, \frac{1}{N_t}, \ldots, \frac{1}{N_t}\right]$, where N_t denotes the number of traces in the operation-trace graph. *Normal Scorer* and *Anomalous Scorer* obtain the normal PPV and anomalous PPV by following the work [39] and the Equation(4). For the tracing data in Fig. 3, we can get the anomalous PPV:

$$F\,[front, recommend, checkout, product] = [0.990, 0.328, 0.328, 1].$$

Table 2: Experiment datasets overview

DataSet	Microservice Benchmark	Fault Injection	Faults Number	Trace Number
$\mathcal{A}$	Hipster-Shop	Inject one fault each time	50	8,384K
$\mathcal{B}$	Hipster-Shop	Injection two faults each time	100	8,244K
C	Production System	Injection one fault each time	7	168k

The result of the anomalous scores indicates that *product* has the highest anomalous score in our example . Actually, *product* is indeed the root cause, which proves the feasibility of our algorithm.

4.5 Weighted Spectrum Ranker

Weighted Spectrum Ranker takes the scores of both normal and anomalous traces generated by *PageRank Scorer* and the number of anomalous and normal traces to construct the spectrum information. The spectrum information of operation O can be computed as:

$$\begin{array}{ll} O_{ef} = F * N_{ef}, & O_{nf} = F * (N_f - N_{ef}) \\ O_{ep} = P * N_{ep}, & O_{np} = P * (N_p - N_{ep}) \end{array}, \tag{8}$$

where F and P represent the anomalous and normal PageRank score of operation O, N_{ef} and N_{ep} denote the number of anomalous and normal traces covered the operation O, N_f and N_p denote the total number of anomalous and normal traces in the current sliding window, respectively. Notice that there are some operations that do not have anomalous scores or some do not have normal scores. If the operation j has not a score, the F or P is set as 0.0001. *MicroRank* then applies spectrum ranking formulae to compute suspiciousness scores for each operation. The operation name (e.g., *front-1/Recv.*) contains the information about which service instance it belongs to. Therefore, *MicroRank* can give a ranking list of service instances for application operators. The right half of Table 1 shows the weighted spectrum information and updated Tarantula scores for each service instance in Fig. 3. According to the Table 1, *MicroRank* boosts the Tarantula spectrum method and ranks *product* as the first, demonstrating the effectiveness of PageRank in locating microservice root causes.

5 EXPERIMENTAL EVALUATIONS

5.1 Datasets

In the study, we used three datasets, namely $\mathcal{A}$, $\mathcal{B}$ and C. $\mathcal{A}$ and $\mathcal{B}$ are based on a widely-used open-source microservice system Hipster-Shop. The major difference between $\mathcal{A}$ and $\mathcal{B}$ is that we inject one fault each time in $\mathcal{A}$ while two faults each time in $\mathcal{B}$. C is based on a production microservice system in China Mobile, the largest telecommunication company in China. Table 2 shows an overview of our experimental datasets.

5.1.1 ***Hipster-Shop Microservice System.*** Hipster-Shop is a web-based e-commerce app where users can browse products, add them to the cart, and purchase them. The application is a widely-used microservice benchmark designed to aid demonstration and testing of microservices and cloud-native technologies [18, 37, 38]. It contains 10 microservices that are implemented in different programming languages and intercommunicate using gRPC. In addition, Hipster-Shop is equipped with a workload generator that simulates concurrent users of that application. The workload law conforms to the real user behavior (more requests are sent to the service *frontend* and *product*, and fewer requests to the service *checkout* and *payment*). To be specific, the initial microservice application open-sourced by Google Cloud does not have complete end-to-end tracing ability. We have given the ability to Hipster-Shop by instrumenting Opentelemetry API for each service. We will open-source the source code of our instrument to GitHub after double-bline review.

Experimental Platform. We construct a distributed testbed that contains 8 virtual machines (VMs) and a ElasticSearch Cluster. Each VM has a 4-core 2.40GHz CPU, 16GB memory and runs with Ubuntu 18.04 OS. We guarantee that all the VMs are in the same local area network to reduce network jitters. We set up a Kubernetes [10] cluster that consists two master nodes and 6 worker nodes based on VMs. The microservice benchmark and Jaeger collector are deployed in that Kubernetes. The service instances first send trace information to Jaeger collector, then Jaeger collector puts aggregated tracing data to a ElasticSearch cluster to persistent tracing data.

Fault Injection. *MicroRank* is applicable to detect and diagnose anomalies that manifest themselves as latency deviations. To mimic latency issues, we inject four types of faults to Hipster-Shop. We use Chaosblade [11], a chaos engineering models, to delay service instance's network packets to stimulate Network Jam and consume CPU heavily to stimulate CPU exhaustion. We utilize Strace [12], which is a diagnostic, debugging and instructional userspace utility for Linux, to delay write or read system call of the corresponding service instance to stimulate the IO read or write busy. Each fault injection in our experiment lasts for 3 minutes. In order to reduce the mutual influence of different injection operations, we make the interval between injection operations greater than 6 minutes. According to Occam's razor theory [13], a complex situation that has more that two root causes simultaneously is of low probability. Therefore, we inject faults with two simultaneous root causes at most. Fig. 7 shows some fault injection examples and *MicroRank*'s results in our experiment. We did 100 injection operations in all experiments and injected 150 faults to Hipster-shop. Fig. 8(a) and Fig. 8(b) show the distribution of the four types of faults among different services on $\mathcal{A}$ and $\mathcal{B}$ respectively.

5.1.2 ***Real-world Microservice System.*** Dataset C is released by 2020 AIOps Challenge Event [14]. C is based on a real-world production microservice system in China Mobile Zhejiang. In particular, the workload of the system in C is a replica of the real-world workload. Note that since this event does not only focus on microservice application, in this paper we only selected those faults related to microservices on May 31st, 2020. Because C does not have the information about operations, we assume each service instance has only an operation.

5.2 Spectrum formulae and definitions

To compare the difference of effectiveness among spectrum formulae, we selected eight formulae in total. Table 3 presents the

[10]Kubernetes, https://kubernetes.io
[11]Chaosblade, https://github.com/chaosblade-io/chaosblade
[12]Strace, https://strace.io
[13]Occam's razor, https://en.wikipedia.org/wiki/Occam%27s_razor
[14]C dataset, http://iops.ai/competition_detail/?competition_id=15&flag=1

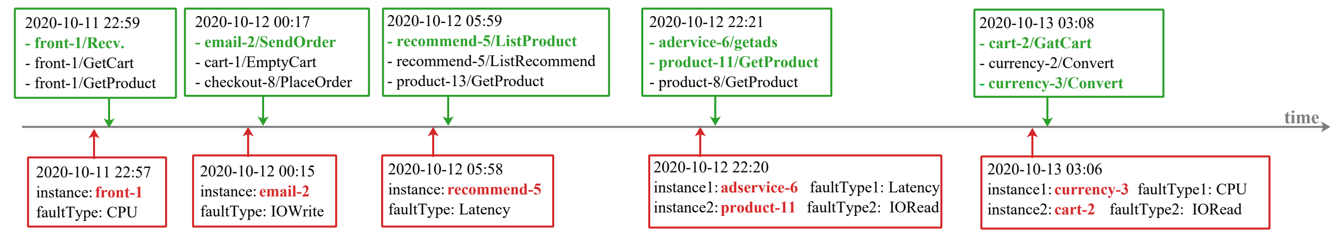

Figure 7: Examples of fault injection and root causes localization on Hipster-Shop

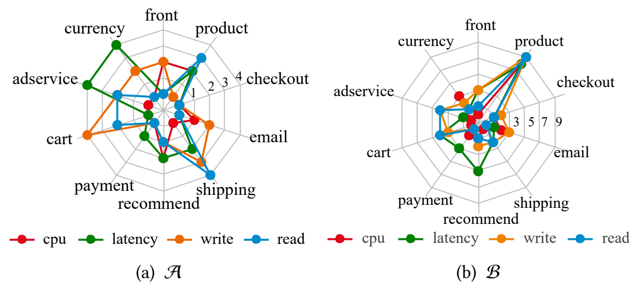

Figure 8: The distribution of faults on 𝒜 and ℬ

Table 3: Spectrum formulae employed in the experiment

Formula	Definition	Formula	Definition
Dstar2	$\frac{O_{ef}^2}{O_{ep}+O_{nf}}$	Ochiai	$\frac{O_{ef}}{\sqrt{(O_{ef}+O_{ep})(O_{ef}+O_{nf})}}$
Goodman	$\frac{2O_{ef}^2-O_{nf}-O_{ep}}{2O_{ef}+O_{nf}+O_{ep}}$	Sørensen	$\frac{2O_{ef}}{2O_{ef}+O_{nf}+O_{ep}}$
Jaccard	$\frac{O_{ef}}{O_{ef}+O_{nf}+O_{ep}}$	RussellRao	$\frac{O_{ef}}{O_{ef}+O_{nf}+O_{ep}+O_{np}}$
M2	$\frac{O_{ef}}{O_{ef}+O_{np}+2O_{ep}+2O_{nf}}$	Dice	$\frac{2O_{ef}}{O_{ef}+O_{ep}+O_{nf}}$

definitions of all formulae used in our evaluation. To be specific, we use the Ochiai, commonly used in recent studies on fault localization [27], as the default spectrum formula in our experiments.

5.3 Evaluation Metric

To evaluate the effectiveness of *MicroRank*, we employed two metrics that are usually employed by exiting studies.

Recall of Top-k (*R@k*) refers to the probability that root causes can be located within the top k service instances among all candidates [12, 19, 41].Higher $R@k$ denotes more effective root cause localization. In a survey conducted by Kochhar et.al.[16], more than 73% developers only consider Top-5 ranked elements. Therefore, in this paper we split the overall results into $R@k(k = 1, 3, 5)$ in single cause experiments and $R@k(k = 2, 3, 5)$ in two-causes experiments. Note that for $R@k$, the higher the better.

EXAM Score refers to measure the mean count of false-positive candidates that have to be excluded manually by operators before locating all the root causes [12, 27, 35]. If the root cause is out of Top-5, we set a default false-positive candidates of 10 for it. Note that for ES, the smaller the better.

5.4 Experiments and Results Analysis

5.4.1 ***MicroRank's overall effectiveness*** . We present the experimental results of *MicroRank* on three datasets $\mathcal{A}$, $\mathcal{B}$ and C. Table 4, Table 5 and Table 6 present the overall results of $\mathcal{A}$, $\mathcal{B}$ and C ,respectively. In those tables, different columns present different effectiveness metrics, namely Column PR represents the *PageRank Anomalous Scorer* in *MicroRank*, Column SP represents the traditional spectrum-based techniques and Column MR represents our *MicroRank*. Different rows present the spectrum formulae used.

From these tables, we have the following observations. First, *MicroRank* can significantly raise the operators' efficiency of finding root causes no matter in single fault or two faults situation. Overall $R@1$ results on single fault datasets $\mathcal{A}$ and C are larger than 88%. $R@3$ results on the two faults dataset $\mathcal{B}$ are larger than 60%. The decline of recall in $\mathcal{B}$ because *MicroRank* should include both two faults in the top ranking list. We also find that the probability that *MicroRank* locates one of the two faults on dataset $\mathcal{B}$ exceeds 90%.

Second, despite the fact that various techniques perform differently in situations with different number of root causes. In most cases, *MicroRank* is able to boost the initial spectrum-based techniques on three datasets. For instance, *MicroRank* that employs RussellRao spectrum measurement enhances the $R@3$ from 50% to 74% on the dataset $\mathcal{B}$ and the $R@1$ from 86% to 92% on the dataset $\mathcal{A}$.

Third, from the perspective of the spectrum formulae, Ochiai (marked in gray) is the most effective formulae in our experiments. In addition, we discover that *MicroRank* is less affected by different formulae. For instance, on $\mathcal{A}$, the difference between the best $R@1$ and the worst $R@1$ is less than 6%.

Finally, we find that both the *MicroRank* and initial spectrum-based method outperform the PageRank techniques. It means that the root causes not always get the highest anomalous score in *PageRank Anomalous Scorer*. The reason is that the microservice system may have some normal service instances covered by each trace. To be specific, we find that the *PageRank Anomalous Scorer* hardly ranks the root cause at Top-1 on C. When we visualize the traces of C, we find that it only has a simple service dependency graph, which contains 13 service instances and it has only one kind of trace. Such simple topology and a single kind of trace limit the capability of PageRank. However *MicroRank* still performs excellently and not worse than the initial spectrum-based techniques even when in a simple microservice system. For instance, on the dataset C, each request needs to query data from db_003, db_007 and db_009. Hence, when injecting a fault to service instance $docker\_008$, each anomalous trace covers db_003, db_007, db_009 and $docker\_008$. The output of Top-4 anomalous ranking list of *PageRank Anomalous Scorer* is $\{db\_003, db\_007, db\_009, docker\_008\}$. While the Top-3 normal ranking list is $\{db\_003, db\_007, db\_009\}$ because every normal trace covers them too. Therefore, the anomalous score of

Table 4: Latency issues localization results on dataset $\mathcal{A}$

DataSet	Formula	R@1			R@3			R@5			Exam Score				
		PR	SP	MR	PR	SP	MR	PR	SP	MR	PR	SP	MR	IMP *	IMP †
$\mathcal{A}$	Dstar2	78	88	94	78	94	96	92	96	96	1.16	0.56	0.42	63.79%	25.00%
	Goodman	78	86	88	86	92	92	92	92	92	1.16	0.88	0.84	27.59%	4.45%
	Jaccard	78	86	88	86	92	92	90	92	92	1.28	0.88	0.84	34.38%	4.45%
	M2	78	90	94	86	94	96	92	94	96	1.16	0.60	0.42	63.79%	30.00%
	Ochiai	78	88	94	86	94	96	92	96	96	1.16	0.56	0.42	63.79%	25.00%
	Sørensen	78	86	88	86	90	92	92	92	92	1.16	0.88	0.84	34.38%	4.45%
	RussellRao	78	86	92	86	92	96	92	96	96	1.28	0.64	0.44	65.63%	31.25%
	Dice	78	86	88	86	92	92	90	92	92	1.32	0.88	0.84	36.36%	4.45%

* denotes the improvement between PR and MR † denotes the improvement between SP and MR

Table 5: Latency issues localization results on dataset $\mathcal{B}$

DataSet	Formula	R@2			R@3			R@5			Exam Score				
		PR	SP	MR	PR	SP	MR	PR	SP	MR	PR	SP	MR	IMP *	IMP †
$\mathcal{B}$	Dstar2	30	52	54	36	62	68	46	70	72	2.46	1.50	1.38	43.90%	8.00%
	Goodman	32	56	58	38	62	66	48	76	84	2.4	1.46	1.18	39.17%	19.18%
	Jaccard	28	54	56	34	58	64	44	74	76	2.54	1.54	1.28	39.37%	16.88%
	M2	28	50	64	34	56	72	44	64	82	2.56	1.68	1.04	59.36%	38.10%
	Ochiai	30	60	66	36	68	76	46	82	84	2.48	1.14	0.94	62.10%	17.54%
	Sørensen	30	56	58	36	62	66	46	78	84	2.48	1.38	1.20	51.61%	13.04%
	RussellRao	30	42	64	36	50	74	46	56	84	2.48	1.96	1.00	59.68%	48.98%
	Dice	26	56	58	32	62	66	44	76	84	2.6	1.40	1.16	55.38%	17.14%

* denotes the improvement between PR and MR † denotes the improvement between SP and MR

Table 6: Latency issues localization results on dataset C

DataSet	Formula	R@1			R@3			R@5			Exam Score				
		PR	SP	MR	PR	SP	MR	PR	SP	MR	PR	SP	MR	IMP*	IMP†
C	Dstar2	0	85	100	0	100	100	100	100	100	3.42	0.14	0.00	100.00%	100.00%
	Goodman	0	85	100	0	100	100	100	100	100	3.42	0.14	0.00	100.00%	100.00%
	Jaccard	0	85	100	0	100	100	100	100	100	3.42	0.14	0.00	100.00%	100.00%
	M2	0	100	100	0	100	100	100	100	100	3.42	0.00	0.00	100.00%	0.00%
	Ochiai	0	85	100	0	100	100	100	100	100	3.42	0.14	0.00	100.00%	100.00%
	Sørensen	0	85	100	0	100	100	100	100	100	3.42	0.14	0.00	100.00%	100.00%
	RussellRao	0	71	100	0	85	100	100	100	100	3.42	0.71	0.00	100.00%	100.00%
	Dice	0	85	100	0	100	100	100	100	100	3.42	0.14	0.00	100.00%	100.00%

* denotes the improvement between PR and MR † denotes the improvement between SP and MR

*db*_003, *db*_007 and *db*_009 can be balanced by their normal scores in Equation(8) and the anomalous score of *docker*_008 remains its effect. Thus, *MicroRank* can still rank *docker*_008 at first. In other words, our method can eliminate the interference of the service instances that are covered by requests every time.

5.4.2 *MicroRank's Anomaly Detector module effectiveness.*

Figure. 9 presents the anomaly detection results of *Anomaly Detector* in *MicroRank* on dataset $\mathcal{A}$ and $\mathcal{B}$. We do not apply *Anomaly Detector* to C because we do not have the complete fault list in that production system. The results show that *Anomaly Detector* performs better on the dataset $\mathcal{B}$ than $\mathcal{A}$. This is because injecting two faults at the same time can make the anomaly more obvious. The *Precision* results of two datasets are slightly lower than the *Recall*. It means that the number of misreport (i.e., the normal trace was determined as abnormal trace) is larger than underreport (i.e., anomaly was not detected). It is acceptable for application operators because missing an anomaly has more serious consequences than false positives.

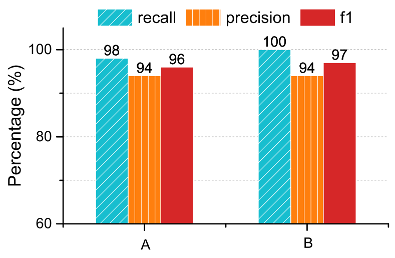

Figure 9: The anomaly detection results on $\mathcal{A}$ and $\mathcal{B}$

5.4.3 *Impacts of configuration.*

In this section, we extend our experiments with different configurations to investigate the influence of damping factors d in Equation(4), call graph weight ω in Equation(6), and preference vector weight φ in Equation(7) on dataset $\mathcal{A}$. Because the Exam Score reflects the overall performance of *MicroRank*, we select the Exam Score to show the effectiveness of *MicroRank* in this subsection.

Damping factor d in Equation(4). Figure. 10 presents the impacts of different d on the effectiveness of *MicroRank*. In this figure, the x axis presents various d values, while the y axis presents the Exam Score. From the figure, we can find that the d does not impact the *MicroRank* effectiveness much. For example, for all the formulae, the largest improvement difference among different damping factors is only 0.2. In addition, for the majority cases, when the d increases, the Exam Score slightly increase. The reason is that when d increases, the trace scope and trace kind would make fewer contributions in localizing the faults.

Call graph weight ω in Equation(6). Figure. 11 presents the impacts of different ω on the effectiveness of *MicroRank*. The x axis presents various ω values, while the y axis presents the Exam Score. In this study, ω is used to introduce call graph to differentiate operations covered by the same anomalous and normal traces. The figure shows that the Exam Score of *MicroRank* dramatically decrease at the very beginning, but then remain stable later. We think the reason to be that when ω is 0, *MicroRank* only use only the trace coverage information and cannot differentiate operations with the same trace coverage, hence the Exam Score is relatively low.

Preference vector weight φ in Equation(7). Figure. 12 presents the impacts of different φ on the effectiveness of *MicroRank*. The x axis presents various φ values, while the y axis presents the Exam Score. In this study, φ is used to balance the importance between trace scope and trace kind. The figure shows that the lines of Exam Score are concave. They dramatically decrease at the very beginning, and gradually increase in the end. We found the reason to be that when φ is 1, the preference vector only considers the trace scope. However, when φ equals 0, the preference vector only takes account of the information about trace kind. Considering only the trace scope or kind of trace can result in a lower accuracy.

5.4.4 *Impact of trace number*.

Figure. 13 shows how the number of traces in one sliding window impacts the effectiveness of *MicroRank*. The x axis presents various the number of traces in the sliding window, while the y axis presents the Exam Score. From the Fig. 13, we can see that Exam Score dramatically decrease at the very beginning, but then slowly decrease after the number of traces is larger than 100. This observation says, our approach does not require lot of traces to localize root causes. Our system is able to locate the root causes in the case of losing partial tracing data.

5.4.5 *Comparisons with state-of-art methods*.

To validate the effectiveness of *MicroRank*, we compare it with several state-of-the-art root cause analysis (RCA) methods including FChain [25], NetMedic [15], Sieve [33], CauseInfer [5], Roots [9], Microscope[19] and AutoMap[22]. To compare with FChain, we first identify the abnormal changes in the latency data of each service, then infer root causes along the service dependency graph with the methods proposed by FChain. To compare with NetMedic, we leverage NetMedic's state correlation approach to estimate service dependencies. Then we use the method presented by NetMedic to locate root causes. To compare with Sieve, we adopt Sysdig [15] to obtain the static service call graph, namely a bi-directed graph then leverage Granger Causality to obtain the dynamic service dependencies with response time metrics. Based on this graph, we identify services that contribute the most changes in the dependency graph when an anomaly occurs. To compare with Roots, we implement the four root cause identification approaches mentioned in Roots. To compare with Microscope, we leverage the method presented in Microscope to locate root causes. To compare with CauseInfer, we capture network packets and leverage lag correlation to find service dependencies. Then, we leverage a Depth First Search based traversal approach to infer root causes. To compare with AutoMap, we build causality graphs under normal and abnormal cases and locate root causes with a random walk approach on the graphs.

Figure. 14 shows the comparison result of $R@1$ among different RCA methods based on dataset $\mathcal{A}$. From this figure, we observe that *MicroRank* achieves a better result than other RCA methods. *MicroRank* achieves 94% in $R@1$, which is at least 6% higher than other methods. *MicroRank* outperforms the latest work, AutoMap, by 6% higher in $R@1$ with a lower cost. Compared with NetMedic, we have over 16% improvement in $R@1$. Fig. 15 compares the performance of different RCA algorithms based on dataset $\mathcal{B}$. We can observe that *MicroRank* outperforms other algorithms on dataset $\mathcal{B}$. We argue that the reasons why *MicroRank* achieves the best performance are two-fold.

- *MicroRank* leverages the detailed end-to-end tracing data to construct the service call graph which is much more accurate than statistic-based approaches like Fchain [25], NetMedic [15], Sieve [33], CauseInfer [5], and AutoMap[22].
- *MicroRank* weave the clues provided by normal and abnormal traces to pinpoint root cause service instances, which is better than those approaches using abnormal traces exclusively like Roots [9] and Microscope[19].

5.5 Discussion

Table 7: Overhead statistics on benchmark experiment.

System Module	Overhead
Instrumented client	2% ± 1% CPU utilization(Single core)
Anomaly Detector	8% ± 4% CPU utilization(Single core)
Data Preparator	60% ± 5% CPU utilization(Single core)
PageRank Scorer	20% ± 5% CPU utilization(Single core)
Spectrum Scorer	2% ± 1% CPU utilization(Single core)
Anomaly Detector	0.8 second (Single physical node)
Data Preparator	1.5 second (Single physical node)
PageRank Scorer	5.5 second (Single physical node)
Spectrum Scorer	0.1 seconds (Single physical node)

Overhead. Table 7 shows the overhead of *MicroRank*. The *Anomaly Detctor* takes about 8% CPU utilization and 0.8 second to count the operation number and distinguish whether the traces are abnormal. When the root cause analysis is triggered, *Data Preparator* needs about 60% CPU utilization and 1.5 seconds to mine the detailed trace information. Given the large volume of tracing data in production systems, the *Data Preparator* module can be accelerated by the MapReduce paradigm. Then, *MicroRank* computes PageRank scores and a spectrum score, which consumes 20% CPU utilization in total. The process consumes approximately 5.5 seconds to calculate PageRank scores and 0.1 second to compute the final score.

Limitation. First, *MicroRank* focuses on the latency issues localization, so it cannot localize faults that would not cause request

[15]Sysdig, https://sysdig.com/opensource/

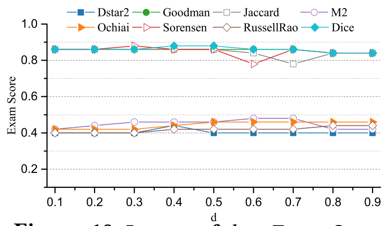

Figure 10: Impact of d on Exam Score

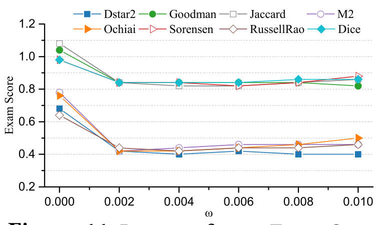

Figure 11: Impact of ω on Exam Score

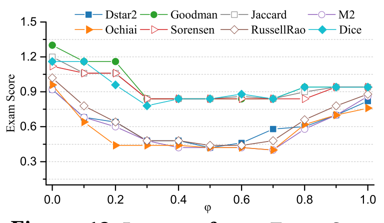

Figure 12: Impact of φ on Exam Score

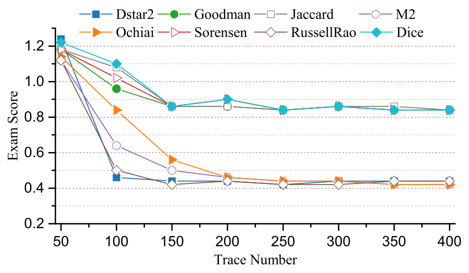

Figure 13: Impact of trace number on Exam Score

latency deviations. Second, *MicroRank* relies on the quality of tracing data. Our system is able to locate root causes in the case of losing some tracing data, but for the platform that losing too many tracing data, the accuracy will be affected inescapably. Moreover, instrumenting tracing codes causes additional overhead, but the tracing data can be utilized not only by *MicroRank* but also other auxiliary procedures like the predictive anomaly module. In addition, our approach is a reactive approach. That means it takes actions only after a latency issue occurs, which is later than the proactive methods.

6 RELATED WORK

Tracing Based Work. Some existing studies provided various solutions on end-to-end tracing [2, 4, 7, 14]. They proposed tracing systems to instruct the source code, collect tracing data, and find out the requests with the long response time. As it is described in Section 3.2, these systems did not fully leverage the tracing data to locate the root causes. Efforts on analyzing tracing data mainly fell into pinpointing performance problems by comparing request flows [6, 29]. Pip [28] leverages developer-provided, component-based expectation of the architecture and latency behavior to compare with actual behavior observed in end-to-end traces. However, Pip excessively relies on developers' specification, which does not work well in a dynamic micorservice environment.

Dependency Graph Based Work. Approaches in this category always build a dependency graph of nodes in distributed systems indicating the relationship of each node before diagnosing or analyzing. Besides, these works commonly utilize monitoring metrics to detect the anomaly and build causality graphs with different approaches, then diagnosing problems with the combination of metrics and dependency graph. CauseInfer [5] constructs a two-layered hierarchical causality graph of the distributed system and infers the root causes of performance problems along the graph by statistical methods. Sieve [33] infers performance metric dependencies between distributed components in the system with the Granger Causality test. Microscope [19] intercepts system calls network sockets to obtain network dependency and builds the dependency graph with network information. For RCA, Microscope finds root cause candidates by comparing the similarity between SLO metrics and the abnormal services. This category of work enjoy the benefit that the source code doesn't need to be instrumented, yet they are not able to provide fine-grained data for analysis compared to trace based work.

Machine Learning Based Work. The machine learning based methods including classification, relevancy analysis and etc. Liu et.al., [20] proposed an approach with a spatial-temporal feature extraction scheme built on the concept of symbolic dynamics to discover causal interactions. Then a Restricted Boltzmann Machine (RBM) is used to detect anomalies and the root causes are obtained by analyzing anomaly propagation. Liu et.al., [21] also introduced a sequential state switching model based on RBM and artificial anomaly association based on deep neural networks to pinpoint the root causes. Li et.al., [17] applied a dynamic latent variable model and dynamic time warping based causality analysis to locate root causes. Compared to our work, the machine learning based methods need more data to adapt to the dynamic microservice systems.

7 CONCLUSION AND FUTURE WORK

This paper designs and implements, *MicroRank*, a novel system to locate root causes that lead to latency issues in microservice environments. *MicroRank* extracts service latency from tracing data then conducts the anomaly detection procedure. By combining PageRank and spectrum analysis, the service instances that lead to latency issues are ranked with high scores. The experimental evaluations in a widely-used open-source system and a production system show that *MicroRank* can localize root causes accurately, which outperforms some state-of-the-art approaches. Moreover, *MicroRank* is relatively lightweight and can scale out readily in large-scale microservice systems. As part of future works, we plan to conduct experiments in benchmarks with more services and more complex dependencies.

ACKNOWLEDGMENTS

The research is supported by the Key-Area Research and Development Program of Guangdong Province (No. 2020B010165002), the National Natural Science Foundation of China (No. 61802448, No. U1811462), the Basic and Applied Basic Research of Guangzhou (No. 202002030328), and the Natural Science Foundation of Guangdong Province (No. 2019A1515012229). The corresponding author is Pengfei Chen.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference*

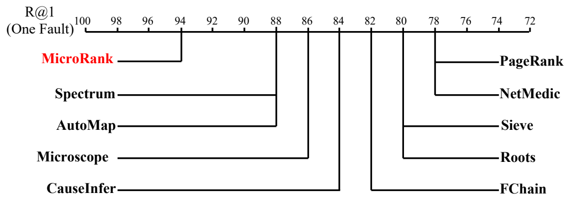

Figure 14: $R@1$ results among different RCA methods on $\mathcal{A}$

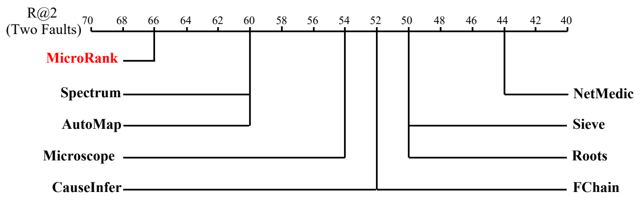

Figure 15: $R@2$ results among different RCA methods on $\mathcal{B}$

Practice and Research Techniques-MUTATION. IEEE, 89–98.

[2] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using magpie for request extraction and workload modelling. In *6th Symposium on Operating System Design and Implementation*. USENIX Association, 259–272.

[3] Salman Abdul Baset. 2012. Cloud SLAs: present and future. *ACM SIGOPS Operation Systems Review* 46, 2 (2012), 57–66.

[4] Mike Y. Chen, Anthony J. Accardi, Emre Kiciman, David A. Patterson, Armando Fox, and Eric A. Brewer. 2004. Path-based failure and evolution management. In *1st Symposium on Networked Systems Design and Implementation*. USENIX, 309–322.

[5] Pengfei Chen, Yong Qi, Pengfei Zheng, and Di Hou. 2014. CauseInfer: automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. In *2014 Conference on Computer Communications*. IEEE, 1887–1895.

[6] Francois Doray and Michel Dagenais. 2017. Diagnosing performance variations by comparing multi-Level execution traces. *IEEE Transactions on Parallel & Distributed Systems* 28, 2 (2017), 462–474.

[7] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. 2007. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*. 20–20.

[8] Yu Gan, Yanqi Zhang, and Dailun Cheng et al. 2019. An open-Source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[9] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. 2017. Performance monitoring and root cause analysis for cloud-hosted web applications. In *Proceedings of the 26th International Conference on World Wide Web*. 469–478.

[10] Glen Jeh and Jennifer Widom. 2003. Scaling personalized web search. In *Proceedings of the Twelfth International World Wide Web Conference*. ACM, 271–279.

[11] Jiajun Jiang, Ran Wang, Yingfei Xiong, Xiangping Chen, and Lu Zhang. 2019. Combining spectrum-based fault localization and statistical debugging: an empirical study. In *34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE / ACM, 502–514.

[12] Jiajun Jiang, Ran Wang, Yingfei Xiong, Xiangping Chen, and Lu Zhang. 2019. Combining spectrum-based fault localization and statistical debugging: an empirical study. In *34th IEEE/ACM International Conference on Automated Software Engineering*. 502–514.

[13] James A. Jones, Mary Jean Harrold, and John T. Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*. ACM, 467–477.

[14] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. 2017. Canopy: an end-to-end performance tracing and analysis system. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*. 34–50.

[15] Srikanth Kandula, Ratul Mahajan, and et al. Verkaik. 2009. Detailed diagnosis in enterprise networks. *ACM SIGCOMM Computer Communication Review* 39, 4 (2009), 243–254.

[16] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 165–176.

[17] Gang Li, Tao Yuan, S Joe Qin, and Tianyou Chai. 2015. Dynamic time warping based causality analysis for root-cause diagnosis of nonstationary fault processes. *IFAC-PapersOnLine* 48, 8 (2015), 1288–1293.

[18] Xing Li, Yan Chen, and Zhiqiang Lin. 2019. Towards automated inter-service authorization for microservice applications. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*. ACM, 3–5.

[19] Jinjin Lin, Pengfei Chen, and Zibin Zheng. 2018. Microscope: pinpoint performance issues with causal graphs in micro-service environments. In *16th International Conference on Service-Oriented Computing*. Springer, 3–20.

[20] Chao Liu, Sambuddha Ghosal, Zhanhong Jiang, and Soumik Sarkar. 2016. An unsupervised spatiotemporal graphical modeling approach to anomaly detection in distributed CPS. In *ACM/IEEE 7th International Conference on Cyber-Physical Systems*. 1–10.

[21] Chao Liu, Kin Gwn Lore, and Soumik Sarkar. 2017. Data-driven root-cause analysis for distributed system anomalies. In *IEEE 56th Annual Conference on Decision and Control*. 5745–5750.

[22] Meng Ma, Jingmin Xu, Yuan Wang, Pengfei Chen, Zonghua Zhang, and Ping Wang. 2020. AutoMAP: diagnose your microservice-based web applications automatically. In *The Web Conference 2020*. ACM / IW3C2, 246–258.

[23] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. 2009. 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*. 1–6.

[24] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology* 20, 3 (2011), 11:1–11:32.

[25] Hiep Nguyen, Zhiming Shen, Yongmin Tan, and Xiaohui Gu. 2013. Fchain: toward black-box online fault localization for cloud systems. In *IEEE 33rd International Conference on Distributed Computing Systems*. 21–30.

[26] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[27] Spencer Pearson and José Campos etc. 2017. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE / ACM, 609–620.

[28] Patrick Reynolds, Charles Edwin Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. 2006. Pip: detecting the unexpected in distributed systems.. In *NSDI*, Vol. 6. 9–9.

[29] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. 2011. Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. 43–56.

[30] Huasong Shan and Yuan Chen etc. 2019. Diagnosis: unsupervised and real-time Diagnosis of small-window long-tail latency in large-scale microservice platforms. In *The World Wide Web Conference*. ACM, 3215–3222.

[31] Yuri Shkuro. 2019. *Mastring Distributed Tracing*. Packt.

[32] Benjamin H Sigelman and et al. Barroso. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).

[33] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. 2017. Sieve: actionable insights from monitored metrics in distributed systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 14–27.

[34] Ping Wang, Jingmin Xu, Meng Ma, Weilan Lin, Disheng Pan, Yuan Wang, and Pengfei Chen. 2018. Cloudranger: root cause identification for cloud native systems. In *Proceedings of the 18th cloud-hosted IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 492–502.

[35] W. Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. 2008. A crosstab-based statistical method for effective fault localization. In *First International Conference on Software Testing, Verification, and Validation*. IEEE, 42–51.

[36] Wenpu Xing and Ali A. Ghorbani. 2004. Weighted PageRank algorithm. In *2nd Annual Conference on Communication Networks and Services Research*. IEEE Computer Society, 305–314.

[37] Guangba Yu, Pengfei Chen, and Zibin Zheng. 2019. Microscaler: automatic scaling for microservices with an online learning Approach. In *IEEE International Conference on Web Services*. IEEE, 68–75.

[38] Guangba Yu, Pengfei Chen, and Zibin Zheng. 2020. Microscaler: cost-effective scaling for microservice applications in the cloud with an online learning Approach. *IEEE Transactions on Cloud Computing* (2020). https://doi.org/10.1109/TCC.2020.2985352

[39] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. 2017. Boosting spectrum-based fault localization using PageRank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 261–272.

[40] Hao Zhou and Ming Chen etc. 2018. Overload control for scaling WeChat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 149–161.

[41] Xiang Zhou, Xin Peng, and Tao Xie. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 683–694.