

# Nezha: Interpretable Fine-Grained Root Causes Analysis for Microservices on Multi-modal Observability Data

Guangba Yu  
Pengfei Chen\*  
Sun Yat-sen University  
China

Yufeng Li  
Hongyang Chen  
Sun Yat-sen University  
China

Xiaoyun Li  
Zibin Zheng  
Sun Yat-sen University  
China

## ABSTRACT

Root cause analysis (RCA) in large-scale microservice systems is a critical and challenging task. To understand and localize root causes of unexpected faults, modern observability tools collect and preserve multi-modal observability data, including metrics, traces, and logs. Since system faults may manifest as anomalies in different data sources, existing RCA approaches that rely on single-modal data are constrained in the granularity and interpretability of root causes. In this study, we present *Nezha*, an interpretable and fine-grained RCA approach that pinpoints root causes at the code region and resource type level by incorporative analysis of multi-modal data. *Nezha* transforms heterogeneous multi-modal data into a homogeneous event representation and extracts event patterns by constructing and mining event graphs. The core idea of *Nezha* is to compare event patterns in the fault-free phase with those in the fault-suffering phase to localize root causes in an interpretable way. Practical implementation and experimental evaluations on two microservice applications show that *Nezha* achieves a high top1 accuracy (89.77%) on average at the code region and resource type level and outperforms state-of-the-art approaches by a large margin. Two ablation studies further confirm the contributions of incorporating multi-modal data.

## CCS CONCEPTS

• **Software and its engineering** → **Software reliability**; **Software performance**; **Cloud computing**.

## KEYWORDS

Root Cause Analysis, Multi-modal Observability Data, Microservice

### ACM Reference Format:

Guangba Yu, Pengfei Chen, Yufeng Li, Hongyang Chen, Xiaoyun Li, and Zibin Zheng. 2023. Nezha: Interpretable Fine-Grained Root Causes Analysis for Microservices on Multi-modal Observability Data. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23)*, December 3–9, 2023, San Francisco, CA, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3611643.3616249>

\*Pengfei Chen is the corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0327-0/23/12...\$15.00

<https://doi.org/10.1145/3611643.3616249>

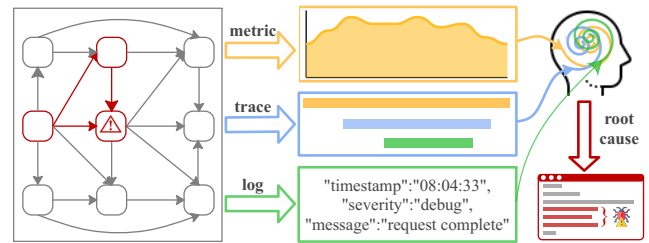


Figure 1: Multi-modal observability data (i.e., metrics, traces, and logs) are used to localize fine-grained root causes (e.g., regions of code defects or resource types).

## 1 INTRODUCTION

Microservice architecture decomposes an application into many small pieces of services, allowing developers to modify and redeploy specific services rather than the entire application [23, 70]. However, interconnected services are deployed across multiple hosts and the number of services in production increases rapidly as the business develops. Many factors (e.g., dynamic container environment and complex service invocations) can affect the performance and availability of microservice applications, leading to more performance and availability issues [68, 71].

To help Site Reliability Engineers (SREs) better understand the performance and availability of applications, observability is proposed with a capability of comprehensive visibility into distributed applications [59]. The primary data used in observability are metrics, logs, and traces, which are referred to as “multi-modal observability data” [59]. As shown in Fig. 1, when a fault occurs, SREs typically need to combine clues extracted from multi-modal observability data to localize the root cause, which is the most fundamental reason for the fault [63].

Manually troubleshooting root causes based on multi-modal data is time-consuming and error-prone in complex microservice applications due to the explosion and heterogeneity of the observability data. Over the years, many metrics-based approaches [6, 26, 35, 62, 64, 65], trace-based approaches [8, 9, 13, 22, 32, 67, 68, 71], and log-based approaches [3, 20, 36, 45, 52, 56] have been devoted to automatically localize root causes. However, we identify three primary limitations of existing root cause analysis (RCA) approaches.

- (1) **Insufficient exploitation on multi-modal data.** Most of the state-of-the-art RCA approaches [3, 32, 35, 56, 65, 68] only use single-modal data to pinpoint root causes. Since system failures may manifest as anomalies in different modal data, adopting single modal data could lose key RCA clues and thus impact the accuracy of RCA approaches (details in § 2.2).
- (2) **Coarse-grained root causes.** Existing single-modal studies mainly localize the root causes of microservice applications

at the service level [3, 6, 35, 52, 65, 67]. Such coarse-grained root causes require SREs to manually troubleshoot the true root causes (e.g., specific code region or resource type) of the faulty services, which delays the failure recovery process.

- (3) **Weak interpretability of root causes.** Existing RCA techniques place insufficient emphasis on the interpretability of root causes, which is crucial for SREs to fully understand the underlying issues. For example, PDiagnose [21] takes multi-modal data as input by transforming heterogeneous logs and traces into time series. However, this approach sacrifices the detailed execution context of requests (e.g., log contents and trace paths). As a result, it fails to provide detailed explanations regarding where and why the faulty requests occurred.

**Nezha Approach.** To overcome the above limitations, we propose *Nezha*, which is a protection deity with multiple arms in Chinese mythology, an interpretable and fine-grained RCA framework by incorporative analysis of multi-modal data. When Anomaly Detector (§ 4.1) detects an anomaly on front-end service requests, Data Integrator (§ 4.2) takes heterogeneous metrics, traces, and logs from the fault-free construction and fault-suffering production phase as inputs and transforms them into the homogeneous event representation. Events in the same service instance are then ordered according to their timestamps and event groups across service instances are connected according to span IDs to construct event graphs while overcoming clock synchronization problems. Pattern Miner (§ 4.3) then extracts the common execution patterns of the application from the event graphs.

After transforming multi-modal data into events (i.e., excluding specific request parameters), the event patterns in the construction phase are similar to those in the production phase. During the fault-suffering phase, faults manifest themselves in one or more data sources, which causes some event patterns to deviate from their expected execution paths. Therefore, the critical problem of *Nezha* is to identify (1) *which event patterns deviate from expected execution paths*, and (2) *how these patterns change* in the actual fault-suffering phase. To localize such faulty event patterns, Expected Pattern Ranker (§ 4.4.1) is proposed to solve the problem (1) by pinpointing expected patterns that frequently occur in the fault-free phase but rarely happen in the fault-suffering phase. The faulty event patterns reflect which code regions or resources type are culprits, which is more fine-grained than the existing RCA approaches of pinpointing faults at the service level (solution of limitation 2).

Actual Pattern Ranker (§ 4.4.2) is used to solve problem (2) by locating the actual patterns that frequently occur in the fault-suffering phase but rarely happen in the fault-free phase. Such newly emerging patterns in the fault-suffering phase facilitate SREs to understand faults. Eventually, Pattern Aggregator (§ 4.5) correlates expected patterns with actual patterns and takes pattern pairs as root cause candidates. *Nezha* outputs actionable root causes with high interpretability by comparing expected patterns with actual patterns (solution of limitation 3).

We conducted extensive studies to evaluate *Nezha* on two popular microservice applications, namely TrainTicket [12] and OnlineBoutique [14]. Benefiting from the use of multi-modal observability data, *Nezha* achieves a high top-1 accuracy (89.77%) and surpasses all compared approaches by a large margin (61.45% ~ 74.63%)

when identifying root causes at the service level. When identifying root causes at the inner-service level (i.e., code region or resource type), *Nezha* outperforms advanced baselines by 67.47% ~ 74.85% in a high top-1 accuracy. Moreover, two ablation studies further confirm the contribution of incorporating multi-modal data.

**Contributions.** This study makes the following contributions,

- We introduce a novel approach to represent heterogeneous multi-modal observability data (i.e., metrics, traces, and logs) in a unified homogeneous event format. This representation enables the construction of event graphs and facilitates the future integrated analysis across multi-modal observability data.
- We present *Nezha*, an interpretable and fine-grained microservice RCA approach. *Nezha* is a statistical method to localize more granular and actionable root causes (i.e., code region or resource type) with high interpretability, which facilitates SREs to take mitigation actions in confidence.
- We implement the prototype of *Nezha* [48] and conduct extensive experiments on two widely-used microservice applications to validate the effectiveness and efficiency of *Nezha*. The results show that *Nezha* outperforms the state-of-the-art RCA approaches at both service and inner-service level.
- We enhance the observability of two widely-used microservice applications OnlineBoutique and Trainticket, and open source them at [46] and [47], which will facilitate the future anomaly detection and RCA research on multi-modal data.

## 2 BACKGROUND AND MOTIVATION

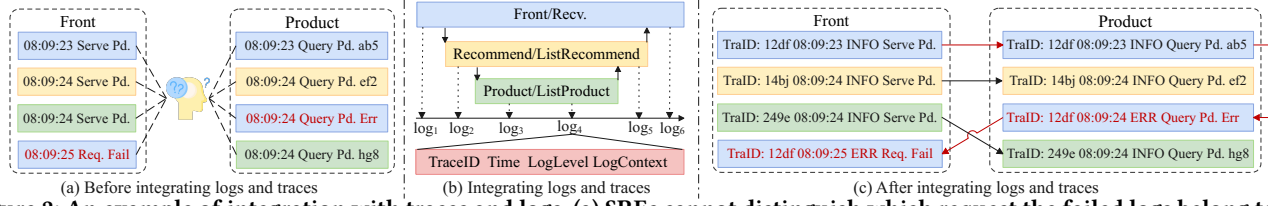
### 2.1 Background

**Observability.** Observability is a measure of how well the internal states of a system can be inferred from knowledge of its external outputs [59]. It has been increasingly applied to microservices because cloud-native environments become more and more complex, and the potential root causes of faults become more challenging to pinpoint. Observability typically uses three types of telemetry data — metrics, traces, and logs — to provide deep visibility into distributed systems. Observability allows SREs to monitor systems more effectively, helping them identify and connect the effects in complex chains and trace them back to their causes.

**Metric.** Metrics are numerical values that describe the status of the microservices and infrastructure over a period of time. Metrics can be further classified as system-level metrics and application-level metrics [27]. System-level metrics such as memory and CPU usage are immediate concerns whenever SREs identify a performance degradation. SREs can identify issues caused by insufficient resources by analyzing system-level metrics. However, system-level metrics cannot help solve problems caused by code defects.

Application-level metrics (e.g., request latency or success ratio) are derived from monitoring requests to describe the application status. Such metrics typically reflect the faults' surface rather than root causes of faults. For example, a decrease in success rate reflects a decrease in availability, and SREs need to further analyze system-level metrics, traces, and logs to identify the root cause. Therefore, we analyze application-level metrics when performing anomaly detection and focus on system-level metrics in RCA.

**Log.** Logs are textual records of what operations are performed during program runtime. A log consists of a timestamp that tells



**Figure 2: An example of integration with traces and logs. (a) SREs cannot distinguish which request the failed logs belong to. (b) The contents of the logs after integration. (c) SREs can track logs belonging to each request after integration.**

when it occurred, with a static structure and free-form text. The log template is the constant part of a log statement in the code [69]. Although logs provide valuable information about the individual service instance, this localized knowledge lacks associations of logs for the same request across different services. Analysis logs on different services independently cannot characterize the behavior of the overall system.

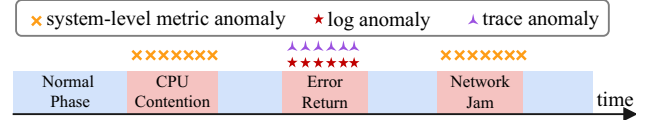
**Trace.** Traces represent the end-to-end paths of requests through a distributed system. As a request moves through a system, every operation performed is called a “span”, which records the caller service, the callee service, and the operation time. Each trace corresponds to a request and has a unique identifier (e.g., trace ID). A span is a named and timed operation that shares the same trace ID within the same trace. Each span also has a unique span ID to signify it. The root span (e.g., *Front/Recv.* in Fig. 2 (b)) is the first span in a trace, while the other spans are correlated by parent-child relationships based on the parent span ID properties (e.g., *Front/Recv.* is the parent of *Recommend/ListRecommend* in Fig. 2 (b)). The contribution of traces is limited in RCA because they only record the coarse-grained service operation-level information.

## 2.2 Motivation

This section presents four motivations with examples from a widely-used microservice application, OnlineBoutique [14]. Details on the application and data collection will be described in § 5.1.

**Motivation 1: Enhancing RCA through Log and Trace Integration.** Logs serve as a valuable resource to capture the internal states and behaviors of individual microservices. However, analyzing logs for each microservice in isolation may result in a loss of global context, leading to ineffective RCA [72]. For an efficient RCA approach, it is essential to combine logs from different microservices. Nonetheless, as illustrated in Fig. 2(a), logs generated by a single microservice can interleave, as the microservice may concurrently serve multiple requests associated with distinct log entries. For a failed request, existent RCA approaches cannot determine which logs spreading across multiple services are associated with the failed request to deduce the root cause.

Traces capture global service interactions across various servers but offer limited insight into local behaviors within individual spans. To capitalize on the benefits of both logs and traces, we propose an integration of traces, representing a coarse-grained global view, and logs, providing a fine-grained local view, to furnish a detailed global perspective for each request. As depicted in Fig. 2(b), we accomplish this integration by inserting trace IDs into log messages, leveraging distributed tracing frameworks such as Opentelemetry [49] and SkyWalking [58]. Inserting trace IDs in log messages is a prevalent practice in numerous industrial systems (e.g., WeChat) and



**Figure 3: The occurrences of related system-level metrics, traces and logs that can reflect anomaly.**

**Table 1: Comparison of state-of-the-art RCA approaches.**

Approach	Metrics	Logs	Traces	RCA Level
MicroScope [35]	✓	✗	✗	Service
MicroRCA [65]	✓	✗	✗	Service
SBLD [56]	✗	✓	✗	Error Log
LogFaultFlagger [1]	✗	✓	✗	Error Log
MicroRank [68]	✗	✗	✓	Service Operation
TraceAnomaly [37]	✗	✗	✓	Service Operation
Dejavu [33]	✓	✗	✓	Fault Type
CloudRCA [74]	✓	✓	✗	Service
PDiagnose [21]	✓	✓	✓	Resource Type or Error Log
Nezha (ours)	✓	✓	✓	Code Region or Resource Type

widely adopted frameworks (e.g., Spring Cloud). By employing this approach, when a request encounters a fault (e.g., the red line in Fig. 2(c)), RCA approaches can track the request-level descriptive information (e.g., logs) to effectively identify the root cause.

**Motivation 2: Enhancing RCA through the Integration of Metrics, Logs, and Traces.** While logs and traces offer abundant clues for RCA, certain anomalies may not be apparent in these sources, leading to a potential loss of critical information pertaining to root causes. Fig. 3 shows three examples of the occurrences of system-level metrics, traces, and logs that can reflect anomalies when CPU contention, error return, and network jam faults were injected into the OnlineBoutique product service. The anomalies in metrics were identified using the  $k$ - $\sigma$  rules, with further details to be presented in § 4.1. Logs and trace spans that exclusively occur during the fault-suffering phase, but not in the fault-free phase, are considered abnormal.

As demonstrated in Fig. 3, log and trace anomalies are absent when CPU contention and network jam faults are injected, because these faults do not alter the execution paths of OnlineBoutique. In these cases, the log-based or trace-based RCA approaches may miss the root causes. Fortunately, system-level metrics can provide valuable insights to locate faults that remain undetected in logs and traces. For instance, anomalies in CPU usage were detected when CPU contention faults were introduced. However, solely relying on metrics-based RCA approaches may not identify faults that are not reflected in metrics (e.g., error return fault in Fig. 3). The examples in Fig. 3 underscore the necessity of integrating metrics with logs and traces to effectively identify root causes.



**Motivation 3: Enhancing RCA through the Integration of Multi-modal Data in a Unified Representation.** As highlighted in Table 1, a significant portion of existing studies primarily focuses on single-modal data for root cause identification. SREs are naturally inclined to merge the results of metrics-based, traces-based, and logs-based methods to accurately pinpoint true root causes. However, this integration process can be labor-intensive and ineffective due to several reasons. (1) When individually analyzing logs, correlation between logs on different machines is not achievable, and a similar issue arises with metrics. Consequently, single-modal techniques are likely to produce less accurate results compared to multi-modal approaches. (2) In situations where SREs employ metrics-based, traces-based, and logs-based approaches to determine root causes, each method may propose a distinct root cause. There is currently no effective strategy to consolidate these outcomes or distinguish the true root cause among them. As a result, SREs are compelled to manually verify each result one by one, which is a labor-intensive task. (3) Independently managing multiple single-modal RCA approaches necessitates increased effort. In light of these challenges, we advocate for the integration of multi-modal data into a unified representation, which will enhance the efficiency and accuracy of RCA.

**Motivation 4: Facilitating Fault Mitigation through Unsupervised Fine-grained RCA with Enhanced Interpretability.** Dejavu [33] and CloudRCA [74] are two supervised multi-modal RCA approaches that necessitate a substantial training dataset with labels. However, acquiring such a dataset can be costly and impractical for each application. The multi-modal method PDiagnose [21] converts heterogeneous multi-modal data into time series and identifies root causes by evaluating abnormal time series. Although this transformation can yield effective results, the process may result in a loss of execution context, which is essential for developers to comprehend the underlying root cause. Preserving the original execution context enhances interpretability in root cause identification, subsequently increasing SREs' confidence in the obtained results. Moreover, localizing root causes at the service level requires a more coarse-grained approach. SREs must expend considerable effort to determine the specific code region or resource accountable for faults. From the SREs' perspective, a fine-grained root cause identification can alleviate their workload and reduce the mean time to mitigation. Therefore, we try to propose an unsupervised fine-grained RCA approach with improved interpretability, which facilitates more effective fault mitigation.

## 2.3 Problem Formulation

We formalize the problem of fine-grained root cause localization using multi-modal observability data. Suppose that a large-scale microservice system with  $N$  microservices, metrics, traces, and logs are aggregated individually at each microservice. In a sliding time window (e.g., 1 minute), we have multi-modal observability data defined as  $\Theta = \left\{ \left( \theta_n^M, \theta_n^T, \theta_n^L \right) \right\}_{n=1}^N$ , where at the  $n$ -th microservice,  $\theta_n^M = \{M_1, \dots, M_m\}$  indicates  $m$  metrics,  $\theta_n^T = \{T_1, \dots, T_t\}$  denotes  $t$  traces, and  $\theta_n^L = \{L_1, \dots, L_\ell\}$  represents  $\ell$  logs.

Given the data  $\Theta_C$  collected from fault-free construction phase  $C$  and  $\Theta_P$  collected from fault-suffering production phase  $P$ , we

first unify the multi-modal data in  $\Theta_C$  and  $\Theta_P$  as events and extract event patterns  $P = \{p_1, \dots, p_k\}$  via constructing and mining event graphs. We attempt to identify fine-grained root causes through three phases: 1) we identify which event patterns do not follow expected execution paths and rank them into expected pattern list  $List_E$ , where  $List_E = \{\dots, (p_i, Score_E(p_i)), \dots\}$ ; 2) we pinpoint how expected patterns change in the actual fault-suffering phase and rank them into actual pattern list  $List_A$ , where  $List_A = \{\dots, (p_j, Score_A(p_j)), \dots\}$ ; 3) we correlate expected patterns with actual patterns and takes pattern pairs as final root cause list  $List_R = \{\dots, (p_i, p_j, Score_E(p_i)), \dots\}$ . SREs can check why  $p_i$  turn into  $p_j$  to determine the final solution to recover applications.

## 3 OVERVIEW

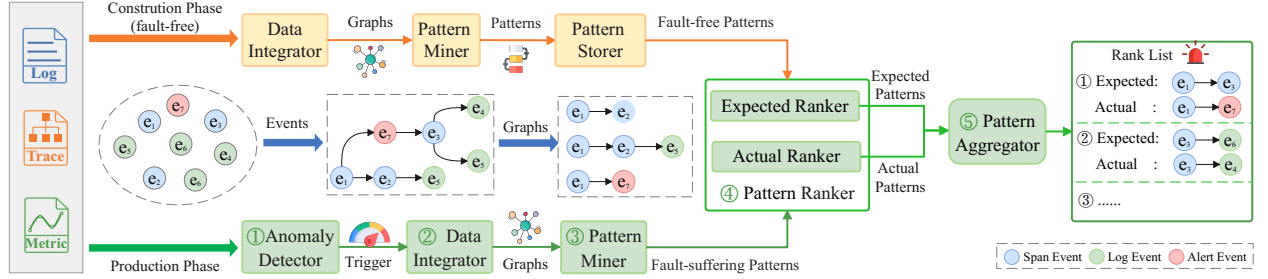
In this study, we present *Nezha*, an unsupervised fine-grained RCA approach by incorporative analysis of multi-modal data in an interpretable manner. Figure 4 shows the overall structure of *Nezha*. The activities of *Nezha* can be divided into the fault-free construction phase and the fault-suffering production phase. In the construction phase, *Nezha* takes the fault-free observability data as input, which contains all request types of the application within a time window. Obtaining a short window of fault-free data is trivial for SREs because most of the time is in normal status and faults are scarce in production environment [30]. We recommend setting the window size to the same interval as the metric collection (1 minute by default in this study). *Nezha* integrates these observability data and mines their patterns offline. Data integration and pattern mining in the construction phase are the same as in the production phase.

In the production phase, ① *Anomaly Detector* (§4.1) detects whether the performance and availability of the system are abnormal in real time. If an anomaly occurs, ② *Data Integrator* (§4.2) unifies the multi-modal data in the abnormal time window into events and consolidates the events into event graphs. ③ *Pattern Miner* (§4.3) extracts common patterns and calculates support for them from the event graphs in parallel. ④ *Expected Ranker* in *Pattern Ranker* (§4.4) ranks the patterns that frequently occur in the fault-free phase but rarely in the fault-suffering phase as the expected patterns. *Actual Ranker* ranks the patterns that frequently occur in the fault-suffering phase but rarely in the fault-free phase as the actual patterns. ⑤ *Pattern Aggregator* (§4.5) aggregates the expected patterns and actual patterns to determine the ranked list of root cause candidates with interpretability. Compared with existent RCA approaches, *Nezha* provides fine-grained root cause candidates (i.e., region of code defect or resource type) and tells SREs why candidates are anomalous. As a result, it is much more convenient and confident for SREs to analyze root causes.

## 4 DETAILED DESIGN

### 4.1 Anomaly Detector

As described in § 2.1, *Nezha* keeps monitoring application-level metrics of systems to detect anomalies. Guaranteeing the user experience, which is typically reflected in availability and performance, is a critical requirement for cloud applications. The availability of systems can be reflected by the request success ratio, which is the fraction of the number of successful requests to total requests over a period of time. The performance issues manifest themselves as



**Figure 4: The overview of Nezha.** *Nezha* uses the multi-modal observability data as input and outputs the ranked list of suspicious fine-grained root causes.

long-time request duration. To capture the tail latency, we use P90 latency, which is the average latency for the slowest 10% of requests over a period of time. Compared to the average latency, P90 can show the tail latency explicitly. As aforementioned, the success ratio and P90 latency delineate the system’s health status. We use *time interval* to denote how often the metrics are collected. The time interval value is set for one minute in this study.

Anomaly Detector uses  $k\text{-}\sigma$  rules [35] to determine whether the target application is in an abnormal status. We calculate the mean  $\mu$  and standard deviation  $\sigma$  of success ratio and P90 latency in the construction phase. In the production phase, Anomaly Detector continually monitors the success ratio and P90 latency of front-end service in a sliding time window. If the success ratio is less than  $\mu - k\sigma$  or P90 latency is greater than  $\mu + k\sigma$  ( $k = 3$  by default), Anomaly Detector declares the current time window is abnormal and triggers a root cause analysis.  $k\text{-}\sigma$  rule is a simple but effective approach and is widely used in academia and industry [35, 68, 75]. In *Nezha*, this module can be easily replaced with other anomaly detection approaches (e.g., USAD [2], RRCF [31] and JumpStarter [43]).

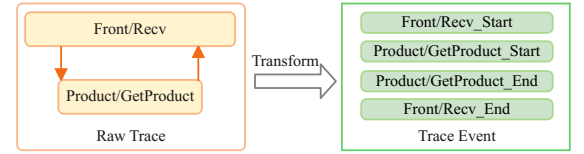
## 4.2 Data Integrator

After Anomaly Detector determines whether the application is under abnormal status, Data Integrator first queries  $\Theta_P$  of the application in the abnormal time window. Considering the heterogeneity of observability data, Data Integrator transforms the observability data in the time windows into events, which is the basic unit of *Nezha*. These events in the same service instance are then ordered according to their timestamps and span IDs while overcoming clock synchronization problems. Each request in the time window corresponds to an event graph.

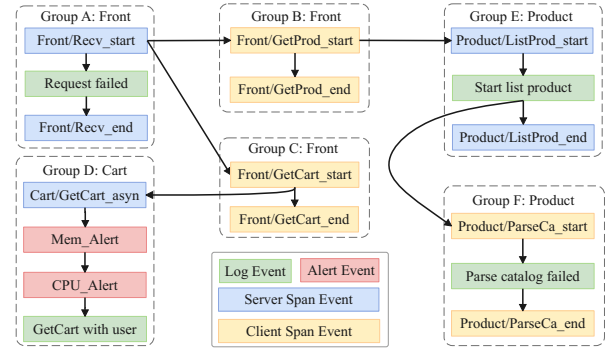
### 4.2.1 Unique Event Generation.

**DEFINITION 1 (EVENT  $e$ ).** An event  $e$  records the execution status of a system at a point in time. We use event set  $E_i = \{e_0, \dots, e_n\}$  to denote the set of all events for request  $i$ .

**Metric.** In each time window, a service has many logs but only one metric sample. To overcome the heterogeneity between numerical metrics and text-structured logs, Data Integrator replaces the set of metrics with suspicious metric alerts. This is reasonable because anomalous metrics that cause alerts provide more information to RCA than normal metrics. Metrics alerts are generated when metric values violate the  $k\text{-}\sigma$  rule or static thresholds. *Nezha* is also compatible with alarm systems such as Prometheus Alertmanager.



**Figure 5: Transformation from a raw trace to trace events.**



**Figure 6: An example of event graph in OnlineBoutique.** All trace events and log events have the same trace ID.

The time between the start and end of an alert is called *alert time*. We treat alerts as events that repeatedly occur within the alert time.

As discussed in § 2.1, application-level metrics typically reflect faults’ symptoms rather than the root causes of faults. Therefore, we only consider the system-level metrics in RCA. In addition, we found that some alerts frequently occur no matter whether there is a fault or not. Such regular alerts are not helpful to RCA and sometimes even mislead SREs. To filter out these irrelevant metric alerts, *Nezha* excludes the alerts in the production phase that also occur in the construction phase.

**Log.** *Nezha* first extracts and records the trace IDs, span IDs (detail shown in § 2.2), and timestamps from raw log messages. Data Integrator adopts a state-of-the-art log parsing approach Drain [19] to extract the static log templates and dynamic log parameters from the raw log messages in a streaming manner. After log parsing, we treat the static log templates as log events. To distinguish the log events associated with different requests, each log event is accompanied by its corresponding trace ID, span ID, and timestamp.

**Trace.** The relation between parent and child span can be divided into synchronous and asynchronous calls. With regard to synchronous calls, we consider the start and end of a span as two trace events. These two event messages can be represented as a concatenation of the span name with “start” or “end” string. Figure 5

shows an example of transforming traces with the synchronous call to trace events. In terms of asynchronous calls, Data Integrator represents them as events consisting of span name and the “asyn” string, e.g.,  $e = \text{“Cart/GetCart\_asyn”}$ . Each trace event is accompanied by its trace ID, span ID, and timestamp from span records.

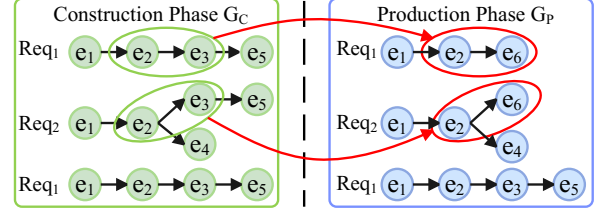
#### 4.2.2 Event Graph Construction.

**DEFINITION 2 (EVENT GRAPH  $g$ ).** An event graph  $g_i = (E_i, \text{Link})$  is a directed graph of events in the event set  $E_i$ . A directed link between  $e_j$  and  $e_{j+1}$  (i.e.,  $e_j \rightarrow e_{j+1}$ ) denotes that  $e_j$  is followed by  $e_{j+1}$  during the execution. We use  $G_C$  and  $G_P$  to denote the set of event graphs for the construction and production phase, respectively.

For each request in a time window, Data Integrator constructs an event graph in the following three steps.

- (1) **Order events in the same span.** For each span, Data Integrator obtains all log and trace events which occurred within that span. Data Integrator then chronologically orders the log and trace events into an event group based on their timestamps and adds a sequence relationship from an event to its next event in the group. Fig. 6 shows some examples of event groups.
- (2) **Insert metric events to event groups.** Data Integrator inserts the alert events after the first event of the event group if the corresponding service has alert events without loss of generality. It can also be inserted in other fixed locations as agreed. If multiple alert events of the same service are detected, all alarm events will be sequentially inserted after the first event. For instance, Data Integrator inserts *Mem\_Alert* and *CPU\_Alert* of cart service into its event group in Fig. 6.
- (3) **Insert child groups to parent groups.** If the child span is in the same service instance as the parent span (i.e., internal function calls), Data Integrator inserts child groups after the last event in parent groups whose timestamp is less than the first event in the child group (e.g., the relation between Group *E* and *F* in Fig. 6). We use timestamps directly because these two groups are on the same service instance and on the same node, so they do not have the problem of clock drift. For RPC call spans across services, Data Integrator inserts the child group after the first event of its parent group based on the parent span ID to overcome the clock drift (e.g., the relation between Group *B* and *E* in Fig. 6).

After these steps, we represent the heterogeneous multi-modal data as homogeneous events and construct the relationships among events as graphs. Though DeepTraLog [72], which focuses on anomaly detection rather than RCA, integrates logs and traces into graphs, it does not consider metric clues. *Nezha* overcomes the shortcoming of DeepTraLog by innovatively transforming metrics into events and incorporating them into the event graph. The reason why we use graphs rather than sequences like Minesweeper [44] to represent the relationships between events is that microservice applications may contain asynchronous calls. The event location of asynchronous calls may change uncertainly in the sequential sequences, making it difficult for *Nezha* to mine for stable patterns. Compared with PDiagnose [18] that transforms heterogeneous multi-modal data as time series, *Nezha* retains the execution context of requests when transforming multi-modal data as homogeneous events, which allows for improved interpretability in RCA.



**Figure 7: Examples of event graphs in construction and production phase.**  $Req_i$  denotes the  $i$ th kind of request (e.g., login or query product). The pattern  $e_2 \rightarrow e_3$  in  $G_C$  turned into  $e_2 \rightarrow e_6$  in  $G_P$  when a fault occurred.

**Table 2: Example of *Nezha* to perform RCA from Fig. 7.**

Pattern	Support		Score		Depth	Rank( $Score_E$ )
	$S_C$	$S_P$	$Score_E$	$Score_P$		
$e_1 \rightarrow e_2$	3	3	0.5	0.5	1	2
$e_2 \rightarrow e_3$	3	1	<b>0.75</b>	0.25	2	1
$e_2 \rightarrow e_4$	1	1	0.5	0.5	2	-
$e_3 \rightarrow e_5$	3	1	<b>0.75</b>	0.25	3	-
$e_2 \rightarrow e_6$	0	2	0.0	1	2	3

“-” means that the pattern is aggregated by *Nezha*.

#### 4.3 Pattern Miner

**DEFINITION 3 (PATTERN  $p$ ).** A pattern  $p$  is a subgraph of contiguous events in the set of event graphs  $G$ .

After integrating multi-modal data into event graphs, *Nezha* extracts the fault-free and fault-suffering patterns from  $G_C$  and  $G_P$  by traversing all event graphs in parallel. The patterns in the event graphs are finite because logs have been parsed into templates and only the directly connected events are considered. As an example in the left part of Fig. 7, a pattern  $e_1 \rightarrow e_2 \rightarrow e_3$  matches the event graph of the first request because the graph has  $e_1$  followed by  $e_2$  and  $e_2$  followed by  $e_3$  without other events involved.

**DEFINITION 4 (SUPPORT  $s$ ).** Given a pattern  $p$ ’s count set  $C_p = \{c_1, \dots, c_k\}$ , where  $c_i$  denotes  $p$  occurs  $c_i$  times in the event graph  $g_i$ , the support  $s(p)$  of pattern  $p$  is the sum of the counts in all graphs, i.e.,  $s(p) = \sum_{i=0}^k c_i$ .  $s_C(p)$  and  $s_P(p)$  denote the support of  $p$  in  $G_C$  and  $G_P$ , respectively. We use  $S_C$  and  $S_P$  to denote the support set of all patterns in the  $G_C$  and  $G_P$ , respectively.

After extracting patterns, Pattern Miner counts the occurrences of each pattern to calculate the support of each pattern. Table 2 shows an example of Pattern Miner mining patterns and calculating supports in Fig. 7. In terms of pattern  $e_2 \rightarrow e_3$ , it occurs in all three graphs in the construction phase. Therefore,  $s_C(e_2 \rightarrow e_3) = 3$  in the construction phase. To prevent repeated calculations of  $S_C$ , we persist the results of  $S_C$  into *Pattern Storer*. When diagnosing faults, we are interested in identifying the root causes that result in a large portion of overall abnormal behavior. Therefore, we discard those pattern that rarely occurs by filtering patterns whose support less than  $s_{min}$  ( $s_{min} = 5$  by default).

#### 4.4 Pattern Ranker

After transforming multi-modal data as generalized events (i.e., excluding specific request parameters), the event patterns in the construction phase are similar to those in the production phase.



During the fault-suffering phase, faults manifest themselves in one or more data sources, which causes some event patterns to change. In other words, some event patterns do not follow their expected execution paths in the fault-free phase. Such patterns are likely to reveal insights into events associated with root causes. For example, when a fault occurs between the code region of  $e_2$  and  $e_3$  in Fig. 7, SREs can check the event graphs one by one and identify that the pattern  $e_2 \rightarrow e_3$  in  $G_C$  turns into  $e_2 \rightarrow e_6$  in  $G_P$ . SREs then take the code region between  $e_2$  and  $e_3$  as the root cause candidate and inspect the reason why  $e_2 \rightarrow e_3$  turns into  $e_2 \rightarrow e_6$  to determine the final solution. We designed two rankers: Expected Ranker and Actual Ranker to automate the above RCA process.

**4.4.1 Expected Pattern Ranker.** Expected Pattern Ranker aims to identify *which event patterns do not follow expected execution paths* and take them as *expected patterns*. Then SREs can check why these patterns do not follow expected execution paths to solve faults. The core idea of Expected Ranker is to rank event patterns that occur multiple times in the fault-free phase but rarely in the fault-suffering phase above other event patterns. A ranking score  $Score_E$  is defined to measure how much utility each pattern contributes to root cause diagnosis. For each pattern  $p$  in  $C$ , Expected Pattern Ranker computes its ranking score  $Score_E(p)$  as follows,

$$Score_E(p) = \Pr(g \in G_C \mid p \sqsubseteq g) = \frac{s_C(p)}{s_P(p) + s_C(p)}. \quad (1)$$

The  $Score_E(p)$  of the pattern  $p$  quantifies how distinctive  $p$  is to the  $G_C$  as opposed to the  $G_P$ . If pattern  $p$  occurs multiple times in  $G_C$  while rarely in  $G_P$ ,  $p$  will be assigned a higher score. Therefore, the patterns with higher ranking scores are more suspicious to be root causes. As an example, an exception fault occurs in the code region of  $e_2$  and  $e_3$  in the production phase in Fig. 7. From Fig. 7, we can find that  $e_2$  is always followed by  $e_3$  in  $G_C$  but rarely in  $G_P$ . Thus, it is intuitive to infer that there is a fault in the code region between  $e_2$  and  $e_3$ , causing  $e_2$  not to follow  $e_3$ . In terms of Pattern Ranker, it computes the ranking score of  $e_2 \rightarrow e_3$  as  $Score_E(e_2 \rightarrow e_3) = \frac{3}{3+1} = 0.75$ , which is the highest score (i.e., most suspicious) in the example.

**4.4.2 Actual Pattern Ranker.** Though Expected Pattern Ranker presents event patterns that do not follow expected execution paths, SREs also expect to know *how these patterns change* in the actual fault-suffering phase, which will facilitate SREs to understand faults. Therefore, Actual Pattern Ranker is designed to pinpoint the newly emerging patterns that break their expected execution paths and take them as *actual patterns*. The core idea of Actual Ranker is to rank event patterns that occur multiple times in the fault-suffering phase but rarely in the fault-free phase above other event patterns. For each pattern  $p$  in  $P$ , Actual Pattern Ranker defines its ranking score  $Score_A(p)$  as follows,

$$Score_A(p) = \Pr(g \in G_P \mid p \sqsubseteq g) = \frac{s_P(p)}{s_C(p) + s_P(p)}. \quad (2)$$

Actually, not all patterns in Expected Pattern Ranker and Actual Pattern Ranker provide useful information for root cause diagnoses.

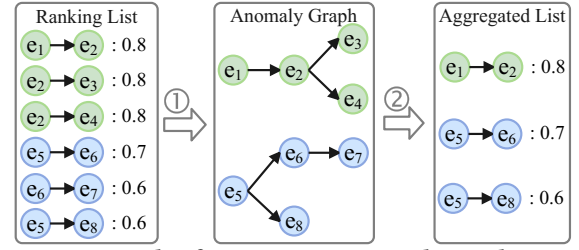


Figure 8: Example of constructing anomaly graphs to get an aggregated list in Pattern Aggregator.

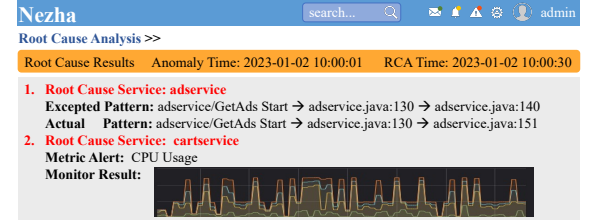


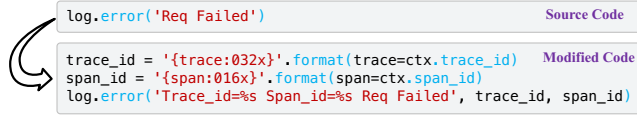
Figure 9: A demo for inspecting root cause candidate.

For example, the pattern  $e_1 \rightarrow e_2$  is given a score 0.5 in Fig. 7. But  $e_1 \rightarrow e_2$  occurs 3 times in both  $G_C$  and  $G_P$ , which is hardly helpful for analyzing the underlying cause. Therefore, we specify a minimum score threshold  $Score_{min}$  to exclude such useless patterns. In this way, the pattern  $p$  is placed in the ranked score list only when  $Score_E(p) \geq Score_{min}$ . After excluding useless patterns, we refer to the ranked pattern list output by Expected Pattern Ranker as the expected patterns list and the ranked pattern list output by Actual Pattern Ranker as the actual patterns list.

## 4.5 Pattern Aggregator

Although Expected Ranker ranks patterns that point towards root causes, it can sometimes return a long list which is unfriendly to SREs. This is because faults may cause downstream patterns of root causes in  $G_P$  to change, resulting in all downstream patterns of root causes in  $G_C$  having a high score as root causes. For instance, in the example in Fig. 7, the patterns  $e_2 \rightarrow e_3$  and  $e_3 \rightarrow e_5$  share the same score as one exception fault between  $e_2$  and  $e_3$  causes neither  $e_3$  nor  $e_5$  to occur in  $G_P$ . Actually,  $e_2 \rightarrow e_3$  and  $e_3 \rightarrow e_5$  point to the same fault. Checking all patterns pointing to the same fault is unnecessary and burdensome for SREs. In this study, we refer to the downstream patterns with the same score like  $e_3 \rightarrow e_5$  as *redundant patterns*. Pattern Aggregator aims to exclude the redundant patterns of Expected Ranker to provide a more valuable list for SREs to speed up troubleshooting.

Pattern Aggregator constructs anomaly graphs by associating expected patterns and filtering redundant patterns by retaining the root patterns of graphs. If both patterns  $e_i \rightarrow e_j$  and  $e_j \rightarrow e_k$  are in the list and  $Score(e_i \rightarrow e_j) \geq Score(e_j \rightarrow e_k)$ , Pattern Aggregator will join  $e_k$  into  $e_i \rightarrow e_j$  (i.e.,  $e_i \rightarrow e_j \rightarrow e_k$ ). After iterating all patterns in the list, Pattern Aggregator obtains some anomaly graphs constructed from these patterns. Phase ① in Fig. 8 shows an example of constructing anomaly graphs. Pattern Aggregator then chooses the root patterns of these anomaly graphs as final expected patterns. Phase ② in Fig. 8 shows that the pattern number decreased from 6 to 3 after excluding redundant patterns.



```

Source Code
log.error('Req Failed')

Modified Code
trace_id = '{trace:032x}'.format(trace=ctx.trace_id)
span_id = '{span:016x}'.format(span=ctx.span_id)
log.error('Trace_id=%s Span_id=%s Req Failed', trace_id, span_id)

```

Figure 10: Example of correlation trace with logs.

To improve the interpretability of *Nezha*, we correlate the expected patterns with actual patterns to provide the complete fault scene. For the expected pattern, we identify its associated actual pattern with the common prefix. For instance, the expected pattern  $e_1 \rightarrow e_2 \rightarrow e_3$  has the common prefix  $e_1 \rightarrow e_2$  as the actual pattern  $e_1 \rightarrow e_2 \rightarrow e_6$  in Fig. 7. If there is more than one actual pattern with the common prefix, we select the actual pattern with the highest score as the actual pattern.

Eventually, *Nezha* outputs a ranked list of root causes candidates to be checked. Candidates are ranked in descending order based on the score of the candidate's expected pattern. For expected patterns with the same score, we place the pattern with the deeper depth in the event graph further up the list because patterns with shallower depths are more likely to be caused by anomaly propagations.

Moreover, we implement a demo of *Nezha*, shown as Fig. 9, to demonstrate root causes candidates for SREs. As shown in Fig. 9, one pair of expected and actual patterns constitutes a root cause candidate. For candidates without metric alert events, *Nezha* shows the service name and the code region between events (e.g., 1st candidate in Fig. 9). Otherwise, *Nezha* displays the metric alert event and corresponding monitoring view (e.g., 2nd candidate in Fig. 9). To sum up, *Nezha* is able to provide fine-grained and actionable root cause candidates and tells SREs why candidates are anomalous.

## 5 EXPERIMENTAL EVALUATION

In this section, we aim to evaluate *Nezha* to answer the following research questions (RQs):

- **RQ1:** How effective is *Nezha* in service-level RCA?
- **RQ2:** How effective is *Nezha* in inner-service-level RCA?
- **RQ3:** How much does multi-modal data contribute to RCA?

### 5.1 Experiment Setup

**Microservice Applications.** We deploy two open-source microservice applications: OnlineBoutique (OB) and TrainTicket (TT) in our testbed. OnlineBoutique is a microservice system for e-commerce and TrainTicket provides a railway ticketing service where users can check, book, and pay for train tickets. Both of them have been widely used in many previous studies [7, 29, 37, 68, 70, 72, 76]. The open-sourced OnlineBoutique and TrainTicket are not equipped with adequate observability (e.g., traces are incomplete and logs do not contain trace ID). In this study, we first instrument the OpenTelemetry SDK [49] for each service to obtain complete traces. We then modify one line of logging pattern in the logging configuration of Java service [39] to insert trace and span IDs into logs. For services implemented in other languages, we need to insert 2 lines of code into the source code based on OpenTelemetry to obtain trace and span IDs and modify the existent log statement for each log (e.g., Fig. 10). With the standard OpenTelemetry toolkit and examples in many programming languages [38–41], it is not difficult for developers to insert trace and span IDs into the logs.

**Experimental Platform.** We deploy the OnlineBoutique and TrainTicket on a Kubernetes platform with 12 virtual machines, each of which has a 8-core 2.10GHz CPU, 16GB memory, and runs with Ubuntu 18.04 OS. We use OpenTelemetry Collector [50] to collect traces and store them in Grafana Tempo [17]. Logs are collected by Grafana Promtail [16] and stored in Grafana Loki [15]. For metrics, we use cAdvisor [5] to collect system-level metrics and Istio [24] to collect application-level metrics (e.g., request latency). Moreover, Prometheus Node Exporter [55] on each node is used to export metrics to Prometheus [54] database to persist them.

**Data Collection.** We use fault injection to mimic application issues following the previous work [35, 37, 42, 68, 72, 73]. To mimic resource issues, we inject CPU contention and network jam faults in the same way as the existing work [35, 37, 68]. Mimicking code defects at program runtime is challenging because we cannot stop or restart services. To achieve this, we design some language-specific fault injectors for the characteristics of program language for Java, Golang, and Python services [4, 28, 53]. We use the above injectors to inject error return and exception code defects following previous work [42, 72, 73]. We set each fault duration to 3 minutes to emulate the process between fault occurrence to fix. We randomly inject one fault into one microservice following previous work [35, 72, 73]. In total, we inject 56 faults (42 resource issues and 14 code defects) into OnlineBoutique and obtain traces and application logs. We inject 45 faults (20 resource issues and 25 code defects) into TrainTicket and get traces and application logs. The collected metrics encompass application-level measurements (e.g., success ratio) and system-level metrics (e.g., CPU usage rate).

**Evaluation Metric.** We use the following two metrics to measure the effectiveness of *Nezha* and baselines because some baselines can only localize root causes at the service level.

- **Top-k accuracy at service level ( $AS@k$ )** refers to the probability that root cause services are included in the top-k results.
- **Top-k accuracy at inner-service level ( $AIS@k$ )** refers to the probability that the inner-service root causes (resource type or code region) are included in the top-k results.

**Implementation and Settings.** We implement the prototype of *Nezha* [48] built on Python 3.6. All experiments are conducted on a Linux server with Intel Xeon Gold 5318Y 2.10GHz CPU, 256 GB RAM, 1TB SSD Disk, and running Ubuntu 18.04. The minimum score threshold  $Score_{min}$ , which is used to avoid the influence of normal fluctuations, is set to 0.67 (i.e.,  $\frac{2}{3}$ ) by default. In this case, a pattern  $p$  in Expected Pattern Ranker is suspicious if the support of  $p$  in fault-suffering phase is less than half of the support of  $p$  in fault-free phase. The influence of  $Score_{min}$  is discussed in § 5.3.5.

### 5.2 Baselines

We use the following six state-of-the-art unsupervised metric-based, trace-based, and log-based RCA approaches as the baselines. We do not consider the supervised approaches because they need a large training dataset with labels, which is hard to obtain in practice.

- MicroScope [35] is a metric-based RCA approach that identifies root causes based on the correlation of metrics along dependency.
- MicroRCA [65] presents a metric-based RCA approach that localizes suspicious services by applying a PageRank method [51] on the extracted anomaly sub-graph.



**Table 3: Comparison of baselines at service level.**

Approach	OnlineBoutique			TrainTicket		
	AS@1	AS@3	AS@5	AS@1	AS@3	AS@5
MicroScope	12.5	41.07	55.35	17.78	26.67	35.56
MicroRCA	16.07	62.5	92.75	20.00	31.11	44.44
SBLD	19.64	23.21	25.00	15.56	22.22	24.44
LogFaultFlagger	19.64	21.42	23.21	17.78	24.44	24.44
MicroRank	41.07	48.21	62.5	15.56	24.44	35.56
TraceAnomaly	30.35	33.92	48.21	13.33	28.89	33.33
PDiagnose	41.07	73.21	82.14	8.89	13.33	22.22
<i>Nezha</i> w/o $M\mathcal{L}$	14.28	17.85	17.85	6.67	8.89	11.11
<i>Nezha</i> w/o $M$	26.78	33.92	35.71	55.56	62.22	68.89
<i>Nezha</i> w/o $\mathcal{L}$	64.28	64.28	64.28	42.22	44.44	44.44
<i>Nezha</i>	<b>92.86</b>	<b>96.43</b>	<b>96.43</b>	<b>86.67</b>	<b>97.78</b>	<b>97.78</b>

- LogFaultFlagger [1] is a log-based RCA approach that compares passing and failing logs to find faults in failing logs.
- SBLD [56] is a log-based RCA approach that analyzes the coverage of log events using Spectrum algorithms [25] to find suspicious log events.
- MicroRank [68] proposes a trace-based RCA approach that combines the personalised Pagerank method and Spectrum method to locate suspicious root causes.
- TraceAnomaly [37] provides a trace-based RCA approach that adopts a deep learning method to learn normal patterns of traces offline and detect anomalous traces online to perform RCA.
- PDiagnose [21] takes metrics, traces, and logs as input and transforms them into time series. Then PDiagnose determines root causes through voting abnormal time series.

### 5.3 Evaluation Results

**5.3.1 RQ1: Effectiveness at service level.** The ground truths at the service level are the known injected services. Table 3 shows the effectiveness evaluation results of different approaches at the service level. From Table 3 Table 3, we can observe that *Nezha* outperforms all the baseline approaches significantly and achieves high accuracy in AS@1 (90%), AS@3 (97%), AS@5 (97%) on average, illustrating that *Nezha* can successfully localize root causes at service level most of the time. The excellent performance of *Nezha* is mainly attributed to the fact that *Nezha* takes multi-modal data as input and fuses them so that it can capture the abnormal behaviours of a wider range of fault situations.

The metric-based approaches MicroScope and MicroRCA achieve low AS@1 and AS@5 on average. After a detailed dissection of their RCA results, we find that MicroScope and MicroRCA are adept at locating the root causes of resource issues but not good at code defects. This is because MicroScope and MicroRCA only take metrics into account, but many code defects do not manifest themselves in metrics. The accuracy of MicroScope is lower than MicroRCA because the design of MicroScope never places the frontend service as the root cause.

The two log-based approaches (i.e., SBLD and LogFaultFlagger) obtain AS@1 and AS@5 less than 30%. These two approaches do not consider metrics. Thus, they miss the root causes of faults caused by resources because these faults would not change the log sequences. The average AS@5 of two trace-based approaches (i.e., MicroRank and TraceAnomaly) reach almost 50% and 40%, respectively. Both approaches take traces as input and use the latency of spans to

**Table 4: Comparison of baselines at inner-service level**

Approach	OnlineBoutique			TrainTicket		
	AIS@1	AIS@3	AIS@5	AIS@1	AIS@3	AIS@5
SBLD	14.28	17.85	17.85	15.56	22.22	24.44
LogFaultFlagger	19.64	21.42	21.42	15.56	24.44	24.44
PDiagnose	35.71	53.57	71.42	8.89	13.33	15.56
<i>Nezha</i> w/o $M$	26.78	33.92	35.71	55.56	62.22	68.89
<i>Nezha</i> w/o $\mathcal{L}$	64.28	64.28	64.28	42.22	44.44	44.44
<i>Nezha</i>	<b>92.86</b>	<b>96.43</b>	<b>96.43</b>	<b>86.67</b>	<b>97.78</b>	<b>97.78</b>

find root causes. Therefore, they can only localize the faults that have significant impacts on latency. However, the error return and exception faults do not manifest as anomalies in latency, which results in low accuracy for MicroRank and TraceAnomaly.

Multi-modal approach PDiagnose achieves better accuracy than single-modal baselines in OnlineBoutique. However, the accuracy of PDiagnose degrades dramatically (from 82.14% to 22.22% at AS@5) in TrainTicket. The poor performance of PDiagnose is attributed to two reasons. (1) PDiagnose transforms multi-modal data as time series and localizes root causes based on the anomalous time series. However, not all faults cause time series anomalies, leading to missing some root causes. (2) PDiagnose performs RCA based on a simple voting mechanism that ignores the service dependency and anomaly propagation, which is challenging to get consensus in TrainTicket with 41 microservices.

In conclusion, *Nezha* is effective in microservice root cause diagnosis at the service level and improves AS@1 by 61.45% ~ 74.63% and AS@5 by 28.51% ~ 73.28% on average compared to baselines. These results also validate our motivation to integrate multi-modal data to facilitate root cause localization in § 2.2.

**5.3.2 RQ2: Effectiveness at inner-service level.** The ground truths at the inner-service level are the code region or resource type extracted from the fault-injected operation. We only compare *Nezha* with SBLD, LogFaultFlagger, and PDiagnose because only these three baselines have the ability to identify root causes at the inner-service level. Considering that the above three baselines are designed to pinpoint error logs rather than code regions, their result would be determined to be correct if their output error logs are within the code region of root causes.

Table 4 shows the effectiveness of different approaches at the inner-service level. *Nezha* performs the best by taking all baselines into consideration, achieving AIS@1 of 87%, AIS@3 of 97%, and AIS@5 of 97% on average. With the incorporation of multi-modal data while retaining execution contexts, *Nezha* can localize root causes at the inner-service level more accurately. SBLD and LogFaultFlagger use the differences in frequency and coverage of logs to locate root causes. Resource faults do not cause a significant difference in frequency and coverage of logs, so SBLD and LogFaultFlagger cannot accurately identify these root causes. PDiagnose performs better than SBLD and LogFaultFlagger because it considers valuable metrics and traces ignored by SBLD and LogFaultFlagger. However, the performance of PDiagnose degrades dramatically in TrainTicket dataset because the voting mechanism of PDiagnose is difficult to get consensus in a system with many services. To overcome the drawbacks of SBLD, LogFaultFlagger, and PDiagnose, *Nezha* inserts metrics alert events into logs events

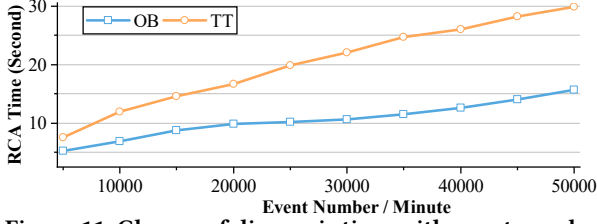


Figure 11: Change of diagnosis time with event number.

so that *Nezha* can handle both the code defects and resource faults while retaining execution context.

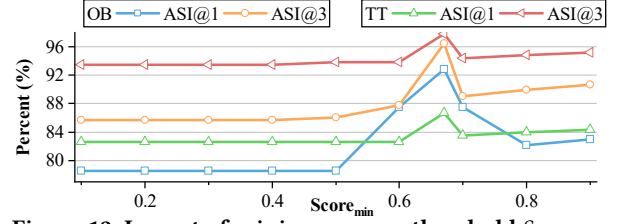
To sum up, the results demonstrate the effectiveness of *Nezha* in localizing root causes at the inner-service level and improves *AIS@1* by 67.47% ~ 74.85% and *AIS@5* by 53.61% ~ 75.96% on average compared to baselines.

**5.3.3 Contribution of Multi-modal Data.** We perform two ablation studies to explore the contribution of multi-modal data, so we derive the following variants: *Nezha w/o ML* that drops metrics and logs, *Nezha w/o M* that drops metrics and *Nezha w/o L* that drops logs. The ablation study results on service and inner-service level are shown at the bottom of Table 3 and Table 4, respectively. We can see that each data source contributes to the effectiveness of *Nezha* because *Nezha* that takes all multi-modal data as input performs the best.

In addition, we observe that the contributions degrees of logs and metrics are not exactly the same in different datasets. *Nezha w/o L* is the second-best in OnlineBoutique dataset while *Nezha w/o M* performs the second-best in TrainTicket dataset. We believe this difference is due to the distribution of fault types. In the OnlineBoutique dataset, there are more resource issues, so *Nezha w/o L* performs better. However, in the TrainTicket dataset, there are more code defects, so *Nezha w/o M* performs better. Though *Nezha w/o ML* performs the worst in ablation studies, it does not mean that the contribution degree of trace data is low because trace is the core of linking logs of different services and building event graphs.

**5.3.4 Efficiency of *Nezha*.** The diagnosis time of *Nezha* is essential for achieving timely root cause analysis. It depends highly on the size of events in a time window. To evaluate the scalability of *Nezha* with event size, we conduct an experiment on the changes of diagnosis time with the increase of event number. The time to calculate fault-free patterns is not included in the diagnosis time because the patterns in the fault-free phase is calculated once and used multiple times. Fig. 11 shows the diagnosis time of *Nezha* under the different event numbers in a time window. It can be seen that the diagnosis time of *Nezha* increases linearly with the number of events. In a time window of 50,000 events, OnlineBoutique and TrainTicket take 16 seconds and 30 seconds to determine root causes, respectively. Analysing TrainTicket data needs longer than OnlineBoutique because TrainTicket has a larger number of spans per trace and more complex dependencies. Thus, *Nezha* takes longer to traverse all spans of traces.

**5.3.5 Sensitivity of *Nezha*.** The minimum score threshold  $Score_{min}$  is one crucial factor that may impact the performance of *Nezha*. As stated in Sec. 4.3,  $Score_{min}$  controls the minimum score of patterns that will be considered as root causes. Fig. 12 shows the impact of

Figure 12: Impact of minimum score threshold  $Score_{min}$ .

$Score_{min}$  on *AIS@1* and *AIS@3* of *Nezha* on two datasets. Smaller  $Score_{min}$  usually leads to lower accuracy as some event patterns with low scores caused by normal fluctuation of the system affect the final results. According to Fig. 12,  $Score_{min} = 0.67$  achieves the best accuracy in both two datasets. The results also demonstrate that *Nezha* exhibits less sensitivity to  $Score_{min}$  when it is set above 0.6. Note that the best configuration of  $Score_{min}$  highly depends on the characteristics of datasets.

## 6 DISCUSSION

### 6.1 Limitations and Future Work

*Nezha* relies on the anomaly detection approach triggers for RCA, thus it cannot identify root causes for faults that escape anomaly detection. Future work can integrate more robust anomaly detection algorithms (e.g., USAD [2] and JumpStarter [43]) and monitor additional metrics beyond the P90 latency and success ratio to avoid missing faults.

The applicability of *Nezha* is limited to troubleshooting faults that exhibit abnormal patterns in multi-modal data. Some byzantine faults, such as returning an unreasonable result to the user, cannot be identified by *Nezha* because these faults do not manifest themselves as any abnormal patterns. Troubleshooting for such byzantine faults like [66] is the future work to improve *Nezha*.

*Nezha* does not incorporate system-level metrics from worker nodes (e.g., physical or virtual servers), which restricts its ability to pinpoint root causes of node failures. In the future, we will try to consider system-level metrics of worker nodes in *Nezha* to pinpoint the root causes of node failures.

### 6.2 Threats to Validity

The threats to the internal validity mainly lie in the fault-free data collection and minimum score threshold  $Score_{min}$ , which can introduce bias on the effectiveness of *Nezha*. (1) The accuracy of *Nezha* can be affected if fault-free data is noisy or lacks certain types of requests. To mitigate this threat, it is recommended to construct fault-free data that includes a wide range of request types and has a similar number of requests to the production phase. Capturing workloads of systems and replaying these workloads is a common approach in the software test phase. SREs can easily collect such fault-free data when replaying workloads. Even without replaying workloads, collecting fault-free data within a short time window is not difficult as the production environment is predominantly in a normal status with limited faults [30]. (2) As the pattern of a similar number of occurrences in the fault and fault-free phases hardly provides a clue for RCA, the minimum score threshold  $Score_{min}$  is used to exclude such uninformative patterns. We recommended setting  $Score_{min}$  above 0.6 because a pattern  $p$  is considered more

suspicious if the support of  $p$  in fault-suffering phase is less than the support of  $p$  in fault-free phase.

The external threat mainly comes from the microservice modification and experimental environment. (1) The integration analysis of *Nezha* relies on the insertion of trace ID into logs, which may not be available in some practical systems. However, with the standard Opentelemetry toolkit and examples in various programming languages [38–41], it is easy for developers to insert trace and span IDs into the logs. For example, for Java applications, we only need to modify one line of logging pattern in the logging configuration [39]. (2) *Nezha* is evaluated on two widely-used microservice systems in a Kubernetes platform. It needs further effort to validate the effectiveness of *Nezha* in more complex real-world systems. However, it is reasonable to believe *Nezha* can work in such a system according to current results. The complexity of the systems is alleviated by constructing datasets that include more than 600 events for a single request and involves parallel and asynchronous service calls. Typically, a request involving dozens of service calls in industrial microservice systems contains hundreds of log events [72]. Therefore, the amount of events is comparable. The complexity of fault scenarios is alleviated by injecting resource issues and code defects from real faults in industrial systems [33, 76] into microservices. Thus, the fault scenarios in our evaluation are representative.

## 7 RELATED WORK

**Metric-based RCA.** Metric-based RCA approaches commonly diagnose problems by mining the relations between different metrics [6, 26, 35, 62, 64, 65]. CauseInfer [6] and Microscope [35] construct a causality graph using the PC-algorithm, and identify root causes based on the correlation of different metrics along the causal paths. MicroRCA [65] first extracts an anomalous sub-graph from an attribute graph including service dependencies and performance metrics. A personalised PageRank method is then used on the anomaly sub-graph to locate root causes. MicroDiag [64] first derives a metric causality graph by Granger Causality tests. Then it weighs the causality graph with the pearson correlation coefficient between two metrics and ranks the root causes with PageRank. GROOT [61] constructs the causality graph using monitoring events such as performance metrics deviation events and ranks the most probable root causes from the event causality graph based on a customized PageRank algorithm. An important drawback of metrics-based RCA approaches is that analyzing metrics provides a superficial analysis of the system’s operations but not a dissection of how the system is actually running.

**Log-based RCA.** Most of log-based RCA methods localize root causes by comparing frequent patterns between logs of normal and abnormal phase [1, 11, 34, 36, 56]. SBLD [56] applies spectrum algorithms [25] to the logging domain by abstracting logs into events and locating root causes by analyzing the coverage of events. Facebook proposes a fast dimensional analysis framework to locate the frequent items most likely to be the root cause by mining the difference between the frequent item sets of normal and abnormal log [34]. LogCluster [36] uses log clustering to mine log sequences and compares production with test log sequences to find previously unseen ones. However, current logs-based RCA approaches do not take into account the contextual information of requests and cannot identify faults where logs have not changed.

**Trace-based RCA.** Various work and tools [10, 49, 57, 58, 60] have been proposed to generate traces by instrumenting trace code into the source code of applications. The empirical study in [76] shows that microservice debugging can be improved by employing proper tracing and visualization techniques and strategies. But the study does not provide an automatic RCA approach based on traces. GMTA [18] is a graph-based trace analysis approach implemented and deployed in eBay. It abstracts traces into different paths and further groups them into business flows for architecture understanding and problem diagnosis. Nevertheless, such aggregated trace analysis approach may mask a small number of abnormal traces that are critical to root cause localization. Based on the insight that a microservice that is traversed by more abnormal and fewer normal traces is likely to be the root cause, T-Rank [67] proposes a lightweight performance diagnostic tool built on spectrum analysis. However, if two different microservices have similar coverage information, T-Rank cannot distinguish between them. MicroRank [68] and TraceRank [71] combine PageRank and spectrum algorithms to distinguish two different microservices with similar coverage. Nevertheless, the root causes output by MicroRank and TraceRank (i.e., service instance level or operation level) are more coarse than *Nezha*. Overall, traces-based RCA approaches are limited by the granularity of traces, which is mostly at the operation level rather than the code region level. Moreover, the absence of system-level metrics also makes traces-based approaches impossible to specify whether the faults are caused by resources.

## 8 CONCLUSION

In this study, we present *Nezha*, an interpretable and fine-grained RCA approach based on multi-modal observability data. *Nezha* unifies multi-modal data as events within a single solution and mines event patterns by constructing event graphs. *Nezha* correlates and ranks event patterns by identifying which event patterns do not follow expected execution paths and how these patterns change in the fault-suffering phase to localize root causes. We have implemented a prototype of *Nezha* and conducted extensive evaluations on two widely-used microservice applications. Our results show that *Nezha* achieves a high top1 accuracy at both the service and inner-service level and outperforms state-of-the-art approaches by a large margin. Moreover, *Nezha* deals with events with high scalability, which makes it practical for industrial systems.

## 9 ARTIFACT AVAILABILITY.

The data and the implementation of *Nezha* are publicly available at Github [48]. The augmented OnlineBoutique and TrainTicket are available at [46] and [47].

## ACKNOWLEDGMENTS

We greatly appreciate the insightful feedback from the anonymous reviewers. The research is supported by the National Key Research and Development Program of China (2019YFB1804002), the National Natural Science Foundation of China (No.62272495), the Guangdong Basic and Applied Basic Research Foundation (No.2023B1515020054), and the CCF-Lenovo Blue Ocean Research Fund.



## REFERENCES

- [1] Anunay Amar and Peter C. Rigby. 2019. Mining historical test logs to predict bugs and localize faults in the test logs. In *ICSE 2019*. IEEE / ACM, 140–151. <https://doi.org/10.1109/ICSE.2019.00031>
- [2] Julien Audibert, Pietro Michiardi, Frédéric Guyard, Sébastien Marti, and Maria A. Zuluaga. 2020. USAD: UnSupervised Anomaly Detection on Multivariate Time Series. In *KDD 2020*. ACM, 3395–3404. <https://doi.org/10.1145/3394486.3403392>
- [3] Chetan Bansal, Sundararajan Renganathan, Ashima Asudani, Olivier Midy, and Mathru Janakiraman. 2020. DeCaf: diagnosing and triaging performance issues in large-scale cloud services. In *ICSE-SEIP 2020*. ACM, 201–210. <https://doi.org/10.1145/3377813.3381353>
- [4] Byteman. 2023. Java Byteman. <https://github.com/bytemanproject/byteman>. Accessed Jan. 6, 2023.
- [5] cAdvisor. 2023. cAdvisor. <https://github.com/google/cadvisor>. Accessed Jan. 6, 2023.
- [6] Pengfei Chen, Yong Qi, Pengfei Zheng, and Di Hou. 2014. CausalInfer: automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. In *INFOCOM 2014*. IEEE, 1887–1895. <https://doi.org/10.1109/INFOCOM.2014.6848128>
- [7] Yufu Chen, Meng Yan, Dan Yang, Xiaohong Zhang, and Ziliang Wang. 2022. Deep Attentive Anomaly Detection for Microservice Systems with Multimodal Time-Series Data. In *ICWS 2022*. IEEE, 373–378. <https://doi.org/10.1109/ICWS55610.2022.00062>
- [8] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *OSDI 2014*. USENIX Association, 217–231.
- [9] Francois Doray and Michel Dagenais. 2017. Diagnosing Performance Variations by Comparing Multi-Level Execution Traces. *IEEE TPDS* 28, 2 (2017), 462–474. <https://doi.org/10.1109/TPDS.2016.2567390>
- [10] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-Trace: A Pervasive Network Tracing Framework. In *NSDI, 2007*. USENIX, 271–284.
- [11] Xiaoyu Fu, Rui Ren, Sally A. McKee, Jianfeng Zhan, and Ninghui Sun. 2014. Digging deeper into cluster system logs for failure prediction and root cause diagnosis. In *CLUSTER 2014*. IEEE, 103–112. <https://doi.org/10.1109/CLUSTER.2014.6968768>
- [12] FudanSELab. 2023. TrainTicket. <https://github.com/FudanSELab/train-ticket>. Accessed Jan. 6, 2023.
- [13] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: practical and scalable ML-driven performance debugging in microservices. In *ASPLOS 2021*. ACM, 135–151. <https://doi.org/10.1145/3445814.3446700>
- [14] GoogleCloudPlatform. 2023. OnlineBoutique. <https://github.com/GoogleCloudPlatform/microservices-demo>. Accessed Jan. 6, 2023.
- [15] Grafana. 2023. Grafana loki. <https://github.com/grafana/loki>. Accessed Jan. 6, 2023.
- [16] Grafana. 2023. Grafana promtail. <https://grafana.com/docs/loki/latest/clients/promtail/>. Accessed Jan. 6, 2023.
- [17] Grafana. 2023. Grafana Tempo. <https://github.com/grafana/tempo>. Accessed Jan. 6, 2023.
- [18] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. 2020. Graph-Based Trace Analysis for Microservice Architecture Understanding and Problem Diagnosis. In *ESEC/FSE 2020*. ACM, 1387–1397. <https://doi.org/10.1145/3368089.3417066>
- [19] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. 2017. Drain: An Online Log Parsing Approach with Fixed Depth Tree. In *ICWS 2017*. IEEE, 33–40. <https://doi.org/10.1109/ICWS.2017.13>
- [20] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2018. Identifying impactful service system problems via log analysis. In *ESEC/FSE 2018*. ACM, 60–70. <https://doi.org/10.1145/3236024.3236083>
- [21] Chuanjia Hou, Tong Jia, Yifan Wu, Ying Li, and Jing Han. 2021. Diagnosing Performance Issues in Microservices with Heterogeneous Data Source. In *ISPA/BD-Cloud/SocialCom/SustainCom, 2021*. IEEE, 493–500. <https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00074>
- [22] Lexiang Huang and Timothy Zhu. 2021. tprof: Performance Profiling via Structural Aggregation and Automated Analysis of Distributed Systems Traces. In *SoCC 2021*. ACM, 76–91. <https://doi.org/10.1145/3472883.3486994>
- [23] Zicheng Huang, Pengfei Chen, Guangba Yu, Hongyang Chen, and Zibin Zheng. 2021. Sieve: Attention-based Sampling of End-to-End Trace Data in Distributed Microservice Systems. In *ICWS 2021*. IEEE, 436–446. <https://doi.org/10.1109/ICWS53863.2021.00063>
- [24] Istio. 2023. Istio. <https://github.com/istio/istio>. Accessed Jan. 6, 2023.
- [25] James A. Jones, Mary Jean Harrold, and John T. Stasko. 2002. Visualization of test information to assist fault localization. In *ICSE 2002*. ACM, 467–477. <https://doi.org/10.1145/581339.581397>
- [26] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. 2009. Detailed diagnosis in enterprise networks. In *SIGCOMM 2009*. ACM, 243–254. <https://doi.org/10.1145/1592568.1592597>
- [27] Kmaork. 2023. Hypno. <https://docs.aws.amazon.com/prescriptive-guidance/latest/implementing-logging-monitoring-cloudwatch/configure-cloudwatch-ec2-on-premises.html>. Accessed Jan. 6, 2023.
- [28] Kmaork. 2023. Hypno. <https://github.com/kmaork/hypno>. Accessed Jan. 6, 2023.
- [29] Xing Li, Yan Chen, and Zhiqiang Lin. 2019. Towards automated inter-service authorization for microservice applications. In *SIGCOMM 2019*. ACM, 3–5. <https://doi.org/10.1145/3342280.3342288>
- [30] Xiaoyun Li, Guangba Yu, Pengfei Chen, Hongyang Chen, and Zhekang Chen. 2022. Going through the Life Cycle of Faults in Clouds: Guidelines on Fault Handling. In *ISSRE 2022*. IEEE, 121–132. <https://doi.org/10.1109/ISSRE55969.2022.00022>
- [31] Yufeng Li, Guangba Yu, Pengfei Chen, Chuanfu Zhang, and Zibin Zheng. 2022. MicroSketch: Lightweight and Adaptive Sketch Based Performance Issue Detection and Localization in Microservice Systems. In *ICSOC 2022 (Lecture Notes in Computer Science, Vol. 13740)*. Springer, 219–236. [https://doi.org/10.1007/978-3-031-20984-0\\_15](https://doi.org/10.1007/978-3-031-20984-0_15)
- [32] Zeyan Li, Junjie Chen, Rui Jiao, Nengwen Zhao, Zhijun Wang, Shuwei Zhang, Yanjun Wu, Long Jiang, Leiqin Yan, and Zikai Wang. 2021. Practical Root Cause Localization for Microservice Systems via Trace Analysis. In *IWQoS 2021*. IEEE, 1–10. <https://doi.org/10.1109/IWQoS52092.2021.9521340>
- [33] Zeyan Li, Nengwen Zhao, Mingjie Li, Xianglin Lu, Lixin Wang, Dongdong Chang, Xiaohui Nie, Li Cao, Wenchi Zhang, Kaixin Sui, Yanhua Wang, Xu Du, Guoqiang Duan, and Dan Pei. 2022. Actionable and interpretable fault localization for recurring failures in online service systems. In *ESEC/FSE 2022*. ACM, 996–1008. <https://doi.org/10.1145/3540250.3549092>
- [34] Fan Fred Lin, Keyur Muzumdar, Nikolay Pavlovich Laptev, Mihai-Valentin Curelea, Seunghak Lee, and Sriram Sankar. 2020. Fast Dimensional Analysis for Root Cause Investigation in a Large-Scale Service Environment. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 2 (2020), 31:1–31:23. <https://doi.org/10.1145/3392149>
- [35] Jinjin Lin, Pengfei Chen, and Zibin Zheng. 2018. Microscope: Pinpoint Performance Issues with Causal Graphs in Micro-service Environments. In *ICSOC 2018*. Springer, 3–20. [https://doi.org/10.1007/978-3-030-03596-9\\_1](https://doi.org/10.1007/978-3-030-03596-9_1)
- [36] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuwei Chen. 2016. Log clustering based problem identification for online service systems. In *ICSE Companion 2016*. ACM, 102–111. <https://doi.org/10.1145/2889160.2889232>
- [37] Ping Liu, Haowen Xu, and et al. 2020. Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In *ISSRE 2020*. IEEE, 48–58. <https://doi.org/10.1109/ISSRE5003.2020.00014>
- [38] Sumo logic. 2023. Go Traceld and SpanId injection into logs configuration. <https://help.sumologic.com/docs/apm/traces/get-started-transaction-tracing/opentelemetry-instrumentation/go/traceld-and-spanid-injection-into-logs/>. Accessed June 6, 2023.
- [39] Sumo logic. 2023. Java Traceld and SpanId injection into logs configuration. <https://help.sumologic.com/docs/apm/traces/get-started-transaction-tracing/opentelemetry-instrumentation/java/traceld-spanid-injection-into-logs-configuration/>. Accessed June 6, 2023.
- [40] Sumo logic. 2023. JavaScript Traceld and SpanId injection into logs configuration. <https://help.sumologic.com/docs/apm/traces/get-started-transaction-tracing/opentelemetry-instrumentation/javascript/traceld-spanid-injection-into-logs/>. Accessed June 6, 2023.
- [41] Sumo logic. 2023. Python Traceld and SpanId injection into logs configuration. <https://help.sumologic.com/docs/apm/traces/get-started-transaction-tracing/opentelemetry-instrumentation/python/traceld-spanid-injection-into-logs/>. Accessed June 6, 2023.
- [42] Chang Lou, Peng Huang, and Scott Smith. 2020. Understanding, Detecting and Localizing Partial Failures in Large System Software. In *NSDI 2020*. USENIX Association, 559–574.
- [43] Minghua Ma, Shenglin Zhang, Junjie Chen, Jim Xu, Haozhe Li, Yongliang Lin, Xiaohui Nie, Bo Zhou, Yong Wang, and Dan Pei. 2021. Jump-Starting Multivariate Time Series Anomaly Detection for Online Service Systems. In *USENIX ATC 2021*. USENIX Association, 413–426.
- [44] Vijayaraghavan Murali, Edward Yao, Umang Mathur, and Satish Chandra. 2021. Scalable Statistical Root Cause Analysis on App Telemetry. In *ICSE (SEIP) 2021*. IEEE, 288–297. <https://doi.org/10.1109/ICSE-SEIP52600.2021.00038>
- [45] Karthik Nagaraj, Charles Edwin Killian, and Jennifer Neville. 2012. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *NSDI 2012*. USENIX Association, 353–366.
- [46] Nezha. 2023. Augmented-OnlineBoutique. <https://github.com/IntelligentDDS/Augmented-OnlineBoutique>. Accessed Jan. 6, 2023.
- [47] Nezha. 2023. Augmented-TrainTicket. <https://github.com/IntelligentDDS/Augmented-TrainTicket>. Accessed Jan. 6, 2023.
- [48] Nezha. 2023. Nezha implementation. <https://github.com/IntelligentDDS/Nezha>. Accessed Jan. 6, 2023.
- [49] Opentelemetry. 2023. Opentelemetry. <https://opentelemetry.io>. Accessed Jan. 6, 2023.
- [50] Opentelemetry. 2023. OpenTelemetry Collector. <https://github.com/open-telemetry/opentelemetry-collector>. Accessed Jan. 6, 2023.
- [51] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web*. Technical Report. Stanford InfoLab.

- [52] Yicheng Pan, Meng Ma, Xinrui Jiang, and Ping Wang. 2021. Faster, deeper, easier: crowdsourcing diagnosis of microservice kernel failure from user space. In *ISSTA 2021*. ACM, 646–657. <https://doi.org/10.1145/3460319.3464805>
- [53] Pingcap. 2023. Golang Failpoint. <https://github.com/pingcap/failpoint>. Accessed Jan. 6, 2023.
- [54] Prometheus. 2023. Prometheus. <https://github.com/prometheus/prometheus>. Accessed Jan. 6, 2023.
- [55] Prometheus. 2023. Prometheus node exporter. [https://github.com/prometheus/node\\_exporter](https://github.com/prometheus/node_exporter). Accessed Jan. 6, 2023.
- [56] Carl Martin Rosenberg and Leon Moonen. 2020. Spectrum-Based Log Diagnosis. In *ESEM 2020*. ACM, 18:1–18:12. <https://doi.org/10.1145/3382494.3410684>
- [57] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspán, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).
- [58] Apache SkyWalking. 2023. Apache SkyWalking. <https://skywalking.apache.org>. Accessed Jan. 6, 2023.
- [59] Cindy Sridharan. 2018. *Distributed systems observability: a guide to building robust systems*. O'Reilly Media.
- [60] Avishay Traeger, Ivan Deras, and Erez Zadok. 2008. DARC: dynamic analysis of root causes of latency distributions. In *SIGMETRICS 2008*. ACM, 277–288. <https://doi.org/10.1145/1375457.1375489>
- [61] Hanzhang Wang, Zhengkai Wu, Huai Jiang, Yichao Huang, Jiamu Wang, Selcuk Kopru, and Tao Xie. 2021. Groot: An event-graph-based approach for root cause analysis in industrial settings. In *ASE 2021*. IEEE, 419–429. <https://doi.org/10.1109/ASE51524.2021.9678708>
- [62] 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 *CCGRID 2018*. IEEE/ACM, 492–502. <https://doi.org/10.1109/CCGRID.2018.00076>
- [63] Paul F Wilson, Larry D Dell, and Gaylor F Anderson. 1996. Root cause analysis: a tool for total quality management. *The Journal for Healthcare Quality (JHQ)* 18, 1 (1996), 40.
- [64] Li Wu, Johan Tordsson, Jasmin Bogatinovski, Erik Elmroth, and Odej Kao. 2021. MicroDiag: Fine-grained Performance Diagnosis for Microservice Systems. In *Cloud Intelligence 2021*. IEEE, 31–36.
- [65] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. MicroRCA: Root Cause Localization of Performance Issues in Microservices. In *NOMS 2020*. IEEE/IFIP, 1–9. <https://doi.org/10.1109/NOMS47738.2020.9110353>
- [66] Hiroyuki Yamada and Jun Nemoto. 2022. Scalar DL: Scalable and Practical Byzantine Fault Detection for Transactional Database Systems. *Proc. VLDB Endow.* 15, 7 (2022), 1324–1336.
- [67] Zihao Ye, Pengfei Chen, and Guangba Yu. 2021. T-Rank: A Lightweight Spectrum based Fault Localization Approach for Microservice Systems. In *CCGrid 2021*. IEEE/ACM, 416–425. <https://doi.org/10.1109/CCGrid51090.2021.00051>
- [68] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Ximmeng Sun, and Xiaoyun Li. 2021. MicroRank: End-to-End Latency Issue Localization with Extended Spectrum Analysis in Microservice Environments. In *WWW 2021*. ACM, 3087–3098. <https://doi.org/10.1145/3442381.3449905>
- [69] Guangba Yu, Pengfei Chen, Pairui Li, Tianjun Weng, Haibing Zheng, and Yuetang Deng. 2023. LogReducer: Identify and Reduce Log Hotspots in Kernel on the Fly. In *ICSE 2023*. IEEE, 1763–1775. <https://doi.org/10.1109/ICSE48619.2023.00151>
- [70] Guangba Yu, Pengfei Chen, and Zibin Zheng. 2019. Microscaler: Automatic Scaling for Microservices with an Online Learning Approach. In *ICWS 2019*. IEEE, 68–75. <https://doi.org/10.1109/ICWS.2019.00023>
- [71] Guangba Yu, Zicheng Huang, and Pengfei Chen. 2021. TraceRank: Abnormal service localization with dis-aggregated end-to-end tracing data in cloud native systems. *Journal of Software: Evolution and Process* (2021), e2413. <https://doi.org/10.1002/smr.2413>
- [72] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. 2022. DeepTraLog: Trace-Log Combined Microservice Anomaly Detection through Graph-based Deep Learning. In *ICSE 2022*. IEEE, 623–634. <https://doi.org/10.1145/3510003.3510180>
- [73] Chenxi Zhang, Xin Peng, Tong Zhou, Chaofeng Sha, Zhenghui Yan, Yiru Chen, and Hong Yang. 2022. TraceCRL: contrastive representation learning for microservice trace analysis. In *ESEC/FSE 2022*. ACM, 1221–1232. <https://doi.org/10.1145/3540250.3549146>
- [74] Yingying Zhang, Zhengxiong Guan, Huajie Qian, Leili Xu, Hengbo Liu, Qingsong Wen, Liang Sun, Junwei Jiang, Lunting Fan, and Min Ke. 2021. CloudRCA: A Root Cause Analysis Framework for Cloud Computing Platforms. In *CIKM 2021*. ACM, 4373–4382. <https://doi.org/10.1145/3459637.3481903>
- [75] Nengwen Zhao, Junjie Chen, Zhaoyang Yu, Honglin Wang, Jiesong Li, Bin Qiu, Hongyu Xu, Wenchi Zhang, Kaixin Sui, and Dan Pei. 2021. Identifying bad software changes via multimodal anomaly detection for online service systems. In *ESEC/FSE '21*. ACM, 527–539. <https://doi.org/10.1145/3468264.3468543>
- [76] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2021. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE TSE* 47, 2 (2021), 243–260. <https://doi.org/10.1109/TSE.2018.2887384>

Received 2023-03-02; accepted 2023-07-27