TrinityRCL: Multi-Granular and Code-Level Root Cause Localization Using Multiple Types of Telemetry Data in Microservice Systems

Shenghui Gu, Guoping Rong, Tian Ren, He Zhang, Haifeng Shen, Yongda Yu, Xian Li, Jian Ouyang, and Chunan Chen

***Abstract*—The microservice architecture has been commonly adopted by large scale software systems exemplified by a wide range of online services. Service monitoring through anomaly detection and root cause analysis (RCA) is crucial for these microservice systems to provide stable and continued services. However, compared with monolithic systems, software systems based on the layered microservice architecture are inherently complex and commonly involve entities at different levels of granularity. Therefore, for effective service monitoring, these systems have a special requirement of multi-granular RCA. Furthermore, as a large proportion of anomalies in microservice systems pertain to problematic code, to timely troubleshoot these anomalies, these systems have another special requirement of RCA at the finest code-level. Microservice systems rely on telemetry data to perform service monitoring and RCA of service anomalies. The majority of existing RCA approaches are only based on a single type of telemetry data and as a result can only support uni-granular RCA at either application-level or service-level. Although there are attempts to combine metric and tracing data in RCA, their objective is to improve RCA's efficiency or accuracy rather than to support multi-granular RCA. In this article, we propose a new RCA solution *TrinityRCL* that is able to localize the root causes of anomalies at multiple levels of granularity including application-level, service-level, host-level, and metric-level, with the unique capability of code-level localization by harnessing all three types of telemetry data to construct a causal graph representing the intricate, dynamic, and nondeterministic relationships among the various entities related to the anomalies. By implementing and deploying *TrinityRCL* in a real production environment, we evaluate *TrinityRCL* against two baseline methods and the results show that *TrinityRCL* has a significant performance advantage in terms of accuracy at the same level of granularity with comparable efficiency and is particularly effective to support large-scale systems with massive telemetry data.**

***Index Terms*—Root cause, telemetry data, microservices.**

Manuscript received 19 August 2022; revised 17 January 2023; accepted 18 January 2023. Date of publication 1 February 2023; date of current version 16 May 2023. This work was supported in part by the National Key Research and Development Program of China under Grant 2019YFE0105500, in part by the Research Council of Norway under Grant 309494, in part by the Meituan, the Key Research and Development Program of Jiangsu Province under Grant BE2021002-2, and in part by the National Natural Science Foundation of China under Grants 62072227 and 62202219. Recommended for acceptance by D. Hao. *(Corresponding author: Guoping Rong.)*

Shenghui Gu, Guoping Rong, He Zhang, Yongda Yu, and Xian Li are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu 210093, China (e-mail: shenghui.gu@smail.nju.edu.cn; ronggp@nju.edu.cn; hezhang@nju.edu.cn; yuyongda@smail.nju.edu.cn; lixian-go@foxmail.com).

Tian Ren, Jian Ouyang, and Chunan Chen are with the Meituan, Beijing 100102, China (e-mail: rentian@meituan.com; ouyangjian@meituan.com; chunan.chen@meituan.com).

Haifeng Shen is with the HilstLab, Peter Faber Business School, Australian Catholic University, Sydney, NSW 2060, Australia (e-mail: haifeng.shen@acu.edu.au).

Digital Object Identifier 10.1109/TSE.2023.3241299

I. Introduction

In recent years, we have witnessed rapid growth of online services, some of which serve a massive number of users. For example, as of the first quarter of 2022, Facebook has roughly 2.93 billion monthly active users. Most of these systems adopt the microservice architecture as it supports faster delivery, better resilience, and higher adaptability to dynamic customer requirements [1], [2]. To ensure a great customer experience, service monitoring is critical to keep such a system healthy and able to provide stable and continued service. As reported in a systematic survey [3], researchers and practitioners have proposed many anomaly detection and root cause analysis (RCA) methods for service monitoring. It should be noted that RCA varies from the localization of a specific erroneous code snippet to the indication of a general direction worthy further exploration. Most RCA approaches only produce informative clues [4] to the genuine root cause that has to be determined through in-depth manual analysis [3]. Furthermore, compared with the monolithic architecture, the microservices put new requirements on service monitoring and anomaly diagnosis [5].

First, the microservice architecture is usually composed of multiple layers from bottom to top in the order of the hardware layer that comprises the actual machines, the communication layer that houses everything relevant to the application, system, and service communication, the application platform that provides system-wide tools and services, and the microservice layer where the microservices are completely abstracted away from the lower infrastructure layers [6]. Therefore, a microservice system is inherently complex and commonly involves entities at different levels of granularity including the application-level, the service-level (an application is a collection of microservices invoking one another), and the host-level (a physical machine hosts one or more microservices). Correspondingly, as illustrated by Fig. 1 and Table I, due to factors such as data availability and cost-effectiveness, it is important that a microservice RCA solution is able to localize the root causes of anomalies at multiple

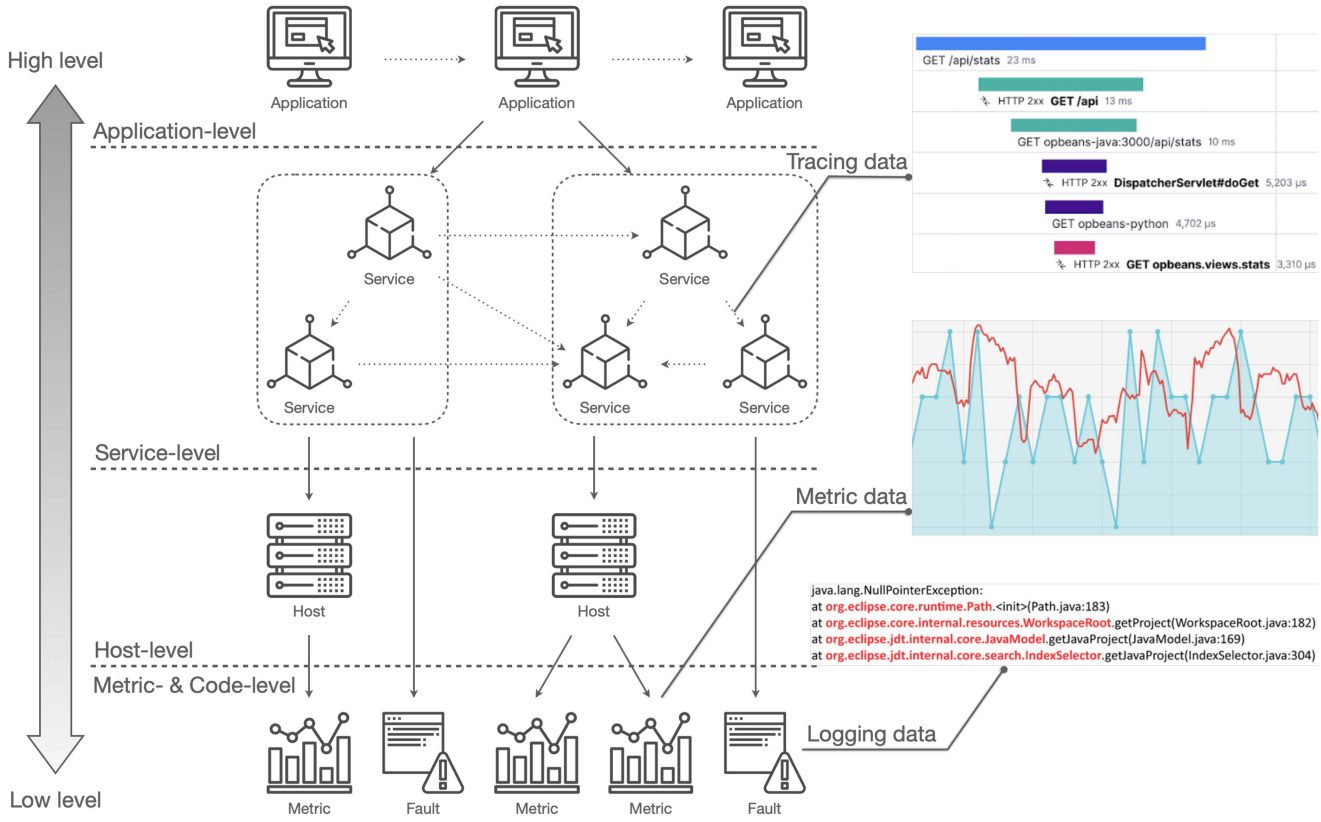


Fig. 1. RCA at multiple levels of granularity and examples of different telemetry data.

TABLE I
DEFINITIONS OF RCA AT MULTIPLE LEVELS OF GRANULARITY

Level	Granularity	Definition
High	Application-level	The RCA can localize anomalous applications.
	Service-level	The RCA can localize anomalous services.
	Host-level	The RCA can localize anomalous hosts/physical servers.
Low	Metric- & Code-level	The RCA can localize anomalous metrics (e.g., CPU usage, response time). The RCA can localize defective source codes.

levels of granularity including those as high as application-level, service-level, and host-level and those as low as metric-level (a specific metric of a host such as CPU usage, response time) and code-level (a specific erroneous code snippet of a microservice running on a host) [3].

It is noteworthy that although localizing a low-level root cause of an anomaly would normally need to first identify high-level ones, an anomaly detected at a high level (service or host) may not always lead to localization of low-level (metric or code) root causes. For example, if there is an intermittent problem with a service external to a microservice system being called, RCA based on the system's internal telemetry data is unlikely to localize the relevant host, metrics, or code of the anomalous external service. Furthermore, there is a non-deterministic relationship between the two low-level RCA. A metric-level anomaly may lead to the localization of defective code if the collected telemetry data provides adequate information or may not necessarily be relevant to faulty code. Likewise, a code-level root cause may or may not exhibit anomalous metrics.

Second, according to a recent industrial survey [7], most anomalies in a microservice system are caused by functional faults. An empirical study based on five open source microservice systems further reveals that general programming errors form the major cause of microservice functional faults [8]. These studies indicate that a large proportion of anomalies in a microservice system pertain to problematic code and as such to timely troubleshoot these anomalies, it is extremely critical that a microservice RCA solution is able to localize the root causes of anomalies at the finest code-level.

In microservice systems, three types of telemetry data are used to perform RCA to service anomalies, which are *metric data*, *logging data*, and *tracing data* [3], [9], [10], as shown in Fig. 1. *Metric data* provides a holistic view of the healthiness of running services and is normally a numeric representation of the data measured over intervals of time [9], which can originate from a variety of sources, e.g., infrastructure, hosts, and services. *Logging data* refers to an immutable, timestamped record of discrete events that happened in a system over time [9] and is usually generated by executing logging statements in running services. *Tracing data* is a representation of a series of causally

related distributed events in a microservice system that can be used to monitor a request from start to end across various components in the system [9].

The majority of microservice RCA approaches are only based on a single type of telemetry data, among which metric-based and tracing-based ones are the mainstream [3]. Metric-based approaches [11], [12], [13], [14] typically utilize metric data to construct causal relationships between the microservices associated with anomalies from which they try to infer the root causes. Usually, they can only identify anomalous applications or services but can not provide adequate information to track down the root causes of the anomalies. Tracing-based approaches [15], [16], [17], [18] mainly use tracing data to model the relationships between anomalous symptoms of several potentially interrelated services (e.g., using call graphs [2], [18], [19]) with which graph-based algorithms (e.g., random walk and breadth-first search) can be employed to perform RCA. However, most of them can only localize the root causes at the application-level or service-level [3].

There are attempts to exploit multiple types of telemetry data in order to improve RCA's efficiency or accuracy. For example, Wang et al. [20] used time series (metric data) and system topology (tracing data) to effectively reduce the analysis time of RCA. Wu et al. [2] used application- and service-levels metrics (metric data) and service call paths (tracing data) to construct a topology graph representing the anomaly propagation in the microservice environments and its accuracy surpassed several state-of-the-art methods. However, to the best of our knowledge, there has been no attempt to support multi-granular RCA by combining multiple types of telemetry data.

Theoretically, by combining the aforementioned three typical types of telemetry data, we may be able to improve system observability [21] for better service monitoring, RCA and service quality assurance. As an anomalous event should leave a "footprint" that can be captured by the three types of telemetry data, a possible RCA approach is to infer vital clues to the anomalous event by aggregating and analyzing information in the "footprint", e.g., timeline, invocation, and causality, which would help localize its root cause. However, such an approach faces a formidable challenge due to the difficulties in making use of sparse information out of voluminous [22], [23], [24] and heterogeneous telemetry data [25], [26], [27], in capturing dynamic and intricate invocation relationships among microservices, and in tracking down nondeterministic propagation of service anomalies. Furthermore, these issues are often entangled, making it even harder to localize the root cause of an anomaly. For instance, as RCA is a posteriori investigation, it may be difficult to determine whether a piece of "footprint" information originates from the anomalous event for which RCA is conducted or from an irrelevant identical anomalous event, which affects the RCA's ability to consolidate a valid clue to the anomaly's root cause.

In this article, we present a new RCA solution referred to as *TrinityRCL*, which exploits all three types of telemetry data for root cause analysis of anomalies at multiple levels of granularity including application-level, service-level, host-level, and metric-level, with the unique ability of localizing root causes at the finest code-level. We overcome the aforementioned challenge by constructing a causal graph to represent the intricate, dynamic, and nondeterministic relationships among the various anomaly-related entities (e.g., services, hosts, metrics, and codes) mined from voluminous and heterogeneous telemetry data. With the causal graph, we further design algorithms to support the localization of root causes.

TrinityRCL has been deployed and evaluated in a real production environment at a world-leading IT company *Meituan*[1]. The results show that: (1) compared with the existing solutions, not only can *TrinityRCL* localize root causes at multiple levels of granularity, but it can also localize them at the finest code-level (e.g., to a code snippet of a microservice on a specific host), and (2) compared with the baseline solutions, it has a significant performance advantage in terms of accuracy at the same level of granularity with comparable efficiency and is particularly effective to support large-scale systems with massive telemetry data. In summary, we make the following contributions in this article:

- We propose a new RCA solution *TrinityRCL* that exploits all three types of telemetry data to support multi-granular and code-level root cause localization.
- The proposed solution mines voluminous and heterogeneous telemetry data to construct a causal graph representing the intricate, dynamic, and nondeterministic relationships among anomaly-related entities.
- With the causality graph, we design algorithms to derive the correlations between the nodes in the graph, which are used to determine a ranking of root causes.
- We implement and deploy the solution in a real production environment and evaluate it against two baseline methods using the dataset collected from the production environment.

The rest of this article is organized as follows. Related work is briefed in Section II. Section III states the studied problem and the associated challenges. Section IV details the *TrinityRCL* approach. In Section V, we describe the experimental evaluation and discuss the related issues in Section VI. The threats to validity are discussed in Section VII. Finally, Section VIII concludes this article with suggestions for future work.

II. Related Work

Significant research endeavor has been devoted to RCA in recently years, some of which has even been marketed as commercial software systems. For example, Dynatrace's Davis AI Causation Engine[2] provides the ability to identify events that share the root cause of an anomaly by utilizing different types of contextual information (telemetry data), e.g., topology, transaction, metrics, and code-level information. However, it is not able to handle most nondeterministic anomaly propagation, which limits its ability to perform multi-granular RCA in microservice systems. In this section, we review statistical RCA methods including metric-based, tracing-based, and logging-based

[1]https://about.meituan.com/en

[2]https://www.dynatrace.com/platform/artificial-intelligence/

approaches as well as multi-source RCA, which are closely related to our work.

A. Metric-Based RCA Approaches

Although metrics provide rich information on the state and trend of a system, it is almost impossible for engineers to manually localize the root cause of an anomaly from metric data due to the gap between metric data and anomalous symptoms. In most cases, the volume of metric data is usually too big for engineers to manually correlate them with anomalous symptoms. Besides, the noises in the metric data make it even harder to manually screen out useful information. Therefore, automatic localization of root causes with metric data has been a focus of research on RCA in recent years. However, it is generally difficult to derive correlations between metric data. When an anomaly occurs, all the associated metrics are jittered due to the existence of indirect anomaly propagation (cf. Section III.B), making it difficult for engineers to locate the anomalous resources (e.g., machines or services).

To overcome the challenges, existing research mainly focuses on constructing causal relationships among microservices. Lin et al. proposed *Microscope*, which found root cause candidates by comparing the similarity between service-level metrics and the abnormal services [11]. Chen et al. proposed *CauseInfer*, which constructed a two-layered hierarchical causal graph of the distributed system [12]. It applied metric data as nodes to indicate service-level dependency, and inferred the root causes of performance problems along the graph using statistical methods. Meng et al. developed a framework named *MicroCause*, which used the Path Condition Time Series (PCTS) algorithm to construct causal graphs based on monitoring metrics in microservices [13]. It then inferred the root cause using the Temporal Cause Oriented Random Walk (TCORW) algorithm. Ma et al. put forward the concept of anomalous behavior graph to describe the correlations between services associated with different types of metrics [14]. Based on the graph, they designed a tool named *AutoMAP*, which enabled dynamic generation of service correlations and automated diagnosis by using multiple types of metrics. Due to the limitations of the data source, these approaches usually infer the causal relationships, which may not be accurate. Moreover, they are not able to diagnose other levels or types of root causes, in spite of their decent performance in localizing the metrics that reflect the root causes.

B. Tracing-Based RCA Approaches

Recent research focuses on three main ways to achieve tracing-based RCA [3]. One way is based on a trace comparison methodology. Historical traces along with RCA results are collected into a database, and when new tracing data comes in, it is matched against the database using similarity algorithms. Brandón et al. presented an RCA framework based on graph similarity, which compared the anomalous graph that happened in the system with a library of anomalous graphs that served as a knowledge base [15].

Another way is based on statistical analysis to automatically determine the possible root causes of anomalies by directly analyzing the status data (e.g., response times) in the service interactions involved in collected traces. Yu et al. designed and implemented *MicroRank* to localize root causes of latency issues in microservice environments [16]. It extracted service latency from tracing data and then inferred the service instances that led to latency issues by combining *PageRank* algorithm and spectrum analysis. Huang et al. proposed *tprof* as a performance profiler that aggregated distributed systems traces to diagnose performance bugs and inefficiencies [17]. It used the structure embedded within tracing data to hierarchically group similar traces and calculate increasingly detailed aggregate statistics to identify the slow parts of the system.

A further way is based on dependency or causal graph models to understand the anomaly propagation path and infer the root cause. Kim et al. proposed an unsupervised algorithm *MonitorRank* that used the historical and current time-series metrics along with the call graph to predict root causes of anomalies in service-oriented architectures by running the *Personalized PageRank* algorithm on the graph [18]. Most tracing-based approaches use metric data as a supplement to provide additional information for RCA. However, these approaches are only able to localize application- or service-level root causes.

C. Logging-Based RCA Approaches

The RCA approaches based on the logging data generated by the services are done by processing the logging data to construct a causal graph where nodes model services and each edge between two nodes models that an anomaly in one service may cause an anomaly in the other. By applying algorithms such as *Random Walk*, the causal graph is then visited to infer potential root causes of an anomaly. Aggarwal et al. presented a golden signal (aka gateway errors that the users face or observe when a system fails) based RCA approach by inferring the causal relationships among services emitting error signals and the one emitting golden signal error [28]. *Random Walk*-based graph centrality approach was used to efficiently localize root causes of the anomalies. Although logging-based approaches are capable of localizing informational causes, the difficulties in log parsing and abnormal information localization from large scale of logs pose great challenges in practice.

D. Multi-Source RCA

Some researchers have attempted to combine multiple sources of telemetry data including tracing data, metric data, and logging data for RCA. Wu et al. presented a system that can infer root causes in real time by correlating application performance symptoms with corresponding system resource utilization metrics [2]. The process of RCA was based on a graph constructed from the invocation relationships to model anomaly propagation across services and machines. Wang et al. proposed an end-to-end anomaly detection and RCA system, which used time series data (metric data) and system topology data (tracing data) to reduce RCA time effectively [20]. They further proposed *Groot*, a graph-based RCA approach [29] that used tracing data and logging data to construct an event causal graph whose basic

nodes were monitoring events such as performance-metric deviation events, status change events, and developer activity events. These events carried detailed information and made it possible to enable accurate RCA. However, existing studies do not fully exploit different types of telemetry data to provide traceable results of RCA. Our *TrinityRCL* approach is particularly designed for providing traceable results to assist with the subsequent failure recovery.

III Problem Statement

In this section, we first describe the problem to be addressed in the article and then elaborate the challenges associated with the problem.

A. Problem Description

We use a structured style to formally describe the problem in this study.

Purpose: To provide multi-granular and code-level RCA for operations staff to timely pin down the genuine root causes of service anomalies.

Approach: By synthesizing all the three types of telemetry data collected by current APM systems.

Context: The proposed RCA is effective to support service monitoring of large-scale microservice systems serving massive users.

B. Challenges Associated With the Problem

The problem stated above comes with major technical challenges outlined in Section I. This section elaborates these challenges.

Difficulty in processing a huge volume of semi-structured telemetry data. Although more telemetry data can potentially provide more valuable information to improve RCA, a massive volume inevitably makes it hard to process and analyze the data [2]. Apart from that, logging data is naturally semi-structured as it highly depends on engineer's experience and expertise, and the heterogeneous nature of telemetry data increases the difficulty in data processing and analysis. In most cases, microservices are developed and maintained by different teams. Although they may adopt similar logging frameworks such as Log4j[3] and Logback[4], the format and content of logging data are usually determined by individual engineers and thus likely to be heterogeneous [25], [26], [30]. One possible reason for this phenomenon is the missing common information needs for the logging data.

Difficulty in capturing intricate and dynamic invocations. Enabling continuous delivery of features motivates the architectural migration from the monolithic to the microservices [31]. While the tracing data captured by APM systems contains the invocation information in a system comprising multiple microservices, the continuous delivery of features supported by various microservices can lead to frequent changes of invocation relationships among the microservices [2]. As a consequence, the ever-changing invocation relationships between microservices are an intrinsic characteristic of a system comprising multiple microservices [16]. In addition, there are other reasons contributing to the dynamic changes of invocation relationships, e.g., flow control, degrading or reallocation of microservice instances, or simply a sudden power outage at the place where the physical computer servers locate. As most machine learning-based and rule-based approaches rely heavily on historical data for model training [20], [32], [33], the constantly changing data may increase the necessity for frequent model updates and inevitably increase the cost of model training.

Difficulty in tracking down nondeterministic anomaly propagation. Anomaly propagation is common in distributed systems and has to be handled by RCA approaches designed for such a system. However, anomaly propagation tends to be nondeterministic in most large-scale microservice systems. On one hand, propagation of service anomalies along explicit service invocation paths may suffer from arbitrary delays when a massive number of microservices are running concurrently, making it difficult to attribute different service anomalies to the identical event source [13]. On the other hand, anomalies can propagate between two co-located microservices even when there is no direct invocation relationship between them [13], [14]. For example, when multiple microservices share the computing resources on the same physical machine, a slow responding service exhausting most computing resources can easily cause the responsiveness of other microservices deployed on the same machine to suffer. As there is no invocation relationship between these services, the tracing data is thus not able to reflect such anomaly propagation.

IV. The TrinityRCL Approach

Fig. 2 provides a holistic view of the the RCA workflow adopted by *Meituan* including (from left to right): (1) the microservice system for business service provisioning, (2) the APM system named *Raptor* for service monitoring, telemetry data collection, and anomaly detection of all the provisioned microservices, and (3) *TrinityRCL* for RCA of detected anomalies with the collected multi-sourced telemetry data. At *Meituan*, due to the massive number of microservices competing for the relatively limited resources, an anomaly usually refers to the abrupt increase or decrease of pre-defined KPIs (Key Performance Indicators), e.g., *"the number and the percentage of successful calls"* to a hot service are typical KPIs supported by *Raptor*. Once an anomaly is detected by the APM system or reported by business staff, *TrinityRCL* is activated to perform RCA and guide operations staff to pin down the most likely root causes.

In this section, we detail each procedure in *TrinityRCL* including (from left to right) data collection, causal graph construction, anomaly score assignment, and root cause localization. It first selects and pre-processes different types of relevant telemetry data during a specified time frame right before the occurrence of the anomaly. The time frame is normally 30 minutes, which is set based on our previous investigation on the aforementioned production environment in *Meituan* [4]. Based on the pre-processed

[3]https://logging.apache.org/log4j/2.x/
[4]https://logback.qos.ch

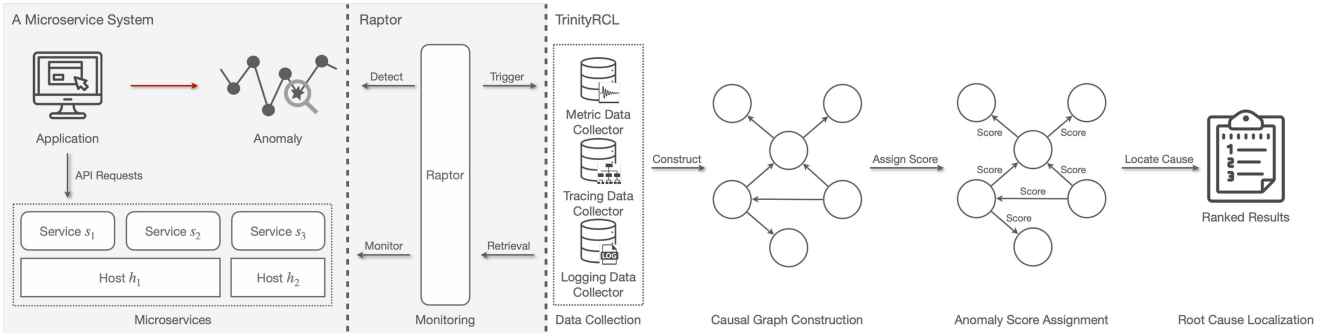


Fig. 2. An overview of the RCA workflow adopted by *Meituan*.

TABLE II
NOTATION LIST

Notation	Description
s_a	The anomalous service.
$s_i \in S$	Service node s_i in the set of service nodes S.
$h_i \in H$	Host node h_i in the set of host nodes H.
H^{s_i}	Set of all host nodes provisioning service s_i.
$m_i \in M$	Metric node m_i in the set of metric nodes M.
M^{h_i}	Set of all metric nodes connected to host node h_i.
$f_i \in F$	Fault node f_i in the set of fault nodes F.
F^{h_i}	Set of all fault nodes connected to host node h_i.
$e_{ij} \in E$	The edge from node i to node j in the set of edges E.
G	The causal graph.
L_i	The number of failures at host node h_i
AS_{ij}	The anomaly score of edge e_{ij}.
R^s_{ij}	The failure rate of invocations from service node s_i to service node s_j.
P_{ij}	The probability of propagating anomaly from node i to node j.
N_t	The times of random walk.
N_s	The steps of each execution of random walk.
c_i	The times that node i is visited.

TABLE III
INFORMATION COLLECTED FOR EACH TYPE OF TELEMETRY DATA

Information	Description
Tracing data	
# total invocations	The number of overall invocations from one service to another.
# failed invocations	The number of failed invocations from one service to another.
Metric data	
Time series data	Time series data of each metric.
Logging data	
Timestamp	Time when the event occurred measured by the clock.
Service identifier	The identifier of the service where the event happened.
Host identifier	The identifier of the host where the event happened.
Severity	The representation of the severity (aka log level), e.g., ERROR, which is used to screen out events containing faults.
Log messages	The body of a log entry, which may contain exception stacks and provide important information for tracking back to the relevant source code.
# executions	The execution count of the event, which is accumulated by a counter whenever the target event is triggered.

telemetry data, *TrinityRCL* then constructs a causal graph able to reflect the status and the propagation of related services. Finally, *TrinityRCL* calculates and assigns various anomaly scores to each node in the causal graph and infers which node is most likely to cause the anomaly. In what follows, we elaborate the main procedures of *TrinityRCL* in detail. For the clarity of presentation, we list all the notations used in this article in Table II.

A. Data Collection

As the starting point of RCA, data collection aims to aggregate different types of telemetry data to provide a basis for later analysis. Apparently, the three types of telemetry data (i.e., *tracing data*, *metric data*, and *logging data*) have different characteristics and are normally used for different operations tasks. In a real production environment, it is not rare that redundant or misleading information in telemetry data may cover up actual root causes [26], [34], making it hard to perform RCA. Therefore, it is critical to pre-process the telemetry data before feeding it to the subsequent procedures of *TrinityRCL*. Table III lists the information that needs to be collected for each type of telemetry data. Then by applying a 30-minute time frame, all the three types of relevant telemetry data are selected and pre-processed by *TrinityRCL*. In the following paragraphs, we elaborate how each type of telemetry data is processed in *TrinityRCL*.

1) Tracing Data: Tracing data is the central pillar in the construction of a causal graph in *TrinityRCL*, which underpins the formation of the graph's skeleton. When an anomaly occurs, *TrinityRCL* starts from the anomalous service s_a and retrieves the services that failed to invoke s_a and those which s_a failed to invoke based on the tracing data. Then *TrinityRCL* recursively searches for the related services according to the retrieved list of services using Algorithm 1. In a production environment, the number of related services could be enormous, and we thus limit the depth of the recursive search to avoid the exponential growth of the number of services, as the likelihood of a service becoming the root cause decreases with the depth. Based on the collected data (cf. Table III), the failure rate of invocations, i.e. the percentage of failed invocations out of the total invocations,

Algorithm 1: The OCF Algorithm.Recursive Search for Anomaly-Related Services

Data: Anomalous service node s_a
Result: List of anomaly-related service nodes L_a
$L_a \leftarrow$ *empty list*;
$L_v \leftarrow$ *empty list*; `// List of visited nodes`
Function `Retrieve(`*node* s_i`)` **is**
  if $s_i \in L_v$ **then**
    return;
  end
  Add s_i to L_a ;
  Add s_i to L_v ;
  for *edge* $e_{ij} \in$ *out-edges of* s_i **do**
    if e_{ij} *has failed invocations* **then**
      `Retrieve(`s_j`);`
    end
  end
  for *edge* $e_{ki} \in$ *in-edges of* s_i **do**
    if e_{ki} *has failed invocations* **then**
      `Retrieve(`s_k`);`
    end
  end
  return;
end
`Retrieve(`s_a`);`
return L_a;

is computed for further use during the anomaly score assignment procedure.

2) Metric Data: Metric data can be collected by the various agents of the APM system deployed on the hosts of microservices. *TrinityRCL* treats all metric data as time-series data, e.g., the CPU usage collected at one-minute intervals from 12:00 to 12:30. However, the time series data collected from a production environment may inevitably contain missing values. Therefore, we apply some strategies to pre-process the metric data, e.g., using the average value of the adjacent data to interpolate the missing data. As most APM systems perform anomaly detection through metric data, anomaly data reflecting the symptoms of an anomaly is also a type of time series metric data. Therefore, *TrinityRCL* adopts the same strategy to handle anomaly data as it does for metric data. Finally, the collected anomaly data is aggregated into specific time span buckets of usually one minute in *TrinityRCL*.

3) Logging Data: Logging data is generated by logging statements instrumented by the developers responsible for the development and operations of microservices. To ensure the validity of logging data, each entry is required to record the timestamp, service identifier, host identifier, severity, log messages, and number of executions, as listed in Table III. The information collected is similar to the log data model (semantic conventions) defined by *OpenTelemetry*[5], which is readily available through most major logging tools (e.g., Log4j). In *TrinityRCL*, the timestamp and execution count of the log entries

[5]https://opentelemetry.io/docs/reference/specification/logs/data-model/

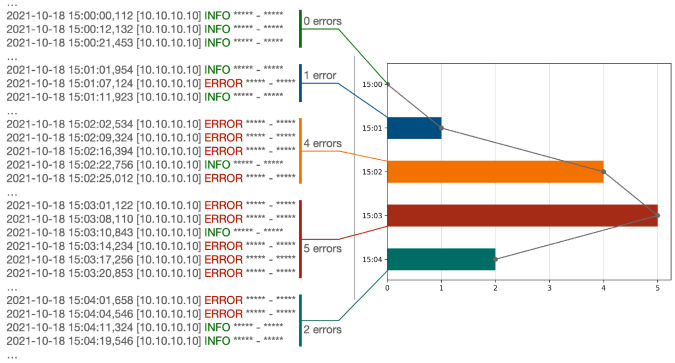


Fig. 3. An example of converting logging data to time series data.

are also converted into time series data by aggregating the count of log entries associated with failures into time buckets based on the timestamp of each log entry, as illustrated by Fig. 3.

B. Causal Graph Construction

To support multi-granular RCA, *TrinityRCL* combines multiple types of telemetry data to construct a causal graph that is able to represent the runtime states of microservices. The skeleton of the graph is based on the tracing data and complemented by the other two types of telemetry data. With the causal graph, once a root cause is localized, the analysis result can be traced from the root cause to the anomaly. To reflect the dynamics of microservices, the causal graph is constructed using the telemetry data retrieved in a 30-minute time frame right before an anomaly occurs instead of using the historical data from static analysis. As mentioned in Section III.B, anomalies may propagate not only among the services along the service call paths, but also to the services co-located on the same machines [19], [35]. To deal with the diversity of anomaly propagation, *TrinityRCL* introduces different types of nodes in the causal graph to comprehensively characterize the runtime state of an entire microservice system. A causal graph consists of the following types of nodes.

- *Service nodes* ($S = s_1, s_2, \ldots, s_k$) represent specific microservices.
- *Host nodes* ($H = h_1, h_2, \ldots, h_k$) represent hosts provisioning one or more microservices.
- *Metric nodes* ($M = m_1, m_2, \ldots, m_k$) represent metrics that reflect the state of a host.
- *Fault nodes* ($F = f_1, f_2, \ldots, f_k$) represent unexpected exception stack that occurs on a host.

Fig. 4 depicts the process of constructing a causal graph in three steps.

Step 1 *Recover invocation relationships between service nodes.* If there is a failed invocation from service nodes s_i to s_j, we add a directed edge from s_i to s_j in the causal graph. The failure rate of the invocation is also recorded at this step. The causal graph in Fig. 4(a) shows the invocation relationships among microservices s_1, s_2, s_3.

Step 2 *Establish hosting relationships between service nodes and host nodes.* If service s_i is allocated to host h_j

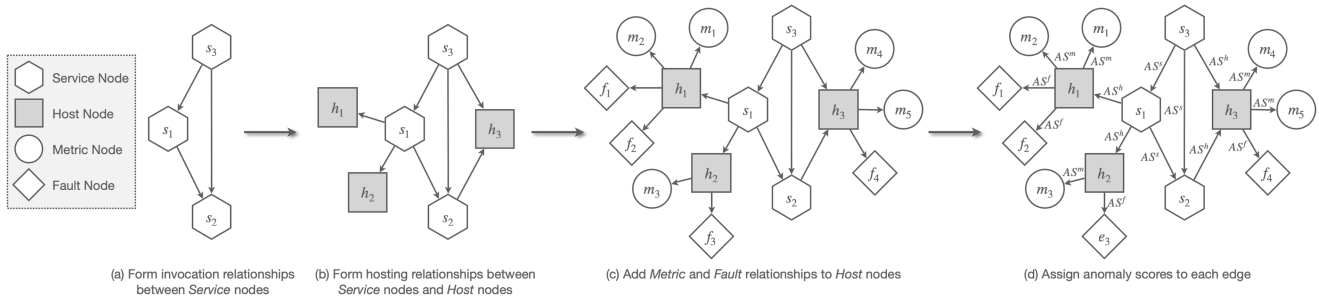


Fig. 4. An example of causal graph construction.

for provisioning, we add a directed edge from s_i to h_j in the causal graph. The causal graph in Fig. 4(b) shows the relationships between hosts h_1, h_2, h_3 and microservices s_1, s_2, s_3.

Step 3 *Add metric and fault relationships to host nodes.* The runtime state of each host h_i is monitored by multiple metrics. For each metric node m_j that records the corresponding type of metric data related to the anomaly, we add a directed edge from host h_i to metric m_j in the causal graph. One or more faults may occur on a host h_i. For each fault node f_j that records the error messages, e.g., exception stacks, we add a directed edge from h_i to f_j in the causal graph. The causal graph in Fig. 4(c) shows metric nodes and fault nodes are attached to host nodes.

C. Anomaly Score Assignment

Once a causal graph is constructed, we proceed to assign an anomaly score to every edge in the graph that reflects the correlation between each node to the anomaly node, as shown in Fig. 4(d). Each anomaly score represents the probability of the anomaly transferring from one node to another and is the basis of the subsequent random walk (cf. Section IV.D). In this procedure, *TrinityRCL* assigns anomaly scores by calculating the correlation coefficients between the connected nodes. For different types of nodes, we apply different correlation coefficient algorithms optimized to balance the effectiveness and the performance of analysis. As discussed in Section III.B, propagation delays can influence the accuracy of correlations between metric data. Therefore, in *TrinityRCL*, we choose algorithms that can quantify synchrony between time series data including:

- *Dynamic Time Warping (DTW)* [36]. It measures the optimal matching (similarity) between two time series that do not synchronize perfectly. Originally devised for speech analysis, *DTW* computes the euclidean distance at each frame across every other frames to compute the minimum path that will match the two time series. Due to the propagation delays of metric data, *DTW* is suitable for describing the difference between time series of two metrics.
- *The First Order Temporal Correlation (CORT)* [37]. It indicates how two time series data co-vary over time, and measures the features of monotonicity and growth rate. The value of *CORT* means the two time series increase or decrease simultaneously with the same growth rate (similar behavior). As the time series data value of a fault node is zero in most cases, *CORT* is adequate to calculate the correlation (based on two time series data) between a fault node and an anomaly. Compared with *DTW* which has a time complexity of $O(n^2)$, *CORT* is more efficient as it has a lower complexity of $O(n)$.

In particular, anomaly scores are calculated in the following ways:

- *Anomaly score of the edge from a host node to a metric node* AS^m_{ij}. For the edge from host node h_i to metric node m_j, we compute *DTW* correlation coefficient (denoted as $corr_{dtw}(m_j, s_a)$) between the time series data of m_j and that of anomalous node s_a.
- *Anomaly score of the edge from a host node to a fault node* AS^f_{ij}. For the edge from host node h_i to fault node f_j, we compute *CORT* correlation coefficient (denoted as $corr_{cort}(f_j, s_a)$) between the time series data of f_j and that of anomalous node s_a.
- *Anomaly score of the edge from a service node to a host node* AS^h_{ij}. For the edge from service node s_i to host node h_j, we first calculate the maximum correlation coefficient among the edges from h_j to all its metric nodes M^{h_j} and all its fault nodes F^{h_j}. We then calculate the ratio of the number of failures in h_j to the number of failures in all hosts H^{s_i} provisioning service s_i, which is denoted by R^h_{ij}. Finally, AS^h_{ij} is assigned as the weighted sum of the failure ratio and the aforementioned maximum correlation coefficient and formulated in (1), where α is a constant denoting the weight of the failure ratio R^h_{ij} and β is another constant denoting the weight of the maximum correlation coefficient of all fault nodes. Such an anomaly score is a weighted sum of the maximum correlation coefficient of all metric nodes and fault nodes under host node h_j and the failure ratio of h_j. An example of such an anomaly score is given in Section IV.E.

$$AS^h_{ij} = (1-\alpha)\left[(1-\beta)\max_{m_k \in M^{h_j}} AS^m_{jk} + \beta \max_{f_l \in F^{h_j}} AS^f_{jl}\right] + \alpha R^h_{ij} \quad (1)$$

where

$$R^h_{ij} = \frac{L_{h_j}}{\sum_{k \in H^{s_i}} L_k}$$

Algorithm 2: Anomaly Score Assignment.

Data: Original causal graph G, anomalous nodes
Result: Updated causal graph G with annotated edges

```
Function Assign(node n_i) is
    for edge e_ij ∈ out-edges of n_i do
        if e_ij is not assigned with a score then
            Assign(n_j);
            e_ij ← AS_ij; // cf. Eq. (3)
        end
    end
    for edge e_ki ∈ in-edges of n_i do
        if e_ki is not assigned with a score then
            Assign(n_k);
            e_ki ← AS_ki; // cf. Eq. (3)
        end
    end
    return;
end
for service node s_i in anomalous nodes do
    Assign(s_i);
end
return Updated G with annotated edges;
```

- *Anomaly score of the edge between two service nodes AS^s_{ij}.* For the edge from service node s_i to s_j, we calculate the maximum correlation coefficient among the edges from s_j to all host nodes H^{s_j} provisioning the service s_j. Then we calculate the failure ratio of invocations from s_i to s_j (cf. Section IV.A.1), which is denoted by R^s_{ij}. AS^s_{ij} is assigned as the weighted sum of the failure ratio of invocations and the aforementioned maximum correlation coefficient. The equation is formulated in (2), where γ is a constant denoting the weight of the failure ratio R^s_{ij}. An example of such an anomaly score is given in Section IV.E.

$$AS^s_{ij} = (1-\gamma) \max_{h_k \in H^{s_j}} AS^h_{jk} + \gamma R^s_{ij} \tag{2}$$

To summarize, the anomaly score AS_{ij} of the edge from nodes i to j is determined by (3) shown at the bottom of this page. It is worth noting that the time series data used to calculate the anomaly scores has been normalized in order to make all anomaly scores comparable.

The procedure of anomaly score assignment is presented in Algorithm 2. At the end of this procedure, we obtain a causal graph whose edges are annotated with the assigned anomaly scores.

D. Root Cause Localization

Although anomaly scores are useful for localizing the potential root causes, they do not necessarily imply causality [18] and should not be the only factor used to determine whether a node has caused an anomaly as root causes usually have a high chance of propagation. A microservice with more abnormal than normal traces passing through it is more likely to be the root cause [38]. To simulate the process of anomaly propagation, we localize root causes from the constructed causal graph with edges annotated by anomaly scores via *Random Walk with Restart (RWR)* [39]—a variant of *Random Walk* algorithm, which has been widely used for root cause localization [18], [19], [40]. As we only focus on each node's relevance w.r.t. the anomalous node, we choose *RWR* that could restart only from the assigned node, i.e. the anomalous node. In general, there are three stages involved in *RWR*.

Stage 1: Transfer Probability Calculation. In this step, we calculate the transfer probability of each node based on the anomaly scores assigned to each edge (cf. Section IV.C), which is detailed as follows:

1) *Forward step:* The algorithm walks from the anomalous node to cause nodes. The transfer probability P_{ij} from nodes i to j equals to the anomaly score AS_{ij} of the edge between the two nodes.
2) *Backward step:* The walker may be trapped in nodes having low correlations with the anomalous node. As there is no way to escape out until the next random walk, we implement a step from the cause node to the anomalous node to avoid local optimum. Formally, if $e_{ji} \in E$ and $e_{ij} \notin E$, $P_{ji} = \rho AS_{ij}$, where $\rho \in [0, 1]$ is a parameter controlling the impact of the backward step.
3) *Self step:* By definition, the walker is enforced to move to another nodes even if the current node has a higher anomaly score while all the neighboring nodes do not. This transfer increases the probability of identifying nodes unrelated to the given anomaly. To avoid this, the walker tends to stay longer at the same node if all its neighboring nodes have lower anomaly scores. Specifically, for each node, the probability of staying equals to the maximum anomaly score of upstream edges subtracting that of downstream edges, which is formulated as $P_{ii} = \max(0, \max_{k:e_{ki}\in E} AS_{ki} - \max_{l:e_{il}\in E} AS_{il})$.

To accelerate the random walk, we apply the alias method [41], which picks a next node to walk into in $O(1)$ time complexity.

Stage 2: Random Walk Execution. With transfer probabilities, we start *RWR* from the anomalous service s_a. The algorithm executes walking for N_t times. For each random walk, the walker stops after N_s steps. In the end, node i is visited c_i times.

$$AS_{ij} = \begin{cases} corr_{dtw}(m_j, s_a) & if\ h_i \in H\ and\ m_j \in M^{h_i} \\ corr_{cort}(f_j, s_a) & if\ h_i \in H\ and\ f_j \in F^{h_i} \\ (1-\alpha)\left[(1-\beta)\max_{m_k \in M^{h_j}} AS^m_{jk} + \beta \max_{f_l \in F^{h_j}} AS^f_{jl}\right] + \alpha R^h_{ij} & if\ s_i \in S\ and\ h_j \in H^{s_j} \\ (1-\gamma)\max_{h_k \in H^{s_j}} AS^h_{jk} + \gamma R^s_{ij} & if\ s_i, s_j \in S \end{cases} \tag{3}$$

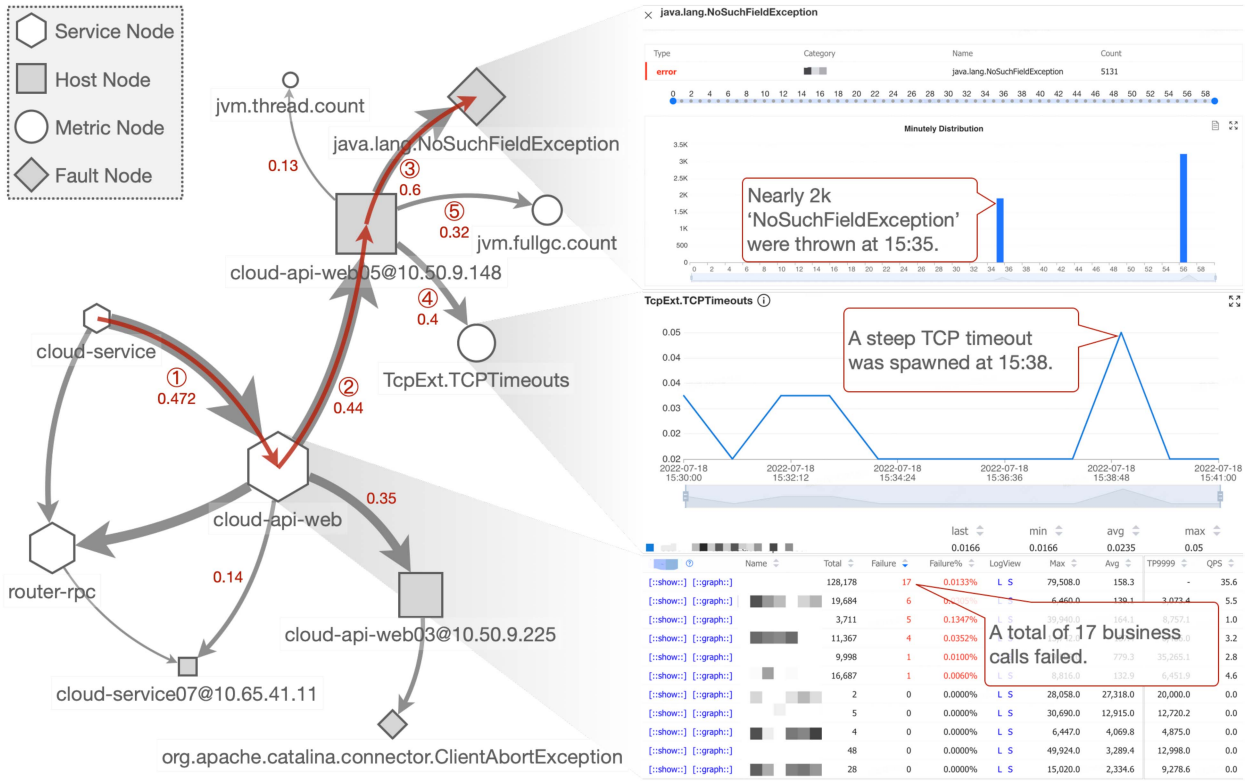


Fig. 5. A live multi-granular RCA case using *TrinityRCL*.

Stage 3: Root Cause Ranking. The final step is to rank the result of the random walk execution. As there are different types of nodes in the causal graph, *TrinityRCL* groups the results according to the type of nodes and arranges the results of each group in descending order according to c_i, i.e. the number of times node i is visited. In the end, *TrinityRCL* provides an interactive web interface that illustrates the route of anomaly propagation with detailed anomaly-related information to facilitate engineers to determine the root cause of the anomaly.

E. A Live Multi-Granular and Code-Level RCA Case

Fig. 5 demonstrates a live case where *TrinityRCL* assists the operations staff at *Meituan* in probing the root causes of an anomaly. The process of RCA is triggered when an anomaly relating to the abnormal behavior of "cloud-service" is detected by the *Raptor* system (see Fig. 2) or directly reported by the business staff. In this case, one abrupt decrease of a monitored KPI (i.e., success rate of order requests drops from $\geq 99.99\%$ to $< 99\%$) has occurred. As the KPI is pertinent to a core service handling orders, the *Raptor* system captures this event according to the configuration settings by the users in *Meituan*. Then, *TrinityRCL* is activated to construct a causal graph with the multi-sourced telemetry data collected by *Raptor*. In the graph, different shapes represent different levels of granularity and each arrow delineates a possible anomaly propagation route (such a route is based on a logic relationship and does not necessarily require the existence of a real and explicit invocation), while the shape and arrow sizes denote the probabilities of anomaly propagation routes determined by their anomaly scores (a bigger size depicts a higher probability). For example, the maximum anomaly scores of all the metric nodes and fault nodes from the host node of "cloud-api-web05@10.50.9.148" are 0.4 and 0.6, respectively. Then the anomaly score of propagation route *arrow*② is $0.44 = (1-0.2) * ((1-0.5) * 0.4 + 0.5 * 0.6) + 0.2 * 0.2$ when the failure ratio of this host node is 0.2, and α and β are set to 0.2 and 0.5, respectively. Since this host node has the maximum anomaly score among all the hosts provisioning service node "cloud-api-web", the anomaly score of propagation route *arrow*① is $0.472 = (1-0.2) * 0.44 + 0.2 * 0.6$ when its failure ratio of invocations is 0.6 and γ is set to 0.2. Generally, the higher the anomaly score of a node is, the more probable a random walker will walk into it (cf. Section IV.D). For the sake of simplicity, yet without loss of generality, *TrinityRCL* only suggests the top 3 most probable propagation routes from each node for operations staff's perusal in this example, but determination of k in top-k ranking can be configured in *TrinityRCL*. It is noted that there can be dozens or even hundreds of propagation routes from each node in a large microservice system and it is difficult, if not entirely impossible, for operations staff to assess each one of them.

By referring to the most probable propagation route *arrow*① from the service node where the anomaly occurs in the visual causal graph, the operations staff can first localize the service-level root causes at the service node of "cloud-api-web" which shows a total of 17 failed calls. Then from this service node, the most probable propagation route is *arrow*②, which points to the host node of "cloud-api-web05@10.50.9.148" where

host-level root causes are likely located. Subsequently from this host node, propagation route *arrow*③ points to the fault node of "java.lang.NoSuchFieldException", which contains information to localize the code-level root causes comprising the corresponding code lines in the source code extracted from relevant error logs. Also from the same host node, propagation routes *arrow*④ and *arrow*⑤ point to the metric nodes of "TcpExt.TCPTimeouts" and "jvm.fullgc.count" that reveal a steep TCP timeout and throws nearly 2000 'NoSuchFieldException' exceptions in a short period of time, respectively, both crucial for localizing metric-level root causes. Finally, through the multi-granular RCA supported by *TrinityRCL*, the operations staff can quickly pin down the actual root cause of the detected anomaly at the service node of "cloud-api-web", which is a new version of source code with updated fields (attributes) and a new object deployed at the host node of "cloud-api-web05@10.50.9.148". Another method accessing this object using the reflection mechanism that has not been synchronized creates a large number of exceptions and leads to a steep TCP timeout, hence devouring the computing resources at this host node and further causing the detected anomaly at the service node. A back-and-forth exploration further reveals that services on more than 20 hosts accessing this new version of source code/object present various anomalous behaviors.

V. Experimental Evaluation

In this section, we evaluate *TrinityRCL* in a real production environment by benchmarking its performance against two state-of-the-art baseline methods in terms of accuracy and efficiency. The two methods are *MicroCause* [13] that only supports metric-level RCA using temporal cause oriented random walk and *MicroRCA* [2] that only supports service-level RCA using causal graph and random walk. Both methods utilize multiple types of telemetry data and adopt similar techniques to those of *TrinityRCL* such as causal graph and random walk. More importantly, both methods achieve relative higher accuracy than others.

A. Experiment Setup

In the following subsections, we elaborate the production environment, the testbed, and the dataset for the experiment, as well as the baseline methods and the metrics used in the evaluation.

1) Production Environment and Testbed: Meituan is one of the world-leading and largest online services providers, serving tens of millions of users per day. To guarantee a great customer experience and the stability of the services, engineers have developed an APM system named *Raptor* to monitor the status of these online services. Especially after a major technological transformation from the monolithic to the microservice architectures, the *Raptor* system monitors over ten thousand distributed microservices and creates over one petabyte of telemetry data each day.

Apparently, having a huge amount of data does not necessarily mean that the root causes of service anomalies will automatically pop up. Engineers need to rely on tools [42] and their personal domain knowledge and experience to search for clues and further localize the root causes, which is neither efficient nor effective. They require an effective tool that can not only localize root causes at multiple levels of granularity but also perform RCA at the finest code-level (better to be code snippets) to be able to efficiently identify and confirm the root causes. Both accuracy and efficiency are important for an RCA solution to be pragmatically adopted in a real production environment.

TABLE IV
Hardware and Software Configuration in the Testbed

Component	Data (traces, metrics, logs) collector	Algorithm executor
Operating system	CentOS Linux release 7.1.1503	macOS Monterey 12.0.1
CPU(s)	64 (Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz)	6 (Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz)
Memory (GB)	512	16 (2667 MHz DDR4)

We evaluate *TrinityRCL* in the production environment at *Meituan*, where more than 100,000 microservices are provisioned. For different types of telemetry data, distinct components of the *Raptor* system are deployed to collect and pre-process the data, as elaborated in Section IV.A. In addition, one server is used to execute the RCA algorithms. Table IV describes the hardware and software specifics of the testbed.

2) Baseline Methods: In our experiments, N_t (the times of random walk) is set to 1,000,000 and N_s (the steps of each execution of random walk) is set to 100. We compare *TrinityRCL* against the following two baseline methods:

- *MicroCause* [13]: It is a framework for localizing root causes in a microservice. Different from *TrinityRCL*, it only uses metric data to construct a causal graph through causal inference. Therefore, we only use collected metric data to build the causal graph for proper comparison. Similarly, they use a novel temporal cause oriented random walk to identify root causes.
- *MicroRCA* [2]: It is another graph-based method to localize root causes in microservice environments. To construct a causal graph, they use application- and service- levels metrics, as well as service call path data. For the random walk algorithm, it applies *Personalized PageRank* [43] to infer root causes. To implement *MicroRCA*, we construct the causal graph by replacing the response time used in *MicroRCA* with the failure rate, for the reason that *TrinityRCL* does not locate performance issues as *MicroRCA* does.

To our best knowledge, there is currently no method that exploits all three types of telemetry data to provide multi-granular and code-level RCA and as such are comparable to *TrinityRCL*. The consideration to select the baseline RCA approaches is three-fold. First, we prefer only replicable approaches, meaning that detailed information to replicate the core algorithm should be attainable. For example, as a promising RCA method, *Groot* [29] also uses both tracing and logging data to achieve better RCA accuracy. However, due to confidentiality restrictions, its implementation details can not be accessible for replication.

TABLE V
THE BASIC INFORMATION ABOUT THE 30 EXPERIMENTAL CASES

Root cause	# cases	# cases *TrinityRCL* is able to localize at least one correct root cause at different levels		Example
Defective code	15	Code Metric Host Service	13 10 15 15	A new version of source code containing updated fields (attributes) and a new object deployed at a host caused another method from a different host accessing this object using the reflection mechanism to throw a massive number of "NoSuchFieldException" exceptions, resulting in a steep TCP timeout within a few seconds.
Metric fluctuations	10	Code Metric Host Service	10 8 10 10	An unknown event triggered massive garbage collection by the Java Virtual Machine, consuming nearly 100% of the CPU resources and leading to timeout error for most incoming services during the same time period.
Host anomalies†	2	Host Service	2 2	Several services located on a host presented anomalous behaviors, but we are not able to localize deeper root causes with the available telemetry data.
Service anomalies†	3	Service	3	An intermittent problem with a service external to *Meituan*'s microservice system was identified, however, the telemetry data collected from the system may not be adequate to support localizing the deeper root causes of the anomalous external service.

† In most cases, "Host" and "Service" provide intermediate RCA results. However, *TrinityRCL* may not be able to perform further RCA due to reasons such as limited telemetry data or unavailable external data.

Second, we prefer the approaches using multiple types of telemetry data as more types can provide more useful information, which will in turn potentially improve the performance of RCA. Third, among the similar approaches (e.g., *MonitorRank* [18], *Microscope* [11], *CloudRanger* [40], *TON18* [19]) identified from a recently systemic survey [3] and our investigation, the accuracy of both *MicroCause* and *MicroRCA* are relatively higher than others [2], [13].

3) Dataset: The dataset is based on the data from the real production environment in 2021, including 30 different randomly selected cases from the hottest product lines in *Meituan*, covering all the three types of telemetry data. Each case is associated with one reported anomaly, which contains a large volume of telemetry data collected from over 100 services, over 50 hosts, and over 100 metrics on each host with a time span of 30 minutes. Take tracing data as an example. If one service invocation creates one entry, there are easily over 1.5 million entries for one case alone. The types of metrics can be roughly classified into the following categories: CPU related metrics, memory related metrics, host load related metrics, network related metrics, kernel related metrics, and Java Virtual Machine (JVM) related metrics. The formats of the tracing, metric and logging data follow the *OpenTelemetry* data models[6]. All these cases had been checked and analyzed by professional operations staff in *Meituan* to identify the anomalies and the corresponding root causes, among which 11 cases were related to problematic source code. Note that one anomaly may be caused by more than one root cause. Table V lists the number of cases categorized by their actual root causes and the number of cases that *TrinityRCL* is able to localize at least one correct root cause at each level of granularity among the 30 experimental cases. One intuitive root cause example for each granularity level is also listed in this table.

4) Evaluation Metrics: To quantify the performance of *TrinityRCL* and the baseline methods, we introduce two most commonly used metrics used in recent works [13], [14], [40], and one metric to evaluate the efficiency.

[6]https://opentelemetry.io/docs/reference/specification/

- ***AC@k*** represents the probability that the top k ranking results given by an algorithm include the actual root causes for all given cases. The algorithm localizes the true root causes more accurately with a smaller k and a higher $AC@k$. Formally, $AC@k$ is defined on a set of given failure case $\mathbb{A}$ as:

$$AC@k = \frac{1}{|\mathbb{A}|} \sum_{a \in \mathbb{A}} \frac{\sum_{i<k} R^a[i] \in V^a_{rc}}{\min(k, |V^a_{rc}|)} \tag{4}$$

  where $R^a[i]$ is the rank of each root cause given by an algorithm and V^a_{rc} is the root cause set for failure case a.
- ***Avg@k*** denotes the overall performance of an algorithm by computing the average $AC@k$. It can be formulated as:

$$Avg@k = \frac{1}{k} \sum_{1 \leqslant j \leqslant k} AC@j \tag{5}$$

- ***T*** denotes the time required for localizing the root cause by an algorithm and is used to evaluate the execution efficiency of an algorithm.

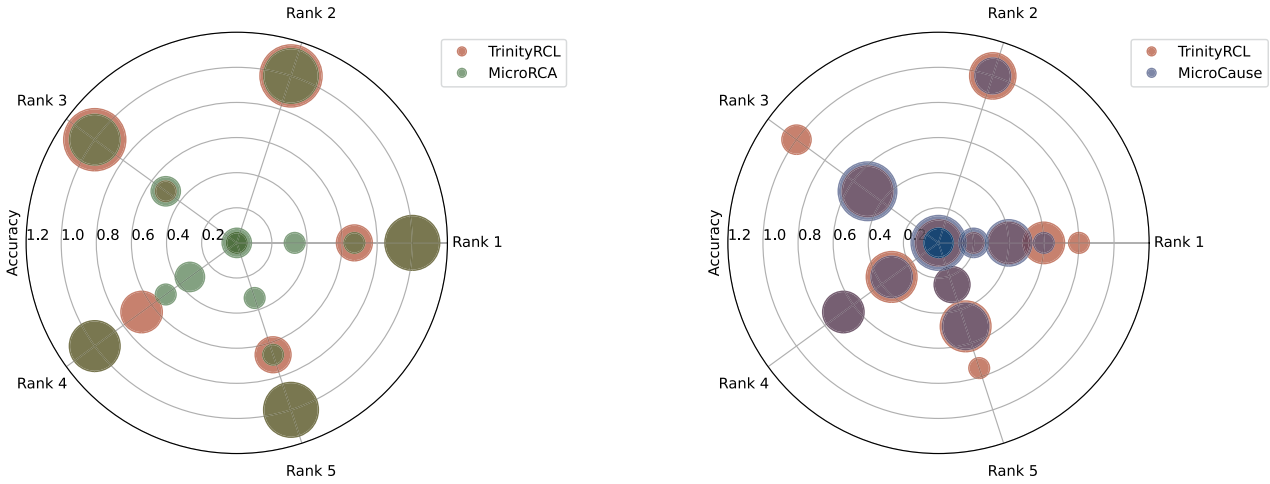
B. Experimental Results

Table VI shows the overall results of *TrinityRCL* and the baseline methods for different types of cases, including service-level, host-level, metric-level, and code-level cases. Note that since no specific external service has been determined, we deem the corresponding analysis as failed RCA to calculate $AC@k$. Meanwhile, for the rest levels, we only use the data with *Meituan* to calculate $AC@k$. As introduced in Section IV.B, each of these cases reflects four different levels of granularity. For example, service-level cases evaluate the performance of methods to localize service-related anomalies. It can be seen that *TrinityRCL* can achieve a top-1 accuracy of at least 80% in localizing service-, host- and code-level anomalies, while it has a relatively low accuracy in identifying metric-level anomalies.

Meanwhile, as shown in Table VI, the comparison between *TrinityRCL* and the baseline methods also unravels favorable results for *TrinityRCL*. As not all the baseline methods can

TABLE VI
ACCURACY OF *TRINITYRCL* AND THE BASELINE METHODS

	TrinityRCL				*MicroCause*				*MicroRCA*			
	$AC@1$	$AC@3$	$AC@5$	$Avg@5$	$AC@1$	$AC@3$	$AC@5$	$Avg@5$	$AC@1$	$AC@3$	$AC@5$	$Avg@5$
Service-level	0.90	0.87	0.90	0.90	/	/	/	/	0.78	0.80	0.89	0.83
Host-level	0.95	0.93	0.92	0.94	/	/	/	/	/	/	/	/
Metric-level	0.50	0.47	0.50	0.48	0.30	0.40	0.30	0.35	/	/	/	/
Code-level	0.82	0.73	0.74	0.77	/	/	/	/	/	/	/	/



(a) Comparison of *TrinityRCL* and *MicroRCA*.

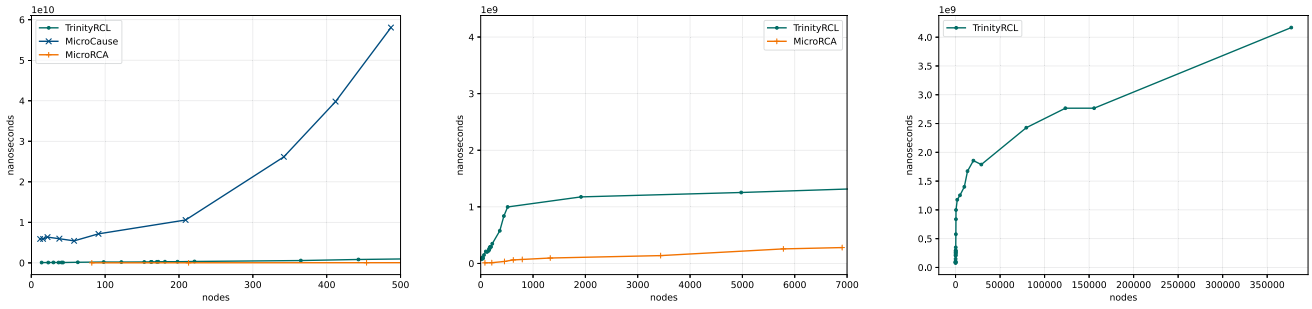
(b) Comparison of *TrinityRCL* and *MicroCause*.

Fig. 6. Root cause rank against accuracy.

localize root causes at each of the four levels, we perform the comparisons separately. In particular, as *MicroRCA* only supports service-level RCA, we compare *TrinityRCL* and *MicroRCA* in terms of $AC@1$, $AC@3$, $AC@5$ and $Avg@5$ on all service-level cases. Likewise, as *MicroCause* only supports metric-level RCA, the performance comparison between *TrinityRCL* and *MicroCause* in terms of $AC@1$, $AC@3$, $AC@5$ and $Avg@5$ is conducted on metric-level cases. We can observe that *TrinityRCL* outperforms *MicroRCA* in terms of both $AC@k$ and $Avg@5$, which are up to 15.7% and 9.3% higher, respectively. *TrinityRCL* has a higher top-1 accuracy than *MicroRCA* does, which means *TrinityRCL* can localize the true root causes more accurately. Similarly, we can observe that *TrinityRCL* also outperforms *MicroCause* both in terms of $AC@k$ and $Avg@5$, which are up to 66.7% and 40.0% higher, respectively. Both comparisons result in less than 0.05 p-value, indicating that the null hypothesis (i.e., no significant difference between two sets of data) can be rejected. It can be noticed that the accuracy of both *TrinityRCL* and *MicroCause* on metric-level cases is not as high as on other types of cases. The reason behind this may be due to the fact that the correlation between the metric data does not reveal causality.

To evaluate the sensitivity to the performance of RCA, we count the number of different ranks of all experimental cases against root cause localization accuracy (i.e., $AC@k$) for *TrinityRCL*, *MicroCause*, and *MicroRCA*, as shown in Fig. 6. In each figure, the more and the larger scatters are near the outer side, the more accurate the method is. It can be seen that both *TrinityRCL* and *MicroRCA* show good performance on service-level anomalies. However, *TrinityRCL* has a higher accuracy when not all root causes are localized. The overall performance of both *TrinityRCL* and *MicroCause* on metric-level cases is not as good as that on service-level cases, but *TrinityRCL* still has a higher accuracy when not all root causes are localized.

To evaluate the efficiency of *TrinityRCL*, *MicroCause*, and *MicroRCA*, we recorded the time required for localizing the root cause (T) of each case. Fig. 7(a) compares T among all the three methods from which it can be observed that *MicroCause* requires significantly more time to localize a root cause compared to the other two methods. In our experiments, *MicroCause* cannot work for those cases with a large number of nodes (over 500 nodes) in the causal graph, ending up with execution timeouts due to the massive computation of partial correlations among the metric data. Fig. 7(b) compares T between *TrinityRCL* and *MicroRCA* from which we can notice that *MicroRCA* took less time to localize the root cause compared to *TrinityRCL*. This is because *MicroRCA* used Pearson correlation coefficient [44] to represent the correlation between service nodes, which had lower complexity, rather than calculating the correlation between the metric data as *TrinityRCL* did, which was rather time-consuming. However, *MicroRCA* stopped working for those cases with more than 7000 nodes in the causal graph due to execution timeouts. Fig. 7 shows T for *TrinityRCL*, indicating that *TrinityRCL* supports massive datasets (e.g., up to 400,000 nodes in the causal graph), while *MicroRCA* and *MicroCause*



(a) Comparison of T for *TrinityRCL*, *MicroCause* and *MicroRCA*.

(b) Comparison of T for *TrinityRCL* and *MicroRCA*.

(c) T for *TrinityRCL*.

Fig. 7. Comparisons of the time required for *TrinityRCL* and the baseline methods.

only support small datasets (up to 7,000 and 500 nodes in the causal graph respectively). Furthermore, the experiments show no correlation between the efficiency of the three methods and the accuracy of the RCA results.

VI. Discussion

By harnessing three types of telemetry data, mining and relating different information pertinent to the identical anomalous event, *TrinityRCL* has presented its merits of allowing multi-granular and code-level RCA with reasonable time efficiency. Compared with only relying on a single type of telemetry data, combining multiple types of telemetry data enables RCA to capture the runtime status of a microservice system from multiple perspectives, which makes it possible to better support root cause localization like *TrinityRCL*. However, the heterogeneous nature of the multiple types of telemetry data, the dynamic and intricate invocation relationships among microservices and the nondeterministic anomaly propagation cast formidable challenges which require proper technical treatments. In addition, several non-technical considerations regarding the application of *TrinityRCL* are equally critical, which will be discussed in this Section.

A. Establish a Suitable Data Basis for RCA

A prerequisite for most RCA approaches is the availability of sufficient and informative telemetry data. To meet this prerequisite, we list several suggestions as follows.

Data sources: This study reveals the value of using multiple types of telemetry data for better RCA. In general, each type of telemetry data may provide specific clues to an anomalous event from a unique perspective. This is partly the reason why practitioners also proposed using multiple telemetry data to improve observability [9]. However, the challenge is that RCA relies heavily on data and consequently heterogeneity in data taxonomy, collection protocol, and the like may severely impact the efficacy of some advanced RCA approaches. A well-known community effort is the *OpenTelemetry*[7] project, which provides a collection of standard tools, APIs, and SDKs to instrument, generate, collect, and export telemetry data.

Data quality: RCA approaches such as *TrinityRCL* rely on various data to perform the analysis. Therefore, the data quality may directly impact the results. However, quality issues such as redundant or useless data cannot be completely mitigated in real production environments [34], [45]. Among the three types of telemetry data, *logging data* is most vulnerable to quality as it is usually susceptible to individual's expertise, experience and preference [46], [47], [48], [49]. In most cases, the content and format of logging data are largely determined by the developers, which creates difficulties to ensure the data quality. However, as a recent systematic study reveals, research in this area still has a long way to go before logging intentions can be well satisfied [30].

Data preparation: In an ecosystem of microservices, plenty of telemetry data, especially *metric data* and *tracing data* such as CPU usage, response time, and invocation information, is created and gathered by off-the-shelf APIs. In this sense, GQM [50] is not a concept that is often mentioned in the context of RCA. Nevertheless, the importance of GQM methodology in RCA should not be ignored. We believe that the data basis for RCA is far from satisfactory at least partially due to the negligence of GQM data preparation. For example, several studies [30], [51], [52] reveal that logging intentions (Goal) have been ignored in most logging practices, leading to insufficient information for log analysis — a critical practice required by RCA. Meanwhile, as indicated in [24], [53], useless telemetry data (even correct data) is also prevalent in RCA in practice, creating a big burden to store the data and a huge challenge to mine valuable information from the voluminous telemetry data. To this end, we argue that the GQM methodology should be applied in order to improve the efficiency in data preparation and subsequently the effectiveness in RCA.

B. Adopt RCA in a Production Environment

There are several concerns that need to be considered when adopting *TrinityRCL* in a real-world production environment.

Importance of balancing: RCA inevitably brings various costs. For example, most RCA approaches consume considerable computation resources, e.g., CPU, memory, and disk. RCA

[7] https://opentelemetry.io/

needs time to perform the analysis, which is a cost to business. It is therefore critical to maintain a balance between the cost of and the benefits from RCA. While simple rules for such a trade-off may not exist, the operations staff may need expert's judgement in most cases. For example, for critical services with multiple backup hosts, a quick RCA to identify and cut off the anomalous host or to downgrade service may make more sense than to pin down the defective source codes. Or alternatively, running *TrinityRCL* routinely to keep the whole software system healthy might be a good operation practice.

Efficiency of RCA in big data scenarios: Some RCA methods have decent performance in small-scale systems with limited data. However, they may face serious efficiency problems in big data scenarios. In [13], researchers used sliding window to solve the time lag problem between metrics and achieved good results. In the causal analysis between metrics, partial correlation coefficients are calculated for each pair of metrics, yielding a time complexity of $O(n^3)$. However, if using the sliding window method to process metric data, it would generate a set of metrics with size $n-k+1$, where n is the total length of the metric data and k is the size of the sliding window. Therefore, the algorithmic complexity of the method for causality construction using sliding window can be as high as $O(n^4)$. In a big data environment, the algorithm complexity of $O(n^4)$ is not able to generate results in time due to the high complexity. To solve this problem, *TrinityRCL* is optimized by adopting *CORT* and *DTW* in correlation calculation (cf. Section IV.C), which have a lower complexity of $O(n)$ and $O(n^2)$ respectively. As *TrinityRCL*'s optimization strategy uses less data than sliding window does in causal analysis, its accuracy is severely impacted. However, this deficiency is addressed by aggregating other types of telemetry data.

RCA scale up: In a real production environment, challenges may be faced in scaling up the RCA approach. Nevertheless, scale-up should not be a tough job in most cases as the service monitoring is performed by the dedicated APM system and the *TrinityRCL* system is hosted by a separated server. As the scope of RCA increases, more servers for the APM system and the *TrinityRCL* system can be deployed to collect more telemetry data and provide more processing capability, respectively.

VII. Threats to Validity

In this section, we cover the threats to validity that may arise from our study.

The relative concept of "root cause": The evaluation of accuracy is determined by whether the root cause has been localized by the three RCA approaches. However, as discussed in Section I, the term "root cause" is a relative concept, which varies from the localization of a specific erroneous code snippet to the indication of a general direction worthy further exploration. Therefore, a literal root cause confirmed by an engineer may not necessarily be a genuine one. For example, high CPU usage for a specific host server is a root cause frequently identified by the evaluated RCA methods. However, we did not perform further investigation to localize what caused high CPU usage (i.e., the genuine root cause) as it may take much time to investigate in an environment with massive microservices. In this sense, the foundation of accurate RCA may be diminished. However, as an informative clue, high CPU usage on a specific host also provides a valuable direction to investigate the genuine root cause, which is justifiable to be a qualified "root cause" on this ground. In this sense, this threat risk could be controlled.

Implementation of the baseline methods: Since none of the baseline methods (i.e., *MicroRCA* and *MicroCause*) provides publicly accessible source code, we had to reproduce the algorithms and develop prototypes of these methods based on the published articles. Therefore, the implementation of the baseline methods may not be exactly equal to the original ones. To mitigate this risk of validity, we evaluated our prototypes using the (simulated) dataset mentioned and obtained similar results before using them to conduct experimental investigation using the dataset in this study. With this strategy, this validity threat can be reasonably minimized.

Dataset: To evaluate *TrinityRCL*, we used the data collected by the *Raptor* system from the real production environment of *Meituan* as described in Section V.A.3. However, anomalies and the propagation of anomalies may take place in other production environments in different ways. Therefore, the dataset derived from *Meituan* may not be representative of other microservice systems in other companies. To mitigate this threat, we applied a randomization strategy in case selection where two researchers independently prepared the dataset and evaluated *TrinityRCL* separately. A total of 30 cases derived from the real production environment were selected, which is a relatively large dataset, and these cases cover RCA at different levels of granularity to simulate real scenarios. In this sense, this threat to validity can be fairly mitigated. Besides, *Raptor* adopts the *OpenTelemetry* standard, and with more and more organizations supporting this standard, this validity issue could be further controlled. Another validity threat pertinent to the dataset is that the "root causes" localized by the three RCA methods may be related to possible undetected anomalies instead of the identified anomalies due to the complexity in analyzing the massive volume of data in each case. We worked with the professional operations staff in *Meituan* to analyze each clue so that this factor can be fairly controlled, however, this risk may not be entirely eliminated.

The 30-minute time span: We applied a time span of 30 minutes to perform RCA using *TrinityRCL* and to prepare the experimental cases. The time span is a variable in *TrinityRCL*, which can be adjusted by operations staff according to their needs. The choice of 30 minutes is empirically based on our experience with *Meituan*'s production environment from previous work [4] by compromising between complexity and performance. Nevertheless, this time span may need to be tuned to fit different application scenarios. Besides, the time span may also impact the data content involved in our experimental evaluations, which raises the necessity to evaluate *TrinityRCL* in more scenarios, e.g., with dynamic length of time spans.

Constants in TrinityRCL*:* The default time frame (cf. Section IV.A) and α, β, γ in (3) are constants that are set based on our previous experience in the real production environment in *Meituan* and the domain knowledge from the operations staff. In order to apply them to different systems, engineers need to manually adjust them to achieve better RCA performance.

Apparently, these constants may vary in different microservice systems. In this sense, more research should be carried out to identify and investigate the factors impacting the value of these constants and explore a systematic way to appropriately set and dynamically adjust them to adapt to different contexts.

Absence of machine learning in TrinityRCL*:* Machine learning techniques have been increasingly adopted in RCA approaches. However, we do not recommend any machine learning techniques in current version of *TrinityRCL*. Although we have tried some in *TrinityRCL*, their results were not satisfactory. Nevertheless, we did not optimize this technical route scrupulously, which leaves a space for the future improvement of *TrinityRCL*.

Telemetry data used by baseline methods: The selected baseline methods perform RCA without using all the three types of telemetry data together, which is part of the reason why they only support uni-granular RCA and their accuracy is lower than that of *TrinityRCL*. As there is currently no method that uses all three types of telemetry data to support RCA, we had to compare several methods that are similar to *TrinityRCL* in terms of localizing root causes at different levels of granularity using different types of telemetry data. In particular, *MicroRCA* combines tracing and metric data to localize service-level root causes, while *MicroCause* only uses metric data to localize metric-level root causes. In this way, we are able to establish a good understanding of the performance of *TrinityRCL* against that of the baseline methods in terms of granularity, accuracy and efficiency.

VIII. Conclusions and Future Work

With the wide adoption of microservice architecture by current large-scale online systems, service monitoring and RCA face new challenges. For an RCA method, its performance in terms of accuracy and efficiency, as well as its ability to support multi-granular and code-level root cause localization are both critical for operations staff to keep online services running healthy. In this article, we propose *TrinityRCL*, which, to the best of our knowledge, is the first method using three different types of telemetry data to perform multi-granular RCA in microservice systems. *TrinityRCL* exploits multiple types of telemetry data to construct a causal graph with which service anomalous symptoms are correlated to infer the anomalous microservices. At this early stage, *TrinityRCL* shows great potential to localize root causes at multiple levels of granularity with its unique ability of localization at the finest code-level. It can outperform other methods in terms of accuracy at the same level of granularity and is particularly effective to support large-scale systems with massive telemetry data. Furthermore, it improves the accuracy of root cause localization in circumstances where dynamic invocation and nondeterministic anomaly propagation co-exist.

However, we also notice several limitations when we apply *TrinityRCL* in the real-world production environment. To this end, we suggest several topics for future work. First, it would be valuable to design and implement a strategy that is able to dynamically set various constants in *TrinityRCL* in order to accommodate different microservice systems. For example, some feedback mechanisms can be used to adjust these constants in a dynamic manner to find the best combination of parameter values. Second, data plays a vital role in RCA, and there are very few studies using all the three types of telemetry data to support RCA. More research endeavor is required to optimize RCA based on multiple sources of telemetry data. For example, one important topic is to improve the quality of telemetry data.

References

[1] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Softw.*, vol. 35, no. 3, pp. 24–35, May/Jun. 2018. [Online]. Available: https://doi.org/10.1109%2Fms.2018.2141039

[2] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root cause localization of performance issues in microservices," in *Proc. IEEE/IFIP Netw. Operations Manage. Symp.*, Budapest, Hungary, 2020, pp. 1–9. [Online]. Available: http://dx.doi.org/10.1109/NOMS47738.2020.9110353

[3] J. Soldani and A. Brogi, "Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey," *ACM Comput. Surveys*, vol. 55, no. 3, pp. 1–39, Feb. 2022. [Online]. Available: https://doi.org/10.1145%2F3501297

[4] G. Rong et al., "Locating anomaly clues for atypical anomalous services: An industrial exploration," *IEEE Trans. Dependable Secure Comput.*, to be published, doi: 10.1109/TDSC.2022.3181143.

[5] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables DevOps: Migration to a cloud-native architecture," *IEEE Softw.*, vol. 33, no. 3, pp. 42–52, May/Jun. 2016. [Online]. Available: https://doi.org/10.1109%2Fms.2016.64

[6] S. J. Fowler, *Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization*. Sebastopol, CA, USA: O'Reilly Media, 2016.

[7] X. Zhou et al., "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Trans. Softw. Eng.*, vol. 47, no. 2, pp. 243–260, Feb. 2021. [Online]. Available: https://doi.org/10.1109%2Ftse.2018.2887384

[8] M. Waseem, P. Liang, M. Shahin, A. Ahmad, and A. R. Nassab, "On the nature of issues in five open source microservices systems: An empirical study," in *Evaluation and Assessment in Software Engineering*. New York, NY, USA, Jun. 2021, pp. 201–210. [Online]. Available: https://doi.org/10.1145/3463274.3463337

[9] R. Gatev, "Observability: Logs, metrics, and traces," in *Introducing Distributed Application Runtime (Dapr)*. New York, NY, USA, Jun. 2021, pp. 233–252. [Online]. Available: https://doi.org/10.1007/978--1-4842-6998-5_12

[10] C. Sridharan,*Distributed Systems Observability: A Guide to Building Robust Systems*. Sebastopol, CA, USA: O'Reilly Media, 2018.

[11] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *Service-Oriented Computing*. Berlin, Germany: Springer, Nov. 2018, pp. 3–20. [Online]. Available: https://doi.org/10.1007%2F978--3-030-03596-9_1

[12] P. Chen, Y. Qi, and D. Hou, "CauseInfer: Automated end-to-end performance diagnosis with hierarchical causality graph in cloud environment," *IEEE Trans. Serv. Comput.*, vol. 12, no. 2, pp. 214–230, Mar./Apr. 2019. [Online]. Available: https://doi.org/10.1109%2Ftsc.2016.2607739

[13] Y. Meng et al., "Localizing failure root causes in a microservice through causality inference," in *Proc. IEEE/ACM 28th Int. Symp. Qual. Service*, Hang Zhou, China, 2020, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/IWQoS49365.2020.9213058

[14] M. Ma, J. Xu, Y. Wang, P. Chen, Z. Zhang, and P. Wang, "AutoMAP: Diagnose your microservice-based web applications automatically," in *Proc. Web Conf.*, Taipei, Taiwan, 2020, pp. 246–258. [Online]. Available: https://doi.org/10.1145/3366423.3380111

[15] Á. Brandón, M. Solé, A. Huélamo, D. Solans, M. S. Pérez, and V. Munt és-Mulero, "Graph-based root cause analysis for service-oriented and microservice architectures," *J. Syst. Softw.*, vol. 159, Jan. 2020, Art. no. 110432. [Online]. Available: https://doi.org/10.1016/j.jss.2019.110432

[16] G. Yu et al., "MicroRank: End-to-end latency issue localization with extended spectrum analysis in microservice environments," in *Proc. ACM Web Conf. Assoc. Comput. Machinery*, 2021, pp. 3087–3098. [Online]. Available: https://doi.org/10.1145/3442381.3449905

[17] L. Huang and T. Zhu, "Tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces," in *Proc. ACM Symp. Cloud Comput. Assoc. Comput. Machinery*, 2021, pp. 76–91. [Online]. Available: https://doi.org/10.1145/3472883.3486994
[18] M. Kim, R. Sumbaly, and S. Shah, "Root cause detection in a service-oriented architecture," *SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 1, pp. 93–104, Jun. 2013. [Online]. Available: https://doi.org/10.1145/2494232.2465753
[19] J. Weng, J. H. Wang, J. Yang, and Y. Yang, "Root cause analysis of anomalies of multitier services in public clouds," *IEEE/ACM Trans. Netw.*, vol. 26, no. 4, pp. 1646–1659, Aug. 2018. [Online]. Available: http://dx.doi.org/10.1109/TNET.2018.2843805
[20] H. Wang et al., "GRANO: Interactive graph-based root cause analysis for cloud-native distributed data platform," *Proc. VLDB Endowment*, vol. 12, no. 12, pp. 1942–1945, Aug. 2019. [Online]. Available: https://doi.org/10.14778/3352063.3352105
[21] Y.-Y. Liu, J.-J. Slotine, and A.-L. Barabási, "Observability of complex systems," *Proc. Nat. Acad. Sci. USA*, vol. 110, no. 7, pp. 2460–2465, Jan. 2013. [Online]. Available: https://doi.org/10.1073/pnas.1215508110
[22] H. Li, T.-H. P. Chen, W. Shang, and A. E. Hassan, "Studying software logging using topic models," *Empirical Softw. Eng.*, vol. 23, no. 5, pp. 2655–2694, Oct. 2018. [Online]. Available: https://doi.org/10.1007/s10664--018-9595-8
[23] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *Proc. IEEE/ACM 37th Int. Conf. Softw. Eng.*, 2015, pp. 415–425. [Online]. Available: https://doi.org/10.1109/icse.2015.60
[24] H. Anu, J. Chen, W. Shi, J. Hou, B. Liang, and B. Qin, "An approach to recommendation of verbosity log levels based on logging intention," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2019, pp. 125–134. [Online]. Available: https://doi.org/10.1109/icsme.2019.00022
[25] X. Liu, T. Jia, Y. Li, H. Yu, Y. Yue, and C. Hou, "Automatically generating descriptive texts in logging statements: How far are we?" in *Proc. Asian Symp. Program. Lang. Syst.*, Fukuoka, Japan, 2020, pp. 251–269. [Online]. Available: https://doi.org/10.1007/978--3-030-64437-6_13
[26] P. He, Z. Chen, S. He, and M. R. Lyu, "Characterizing the natural language descriptions in software logging statements," in *Proc. IEEE/ACM 33rd Int. Conf. Automated Softw. Eng.*, Montpellier, France, 2018, pp. 178–189. [Online]. Available: https://doi.org/10.1145/3238147.3238193
[27] M. Marron, "Log++ logging for a cloud-native world," in *Proc. 14th ACM SIGPLAN Int. Symp. Dynamic Lang.*, 2018, pp. 25–36. [Online]. Available: https://doi.org/10.1145/3276945.3276952
[28] P. Aggarwal et al., "Localization of operational faults in cloud applications by mining causal dependencies in logs using golden signals," in *Proc. Int. Conf. Service-Oriented Comput. Workshops*, 2021, pp. 137–149. [Online]. Available: https://doi.org/10.1007/978--3-030-76352-7_17
[29] H. Wang et al., "Groot: An event-graph-based approach for root cause analysis in industrial settings," in *Proc. IEEE/ACM 36th Int. Conf. Automated Softw. Eng.*, 2021, pp. 419–429. [Online]. Available: https://doi.org/10.1109/ase51524.2021.9678708
[30] S. Gu, G. Rong, H. Zhang, and H. Shen, "Logging practices in software engineering: A systematic mapping study," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2022.3166924.
[31] L. Chen, "Microservices: Architecting for continuous delivery and DevOps," in *Proc. IEEE Int. Conf. Softw. Archit.*, 2018, pp. 39–46. [Online]. Available: https://doi.org/10.1109/icsa.2018.00013
[32] Y. Gan et al., "Seer: Leveraging Big Data to navigate the complexity of performance debugging in cloud microservices," in *Proc. 24th Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2019, pp. 19–33.
[33] P. Liu et al., "Unsupervised detection of microservice trace anomalies through service-level deep Bayesian networks," in *Proc. IEEE 31st Int. Symp. Softw. Rel. Eng.*, Coimbra, Portugal, 2020, pp. 48–58. [Online]. Available: https://doi.org/10.1109/issre5003.2020.00014
[34] Q. Fu et al., "Where do developers log? An empirical study on logging practices in industry," in *Proc. 36th Int. Conf. Softw. Eng.*, Hyderabad, India, 2014, pp. 24–33. [Online]. Available: https://doi.org/10.1145/2591062.2591175
[35] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Comput. Surveys*, vol. 48, no. 1, pp. 1–35, Jul. 2015. [Online]. Available: https://doi.org/10.1145/2791120
[36] D. J. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series," in *Proc. AAAI Int. Conf. Knowl. Discov. Data Mining Workshop*, Seattle, WA, USA, 1994, pp. 359–370.
[37] A. D. Chouakria and P. N. Nagabhushan, "Adaptive dissimilarity index for measuring time series proximity," *Adv. Data Anal. Classification*, vol. 1, no. 1, pp. 5–21, Jan. 2007. [Online]. Available: https://doi.org/10.1007%2Fs11634--006-0004-6
[38] Z. Li et al., "Practical root cause localization for microservice systems via trace analysis," in *Proc. IEEE/ACM 29th Int. Symp. Qual. Serv.*, Tokyo, Japan, 2021, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/IWQOS52092.2021.9521340
[39] J.-Y. Pan, H.-J. Yang, C. Faloutsos, and P. Duygulu, "Automatic multimedia cross-modal correlation discovery," in *Proc. 10th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, Seattle, WA, USA, 2004, pp. 653–658. [Online]. Available: https://doi.org/10.1145/1014052.1014135
[40] P. Wang et al., "CloudRanger: Root cause identification for cloud native systems," in *Proc. 18th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, Washington, DC, USA, 2018, pp. 492–502. [Online]. Available: https://doi.org/10.1109/ccgrid.2018.00076
[41] A. J. Walker, "New fast method for generating discrete random numbers with arbitrary frequency distributions," *Electron. Lett.*, vol. 10, no. 8, pp. 127–128, 1974. [Online]. Available: https://doi.org/10.1049/el:19740097
[42] G. Rong et al., "Locating the clues of declining success rate of service calls," in *Proc. IEEE 31st Int. Symp. Softw. Rel. Eng.*, Coimbra, Portugal, 2020, pp. 335–345. [Online]. Available: https://doi.org/10.1109%2Fissre5003.2020.00039
[43] G. Jeh and J. Widom, "Scaling personalized web search," in *Proc. 12th Int. Conf. World Wide Web*, New York, NY, USA, 2003, pp. 271–279. [Online]. Available: https://doi.org/10.1145%2F775152.775191
[44] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise Reduction in Speech Processing*. Berlin, Germany: Springer, 2009, pp. 1–4.
[45] C. Zhi, J. Yin, S. Deng, M. Ye, M. Fu, and T. Xie, "An exploratory study of logging configuration practice in Java," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2019, pp. 459–469. [Online]. Available: https://doi.org/10.1109/icsme.2019.00079
[46] G. Rong, S. Gu, H. Zhang, D. Shao, and W. Liu, "How is logging practice implemented in open source software projects? A preliminary exploration," in *Proc. IEEE 25th Australas. Softw. Eng. Conf.*, 2018, pp. 171–180. [Online]. Available: https://doi.org/10.1109/aswec.2018.00031
[47] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo, "Industry practices and event logging: Assessment of a critical software development process," in *Proc. IEEE/ACM 37th Int. Conf. Softw. Eng.*, Florence, Italy, 2015, pp. 169–178. [Online]. Available: https://doi.org/10.1109/icse.2015.145
[48] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *Proc. IEEE 34th Int. Conf. Softw. Eng.*, 2012, pp. 102–112. [Online]. Available: https://doi.org/10.1109/icse.2012.6227202
[49] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, Feb. 2012, Art. no. 4. [Online]. Available: https://doi.org/10.1145/2110356.2110360
[50] V. R. Basili, "Goal question metric paradigm," *Encyclopedia Softw. Eng.*, vol. 1, pp. 528–532, 1994.
[51] G. Rong, Y. Xu, S. Gu, H. Zhang, and D. Shao, "Can you capture information as you intend to? A case study on logging practice in industry," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2020, pp. 12–22. [Online]. Available: https://doi.org/10.1109/icsme46990.2020.00012
[52] H. Li, W. Shang, B. Adams, M. Sayagh, and A. E. Hassan, "A qualitative study of the benefits and costs of logging from developers' perspectives," *IEEE Trans. on Softw. Eng.*, vol. 47, no. 12, pp. 2858–2873, Dec. 2021. [Online]. Available: https://doi.org/10.1109/tse.2020.2970422
[53] R. Andrews, F. Emamjome, A. H. ter Hofstede, and H. A. Reijers, "Root-cause analysis of process-data quality problems," *J. Bus. Analytics*, vol. 5, no. 1, pp. 51–75, Aug. 2021. [Online]. Available: https://doi.org/10.1080/2573234x.2021.1947751

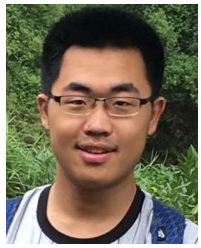
Shenghui Gu received the BSc degree from Nanjing University, China. He is currently working toward the PhD degree with the Software Institute, Nanjing University, China. His research interests are in software engineering, particularly in AIOps, software log analytics, DevOps, as well as empirical and evidence-based software engineering.

Guoping Rong received the BSc degree in computer science and technology, the MSc degree in software theory and PhD degree in applied software engineering, all from Nanjing University. He now is a faculty member with the Software Institute, Nanjing University and the director of the joint laboratory of Nanjing University and Transwarp on data technology. His research area includes software process, DevOps, AIOps and empirical methodology, etc.

Tian Ren received the MS degree in software engineering from Nanjing University, Nanjing, China, in 2018. He has several years of industrial experience as a software developer and is currently a senior engineer with Meituan, China. His research interests include software observability, software operations, and AIOps.

He Zhang is a full professor of Software Engineering and the director of DevOps+ Research Laboratory with the Nanjing University, China, also a principal scientist with CSIRO, Australia. He undertakes research in software engineering, in particular software & systems process, software architecture, DevOps, software security, blockchain-oriented software engineering, empirical and evidence-based software engineering. He has published more than 180 peer-reviewed papers in high quality international conferences and journals, and won 11 Best/Distinguished Paper Awards from several prestigious international conferences and journals in software engineering community.

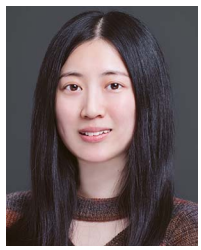
Haifeng Shen is an associate professor and the director of the HilstLab with the Faculty of Law and Business, Australian Catholic University. His research expertise is interdisciplinary and revolves around human-centred artificial intelligence and software technologies, which is uniquely positioned at the intersection of human computer interaction, software engineering, and artificial intelligence with a unique focus on 'interaction' and 'integration': human-AI interaction, human-software interaction, and integration of AI and software.



Yongda Yu is currently working toward the PhD degree with Software Engineering at Nanjing University. His research interests include Artificial Intelligence for software engineering and Artificial Intelligence for IT Operations, especially accelerating software development and improving software quality by using process data.

Xian Li is currently working toward the professional master's degree with the Software Institute of Nanjing University, China. To help software operations engineers find, locate and solve problems faster, his research focuses on anomaly detection and root cause location.

Jian Ouyang received the MS degree in computer science from Tsinghua University, Beijing, China, in 2006. He has more than 10 years of industrial experience as a software developer and is currently a technical director of system software with Meituan, China. His research interests include software observability, operating system, cluster scheduling, cloud native and AIOps.



Chunan Chen received the MS degree in software engineering from Nanjing University, Nanjing, China, in 2015. She has several years of industrial experience as a software developer and is currently a technical expert with Meituan, China. Her research interests include software observability, software operations, and AIOps.