CMDiagnostor: An Ambiguity-Aware Root Cause Localization Approach Based on Call Metric Data

Qingyang Yu
Tsinghua University
Beijing, China

Changhua Pei*
Computer Network Information Center, Chinese Academy of Sciences
Beijing, China

Bowen Hao
Nankai University
Tianjin, China

Mingjie Li
Tsinghua University
Beijing, China

Zeyan Li
Tsinghua University
Beijing, China

Shenglin Zhang†
Nankai University, HL-IT
Tianjin, China

Xianglin Lu
Tsinghua University
Beijing, China

Rui Wang
Jiaqi Li
Zhenyu Wu
Tencent
Beijing, China

Dan Pei
Tsinghua University
Beijing, China

ABSTRACT

The availability of online services is vital as its strong relevance to revenue and user experience. To ensure online services' availability, quickly localizing the root causes of system failures is crucial. Given the high resource consumption of traces, call metric data are widely used by existing approaches to construct call graphs in practice. However, ambiguous correspondences between upstream and downstream calls may exist and result in exploring unexpected edges in the constructed call graph. Conducting root cause localization on this graph may lead to misjudgments of real root causes. To the best of our knowledge, we are the first to investigate such ambiguity, which is overlooked in the existing literature. Inspired by the law of large numbers and the Markov properties of network traffic, we propose a regression-based method (named AmSitor) to address this problem effectively. Based on AmSitor, we propose an ambiguity-aware root cause localization approach based on Call Metric Data named CMDiagnostor, containing metric anomaly detection, ambiguity-free call graph construction, root cause exploration, and candidate root cause ranking modules. The comprehensive experimental evaluations conducted on real-world datasets show that our CMDiagnostor can outperform the state-of-the-art approaches by 14% on the top-5 hit rate. Moreover, AmSitor can also be applied to existing baseline approaches separately to improve their performances one step further. The source code is released at https://github.com/NetManAIOps/CMDiagnostor.

*Changhua Pei is the corresponding author. Email: chpei@cnic.cn
†HL-IT is short for Haihe Laboratory of Information Technology Application Innovation.

WWW '23, May 1–5, 2023, Austin, TX, USA

ACM ISBN 978-1-4503-9416-1/23/04.
https://doi.org/10.1145/3543507.3583302

CCS CONCEPTS

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

KEYWORDS

online service, root cause localization, call metric data, ambiguity

ACM Reference Format:
Qingyang Yu, Changhua Pei, Bowen Hao, Mingjie Li, Zeyan Li, Shenglin Zhang, Xianglin Lu, Rui Wang, Jiaqi Li, Zhenyu Wu, and Dan Pei. 2023. CMDiagnostor: An Ambiguity-Aware Root Cause Localization Approach Based on Call Metric Data. In *Proceedings of the ACM Web Conference 2023 (WWW '23), May 1–5, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3543507.3583302

1 INTRODUCTION

Recent years have witnessed the booming of online services, *e.g.*, online shopping websites like Amazon, short-video platforms like Tiktok, and social media networks like Instagram. The availability of online services is crucial as its strong relevance to revenue and user experiences. For example, Amazon's one-hour downtime on Prime Day may lead to the loss of up to $100 million in sales [26]. How to quickly locate the root causes of system failures has become an active research topic these years. This paper mainly focuses on root cause localization (**RCL**) based on call metric data.

Among different kinds of system data used for RCL, Call Metric Data (**CMD**) are widely used [16] as the reasonable trade-off between information capacity and system collection burden. Some leading internet companies like Alibaba [16] and Tencent (this paper) use CMD for RCL. Table 1 gives an example record of CMD. Each piece of CMD indicates the call events between two nodes which are labeled as caller and callee according to the call direction. The node can be any system entity, ranging from fine-grained method level (this paper) to coarse-grained service level. For each call event with the same caller and callee, CMD records its statistical information, which is shown as metric in Table 1. Compared with traditional single-node KPI time series data [21], CMD provides real

Table 1: The fields and the meanings of call metric data.

Category	Field	Meaning
Caller	Caller_service	Service name of the caller
	Caller_method	Method name of the caller
Callee	Callee_service	Service name of the callee
	Callee_method	Method name of the callee
Metric	RC	Request count per minute
	RT	Average response time per minute
	EC	Error count per minute

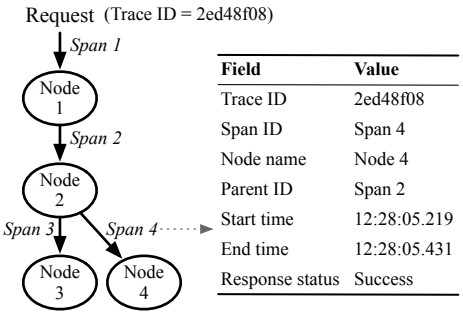


Figure 1: The call path of an user request and the snapshot of one piece of trace data.

call relationships between two nodes, which are more friendly for RCL. Compared with the pure call graph, CMD provides statistical results for various call metrics, which help identify the anomalous status of call events.

Detailed trace data [5, 8, 24] are prevalent these years, which provide more information than CMD. Figure 1 shows a trace recording example. A unique trace ID is generated for each user request. A complete trace consists of multiple calls (shown as Span in Figure 1). CMD can be considered the aggregations of spans with the same caller and callee but ignoring different requests. Although trace data distinguish among different requests, they are challenging for large-scale real-world online services to use. First, the high transmission cost and storage burden should be considered. The extra transmission bandwidth and storage are $10^3\times$ to $10^4\times$ larger compared with CMD under different compression rates (*e.g.*, a real-world trace dataset $\mathcal{D}_{RT}$ used in Section 5.2.2 converted to CMD achieves 2500× compression). There are huge resource wastes because the majority of traces are normal, which are useless for RCL. Second, the massive trace data bring extra computation costs for RCL algorithms, *e.g.*, parsing, reconstruction, and analyzing. Hence, this paper mainly focuses on RCL based on CMD.

Though widely usage of CMD, the threat of "ambiguity" is overlooked in existing RCL algorithms. The ambiguity arises when a node is called by multiple nodes and calls at least one node. We give a toy example of the ambiguity in Figure 2, where node B has two callers (A and C) and one callee D. Figure 2(a) shows three pieces of CMD data. Based on Figure 2(a), existing methods like MicroHECL[16] construct a call graph, as shown in Figure 2(b). Unfortunately, the call graph in Figure 2(b) may be inaccurate. Actually, three possible control flows are shown in Figure 2(c). If the calls $A \rightarrow B$ and $B \rightarrow D$ belong to different user requests, the

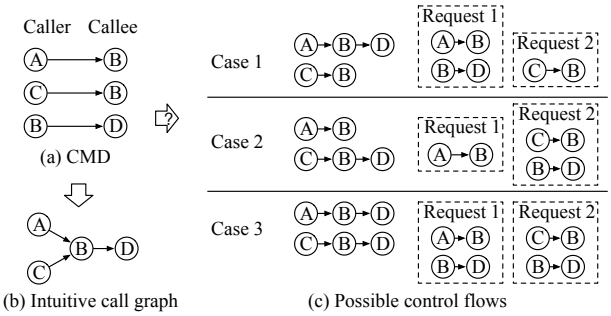


Figure 2: Illustration of the ambiguity faced by existing root cause localization methods when constructing the call graph from call metric data. (a) shows the snapshot of call metric data. (b) shows the call graph constructed by existing methods like MicroHECL [16], which is one of the three possible control flows shown in (c). In this paper, we try to identify the right one to eliminate the negative impact of ambiguity.

correct control flow should be case 2, which is different from Figure 2(b). The core idea of this work is to benefit RCL by identifying the actual call graph from all possible cases.

In this paper, we propose an ambiguity-aware RCL approach called CMDiagnostor. First, we conduct theoretical analyses to reveal the feasibility of eliminating ambiguity without the help of resource-consuming trace data. Based on the analyses, a disambiguation algorithm called AmSitor is proposed and integrated into CMDiagnostor to construct the ambiguity-free call graph. Unsupervised anomaly detection methods are proposed to identify the anomaly calls in the graph. Then, an explainable root cause exploration method is proposed to identify the potential root causes based on the ambiguity-free call graph. At last, we rank the candidate root causes based on our proposed key indicators. To summarize, we make the following contributions.

- To the best of our knowledge, we are the first to investigate the ambiguity in the call graph. Based on the law of large numbers and the Markov properties of network traffic, a simple but effective regression-based method called AmSitor is proposed to eliminate ambiguity. Evaluation results show that AmSitor can improve existing RCL methods.
- CMDiagnostor, a four-stage (*i.e.*, detection, construction, exploration, and ranking) based framework, is designed to identify the root causes in an efficient and explainable way. The evaluation results show that our CMDiagnostor outperforms the state-of-the-art algorithms by 14% on the top-5 hit rate.
- In this paper, we conduct comprehensive experimental evaluations based on real-world datasets. Detailed case studies are provided with practical experiences. Our code has been released.

2 SYSTEM OVERVIEW

2.1 Problem Statement

In this section, we first introduce the essential notations. Then we give the formal definition of our problem in this paper.

SLO: Service Level Objectives (SLOs) are adopted by the system we studied to measure customer satisfaction [1]. Once SLOs are violated, alerts covering information, such as alerting services, will be generated to notify operators.

Node: A node represents a virtual entity belonging to some online service, which can be method-level, service-level, etc. A service consists of multiple methods. Our root cause algorithm is built on a method-level dataset (shown in Table 1) because it provides finer-grained information. We output the service-level root causes as the operators usually focus on root cause services rather than a mass of methods. It is noteworthy that the method-level causes can also be provided to operators for further failure recovery.

Call: A call refers to that a service node (A) invokes another service node (B) with corresponding minute-granularity metrics, as the record in Table 1. The call here can be represented as $A \rightarrow B$. A service node in this paper refers to a method. If there are a call ($A \rightarrow B$) invokes a node (B) and a call ($B \rightarrow C$) invoked by the same node (B), we call the former call the *upstream call* and the latter call the *downstream call*. The calls whose caller or callee service reports a SLO violation alert are called *entry calls*.

The main objective of our paper is: **Once the system alert for SLO violation is reported, we need to identify the most-possible cause services as soon as possible based on previously collected method-level call metric data**.

2.2 Challenges

To achieve the above goal, there are three main challenges.

- **Ambiguity**: As illustrated in Figure 2, the ambiguity may cause a wrongly constructed graph which may reduce the performance of RCL. Eliminating the negative effects without the need to collect more detailed trace data is challenging.
- **Efficiency**: We aim to design a RCL algorithm for one top-tier online service provider. Once an alert is reported, hundreds of thousands of methods are potential causes. How efficiently identifying the top causes is challenging.
- **Explainability**: Root cause localization is not the final stop for operators. They need to take action to eliminate failures. An explainable algorithm can provide more detailed information, friendly to the following failure-eliminating process.

2.3 Design Overview of CMDiagnostor

To address the above challenges, we propose CMDiagnostor, an ambiguity-aware RCL approach. The overview is shown in Figure 3. CMDiagnostor inputs alerting services and CMD, and outputs the ranked root cause service list. The entire process includes four modules: **Metric Anomaly Detection**, **Ambiguity-free Call Graph Construction**, **Root Cause Exploration**, and **Candidate Root Cause Ranking**. Once a service alert arises, CMDiagnostor first detects anomalies of entry calls and constructs the call graph with AmSitor. Next, CMDiagnostor traverses the calls from abnormal entry calls in the graph using three pruning strategies and ranks the potential root causes based on our proposed key indicators.

Metric Anomaly Detection. This part identifies the abnormal calls from CMD for the subsequent two modules. First, it finds the abnormal entry calls for the ambiguity-free call graph construction stage. The entry call refers to the call whose caller or callee service violates the SLO requirements. It is time-consuming to conduct RCL for all entry calls (challenge of **efficiency**) as numerous entry calls can be found for an alerting service. Based on the observation that the root causes are also anomalous, this module identifies

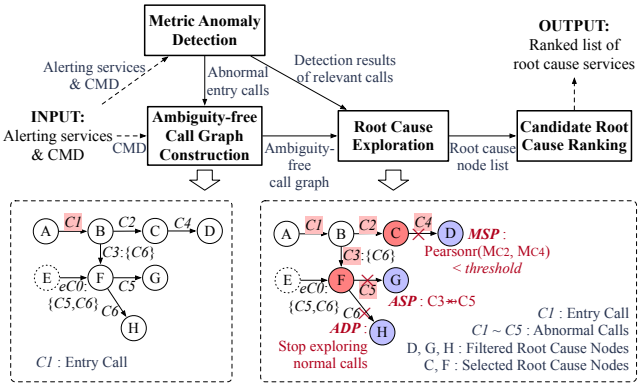


Figure 3: Design overview of CMDiagnostor.

the abnormal entry calls to improve efficiency. Moreover, it also identifies the relevant abnormal calls for the root cause exploration stage. More details can be found in Section 4.1.

Ambiguity-free Call Graph Construction: This part aims to construct the call graph for each abnormal entry call given by the previous module. However, the ambiguity, as shown in Figure 2, results in multiple possible control flows, which may reduce the RCL performance (challenge of **ambiguity**). Based on the theoretical analyses, we propose a traffic regression method (called AmSitor) to handle the ambiguity and construct the ambiguity-free call graph. More detailed information can be found in Section 3 and Section 4.2.

Root Cause Exploration and **Candidate Root Cause Ranking**: Based on the ambiguity-free call graphs and anomaly detection results provided by the previous two modules, we propose a pruning-based exploration method to exclude those false-positive root causes. The detailed pruning strategies can be found in Section 4.3. To further pick the most likely root causes, we rank the remaining root causes after the exploration process based on the carefully designed key indicators. More details about the ranking algorithm can be found in Section 4.4. Instead of model-based methods [22], we adopt rule-based methods in the exploration and ranking stages for the sake of **explainability** and **efficiency**.

3 AMBIGUITY AND SOLUTION

In this section, we first summarize the **Am**biguous **Sit**uation (short as AmSit hereinafter) where the ambiguity arises. Then we give the theoretical analyses of the traffic in AmSit and propose an algorithm called AmSitor to solve the AmSit. It is noteworthy that AmSitor can not only be applied in call graph construction by CMD but also work well for other call graph construction scenarios.

3.1 AmSit

Ambiguity arises in the situation where a node has at least two upstream calls and at least one downstream call. Figure 2 gives a simple example of the situation, which has two upstream calls ($A \rightarrow B$, $C \rightarrow B$) and one downstream call ($B \rightarrow D$). From the three calls in Figure 2(a), existing methods like MicroHECL [16] construct the call graph shown in Figure 2(b). However, the graph is ambiguous because the actual control flow may be one of the three cases in Figure 2(c). For the convenience of description, we call such a situation **AmSit**. AmSit can be solved by complete trace data that

are expensive to record and rarely available in large service systems. Therefore we focus on solving AmSit using easily acquired CMD.

3.2 Theoretical Analyses of Traffic

For readability, we give the following corollaries based on the law of large numbers and the Markov properties of network traffic. The detailed mathematical derivation is provided in Appendix A.

Corollary 3.1. *For each time slicing t, the overall number of calls from node A to its downstream node B can be approximated as the weighted sum of call numbers from A's one-hop upstream nodes.*

Corollary 3.2. *The weight in Corollary 3.1 is the expected number of A's calling B per U's calling A, represented as $\mathbb{E}\left(n_{A\to B}|U\to A\right)$.*

3.3 Algorithm for Solving the AmSit

According to Corollary 3.1 and Corollary 3.2, from the CMD dataset, we can get the number of calls from A to B at the time t and the number of each kind of upstream call at the time t. The only thing we want is the "weight" mentioned in Corollary 3.2, *i.e.*, the expected number of A's calling B per U's calling A. We naturally think of linearly regressing one downstream traffic on its possible upstream traffic. Time series with the same start and end times are used for the regression. The regression coefficient of each upstream traffic can be considered as its expectation. Upstream calls with a low coefficient, *e.g.*, less than or equal to a threshold (*e.g.*, 0.005), will be dropped. Notice that we should keep the coefficient non-negative during the linear regression analysis, as $\mathbb{E}\left(n_{A\to B}|U\to A\right)$ means the expectation of the number of the calls under the given condition.

Based on the analyses above, we propose AmSitor to determine the correspondences of upstream and downstream calls in AmSits based on their traffic. The pseudo-code is shown in Algorithm 1. We use *scipy.optimize.dual_annealing* [4], a python-based global optimization function, to implement linear regression.

Algorithm 1 (AmSitor): Determine correspondences of upstream and downstream calls in the AmSit based on traffic regression

Input: Upstream calls $\mathbf{C}_U, |\mathbf{C}_U| \geq 2$ and their traffic $\mathbf{T}_U = \{\mathbf{T}_{U,j} \mid j = 1, 2, \cdots, |\mathbf{C}_U|\}$; Downstream calls $\mathbf{C}_D, |\mathbf{C}_D| \geq 1$ and their traffic $\mathbf{T}_D = \{\mathbf{T}_{D,i} \mid i = 1, 2, \cdots, |\mathbf{C}_D|\}$;

Output: A mapping from $\mathbf{C}_D$ to their upstream calls;

```
results ← A mapping from C_D to their upstream calls
for all C_i ∈ C_D do
    linear regress T_{D,i} = Σ_{C_U} r_j T_{U,j} while keeping r_j ≥ 0
    results[C_i] ← {C_{U,j} | r_j > threshold ∧ C_{U,j} ∈ C_U}
end for
return results
```

4 METHODOLOGY

In this section, we introduce the details of CMDiagnostor, *i.e.*, the four stages of **Metric Anomaly Detection**, **Ambiguity-free Call Graph Construction**, **Root Cause Exploration**, and **Candidate Root Cause Ranking**, as shown in Figure 3.

4.1 Metric Anomaly Detection

The purpose of this module is to detect anomalous calls in CMD using metrics that are attached to these calls. As shown in Table 1, three types of metrics can be used, *i.e.*, request count per minute (RC), average response time per minute (RT), and error count per minute (EC). The detection results are used in the two following modules. In the ambiguity-free call graph construction stage, the anomalous calls which contain alerting service nodes are used as the *entry* calls for graph construction. In the root cause exploration, only the anomalous calls are traversed because of the common knowledge that the normal call can not become the root cause.

4.1.1 Anomaly Propagation Directions. Once a service alert is reported, the straightforward way is to detect the anomalous calls in both directions, *i.e.*, upstream and downstream. However, the anomaly propagation directions are different for different metrics. According to [16], the call metrics can be divided into three categories: traffic-related metrics (like RC), performance-related metrics (like RT), and reliability-related metrics (like EC). For traffic-related metrics, the increase/decrease in upstream calls leads to an increase/decrease in downstream calls. Thus the propagation direction of traffic-related metrics is from upstream to downstream. Contrary to the above analysis, the propagation direction is from downstream to upstream for the other two kinds of metrics.

4.1.2 Anomaly Metric Selection. Among the three metrics shown in Table 1, we choose RT and EC instead of RC for anomaly detection and root cause localization for two reasons. First, the majority of RC anomalies can be attributed to RT and EC anomalies. Second, RT and EC anomalies provide more clues for failure elimination. For example, for RT anomalies, the service operators can deploy more machines. But for RC anomalies, no clear instructions can be provided. A more detailed analysis is provided in Appendix B.

4.1.3 Anomaly Comparison Window. Given the alerting time t, data within $(t-10min, t]$ are considered to sculpture the failure, named a *detection window*. Besides, a *comparison window* is needed to learn the metrics' normal patterns. Considering the periodicity of metrics, we also select those time windows at the same period of the previous day and on the same day of the previous week. In other words, we consider three windows for comparison, *i.e.*, $(t-70min, t-10min]$, $(t-60min-24h, t-24h]$, $(t-60min-7*24h, t-7*24h]$.

4.1.4 Anomaly Detection. **Performance Anomaly Detection:** We choose the isolated forest (iForest) [18] to detect abnormal RT fluctuations. iForest identifies anomalies by explicitly isolating anomalies instead of normal instance profiling. Besides, it works well in high-dimensional problems and works in an unsupervised way. We use the same method as [16] to extract four features for RT fluctuations. The first two are computed by comparing the values in the detection window with the maximum value in the comparison window. Specifically, they are the value count in the detection window exceeding the maximum value in the comparison window and the difference between the maximum values of the two windows. The last two are computed by comparing the values in the detection window with the maximum moving average value in the comparison window. Specifically, they are the value count in the detection window exceeding the maximum moving average

value in the comparison window and the proportion of the mean in the detection window to the maximum moving average value in the comparison window. At last, we combine four features with three comparison windows to get 12 features for model training. Implementation details can be found in Appendix C.1.

Reliability Anomaly Detection: We propose a simple method to detect EC anomalies. The 95^{th} percentile of EC in the comparison period is used as the splitting point of the normal and abnormal values. We calculate the 95^{th} quantiles of three comparison windows and take their minimum min_{95}. We compare each of the 10 values in the detection window with min_{95}. The EC metric is considered abnormal if there is more than one value greater than min_{95}.

It is noteworthy that other existing anomaly detection algorithms [2, 23] can also be used in this stage as needed.

4.2 Ambiguity-free Call Graph Construction

This module aims to build the ambiguity-free call graph for abnormal entry calls provided by the **Metric Anomaly Detection** module. First, we generate the call graph starting from each abnormal entry call by extending downstream, adopting a breadth-first search (BFS), as we only explore RT and EC root causes downstream opposite their propagation direction. Then we apply AmSitor introduced in Section 3.3 to solve the ambiguous problem.

Take the bottom-left part of Figure 3 as an example where the abnormal entry call is *C1*. Add the callee of the entry call to a list, and find the related calls of the list nodes, where the nodes are the caller or callee of these related calls. If the callee of a newly found call is not in the list, add the callee to the list. Iterate this process until all nodes in the list are traversed and no new nodes are added. Finally, the graphs are merged if there are overlapping parts. Among the calls, a call added only as its callee is in the list is called an extra call (for example, *eC0*). Extra calls do not participate in anomaly detection or root cause exploration. They are only used to disambiguate for AmSits. For each AmSit, the correspondences of upstream and downstream calls are identified using AmSitor. Since root cause exploration is from upstream to downstream, a list of downstream calls corresponding to each upstream call is stored. For example, *C3*:{*C6*} in the figure indicates that the downstream list of *C3* is {*C6*}. As a result, we get the ambiguity-free call graph.

4.3 Root Cause Exploration

Based on the constructed graph from the previous section, we traverse the graph using a depth-first search (DFS) method to explore the potential root causes. Three different pruning methods are used to filter out the non-root causes. The bottom-right part of Figure 3 shows their examples.

AmSit-based Pruning (ASP). For a current call, if it and its upstream calls form an AmSit, the call and its downstream will be pruned by ASP when the call is not the downstream call corresponding to the path (*e.g.*, *C5* is not downstream of *C3*).

Metric Similarity-based Pruning (MSP). This analysis is based on an assumption from MicroHECL [16] that two successive calls in an anomaly propagation chain have similar change trends in corresponding metrics. For a current call, check the Pearson correlation coefficient between the anomaly metric of its upstream call and its corresponding metric based on the period of the last 30 minutes before the alert. If the coefficient is lower than a threshold (*e.g.*, 0.7), the call and its downstream will be pruned by MSP.

Anomaly Detection-based Pruning (ADP). There is a general consensus [16]: if the metric of one call on the exploration path is normal, then it and its subsequent calls will not be the root causes and can be dropped. Specifically, for a current call, perform anomaly detection for the corresponding metric. If the metric is normal, the call and its downstream will be pruned by ADP.

After the DFS exploration, CMDiagnostor adds the end-most nodes on all paths in the pruned graph to the root cause list, *e.g.*, node C and node F in the figure. The root cause node lists from RT and EC are merged to obtain the final candidate root cause node list. In addition, as required for subsequent ranking, we also store the calls corresponding to these end-most nodes (*e.g.*, *C2* and *C3*) and their three types of metrics (RC, RT, EC).

4.4 Candidate Root Cause Ranking

After root cause exploration, we get a candidate root cause node (method) list. To get the final root cause service ranking, we introduce three ranking keys as follows.

4.4.1 Root Cause Node Count (RCNC). Many nodes (methods) belonging to different services may exist in the list. We introduce RCNC because we found in other historical datasets that the root cause probability of a microservice is positively correlated with its RCNC, *i.e.*, the more root cause nodes in a service, the more likely the service is the root cause service. So we count the RCNC of each service to rank services. The larger the RCNC, the higher the service ranking.

4.4.2 Average Error Rate (AER). AER is a commonly used metric for ranking root causes. We first find the call corresponding to each root cause node and calculate the mean of error rates ($error\ rate = EC/RC$) of 10 values in the detection window. Then we calculate the average of the error rate means of calls in service as the AER of service. The higher the AER, the higher the service ranking.

4.4.3 Maximum Metric Similarity (MMS). MMS is introduced based on existing works [14, 16] that think the anomaly metric of the entry has similar change trends to that of the root cause. Specifically, we first find the root cause call corresponding to each root cause node and then calculate the Pearson correlation coefficient of the anomaly metric (RT or EC) with the corresponding metric of its entry call. Use the coefficient maximum (*i.e.*, MMS) of each service as the key. The higher the MMS, the higher the ranking.

We rank services with RCNC as the primary key and AER as the secondary key. MMS is used to rank MicroHECL in Section 5.1.3. We discuss the applicability of the ranking keys above in Section 5.4.3.

5 EVALUATION

In this section, we conduct experimental studies to answer the following research questions. All the experiments are conducted on a server with 22-core CPU (Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz) and 57GB RAM.

RQ1 How does AmSitor perform in identifying upstream and downstream correspondences in AmSits?

RQ2 How does CMDiagnostor perform in root cause localization?

RQ3 Can each component of CMDiagnostor contribute to the overall performance?

RQ4 Can AmSitor improve the performance of the other root cause localization approaches?

5.1 Experimental Setup

In this section, we first introduce the datasets we used for evaluation. Then we introduce the baselines and metrics for the AmSit problem and RCL task correspondingly.

5.1.1 Datasets. **Simulated Trace Dataset ($\mathcal{D}_{ST}$).** We generate $\mathcal{D}_{ST}$ based on the train-ticket testbed [29], which is a benchmark system of microservice architecture with 41 microservices. $\mathcal{D}_{ST}$ spans 1 hour with 150,000 span records.

Real-world Trace Dataset ($\mathcal{D}_{RT}$). We collect $\mathcal{D}_{RT}$ from a large-scale online advertising service system with 1.8K+ microservices and 0.23 million service nodes. $\mathcal{D}_{RT}$ contains 0.5 billion span records lasting 6 hours.

Real-world Failure Dataset ($\mathcal{D}_{RF}$). $\mathcal{D}_{RF}$ contains 65 real-world failures from an online service system of a top-tier global multimedia service provider. The system consists of 8K+ microservices and 0.57 million service nodes. We collect $\mathcal{D}_{RF}$ from December 2021 to July 2022. The content of each data record is shown in Table 1. On average, there are 0.8 million CMD records per day.

5.1.2 AmSit Problem.

Baselines. To the best of our knowledge, we are the first to address the ambiguity in CMD. As a result, we find no existing solutions for AmSit. Hence, we propose the following two straw man methods as baselines instead.

PWS: *Pair-Wise Similarity* (PWS) determines the relationship between a downstream call and each upstream one based on the intuitive traffic similarity. Specifically, PWS measures the similarity with the Pearson coefficient. A relation will be removed if the similarity is lower than a threshold. In this paper, we select the threshold of 0.3, which has the best overall performance.

CBS: The traffic of a downstream call may come from multiple upstream calls. As a result, the similarity between the downstream traffic and each upstream traffic can be intangible. *Combination-Based Similarity* (CBS) is proposed based on PWS. Specifically speaking, given a downstream call, CBS sums up the traffic for each combination of upstream calls. Calls in the combination with the maximal similarity will be considered as the actual upstream calls of the given downstream one.

Evaluation Metrics. We adopt accuracy, precision, and recall of AmSit cases to evaluate the performance of each AmSit solution. An AmSit case refers to a downstream call and the set of corresponding upstream ones S_u. Let N be the number of AmSit cases in a dataset and $\hat{\mathrm{S}}_u$ be the set of upstream calls identified by a AmSit solutions. Eq.(1) shows the definitions for the precision and recall for i^{th} AmSit case, as well as the accuracy. We present the average of the precision and recall in the rest of this section.

$$\begin{aligned} Accuracy &= |\{i \mid \mathrm{S}_{u,i} \equiv \hat{\mathrm{S}}_{u,i}, i = 1, 2, \cdots, N\}| \,/\, N \\ Precision_i &= |\mathrm{S}_{u,i} \cap \hat{\mathrm{S}}_{u,i}| \,/\, |\hat{\mathrm{S}}_{u,i}| \\ Recall_i &= |\mathrm{S}_{u,i} \cap \hat{\mathrm{S}}_{u,i}| \,/\, |\mathrm{S}_{u,i}| \end{aligned} \tag{1}$$

5.1.3 Root Cause Localization.

Baselines. Existing root-cause service localization works are mainly of two categories, i.e., topological graph-based and causal graph-based. We choose the state-of-the-art approaches from each as baselines, i.e., MonitorRank [10] (topological graph-based random walk), MicroHECL [16] (topological graph-based DFS), Microscope [14] (causal graph-based DFS), and AutoMap [22] (causal graph-based random walk). As the high time complexity of the PC algorithm, it is infeasible for Microscope and AutoMap to work in the method level. We transform the dataset they use to the service level. MonitorRank and MicroHECL use the same method-level CMD as CMDiagnostor. More details can be found in Appendix C.2.

Beyond these two categories, there are also some other categories of root cause analysis approaches, such as trace-based [13, 27], log analysis-based [7, 15], and direct KPI correlation [19] approaches. We do not choose baselines from these works since their inputs and outputs are much different from our scenario.

Evaluation Metrics. Following the existing works [16], we adopt the top-k hit ratio (HR@k) and mean reciprocal rank (MRR) to measure the performance of each RCL method. HR@k represents the proportion of the top-k candidate root cause lists containing the actual root causes. For a given failure case, the reciprocal rank is the multiplicative inverse of the rank of the first correct answer. MRR further averages the reciprocal ranks in the dataset. If the correct answer of a case is not included in the list, its reciprocal rank is taken as 0.

5.2 Performance of AmSitor (RQ1)

It is impractical to obtain explicit call correspondences in AmSits from $\mathcal{D}_{RF}$. For evaluating AmSitor (Algorithm 1), we use both simulated and real-world trace datasets ($\mathcal{D}_{ST}$ and $\mathcal{D}_{RT}$) that can obtain actual call correspondences. Before the evaluation, we first transform the trace data into CMD with a window size of 1 minute.

5.2.1 Performance in $\mathcal{D}_{ST}$. The experimental results are shown in the left part of Table 2. We perform data transformation with *operationName* as the node, and get 71 different calls and 13 AmSits. The AmSits in $\mathcal{D}_{ST}$ are very simple, with 2 to 3 upstream calls corresponding to one downstream call. Each upstream call execution will inevitably invoke its downstream call. Therefore, AmSits in the train-ticket testbed are simple to solve, which is confirmed by the high precision of all methods in Table 2.

PWS determines the correspondence of the downstream call and each upstream call by their traffic similarity. Once the traffic of the downstream call comes from multiple upstream calls, the similarity between the downstream traffic and each upstream traffic is less obvious. This makes PWS not always effective. Both CBS and AmSitor can handle these AmSits with complete accuracy. However, CBS assumes that a request execution will invariably invoke each relevant call once and only once. But this assumption may not always hold, especially in some complex environments. To evaluate the performance of the three methods in a more complex real-world service environment, we use $\mathcal{D}_{RT}$ for further evaluation.

5.2.2 Performance in $\mathcal{D}_{RT}$. The experimental results are shown in the right part of Table 2. We get 0.2 million different method calls

Table 2: Comparison among AmSit solutions in $\mathcal{D}_{ST}$ and $\mathcal{D}_{RT}$ (A = Accuracy, P = Average Precision, R = Average Recall).

Method	$\mathcal{D}_{ST}$			$\mathcal{D}_{RT}$		
	A	P	R	A	P	R
PWS	61.54%	1	0.87	52.29%	0.69	0.76
CBS	100%	1	1	55.13%	0.73	0.79
AmSitor	100%	1	1	89.51%	0.96	0.97

Table 3: Performance of AmSitor with different numbers of upstream calls.

#Upstream Call	Frequency of AmSit	Accuracy	Average Precision	Average Recall
$[2, 10]$	88%	95.67%	0.98	0.99
$[11, 100]$	11%	80.75%	0.92	0.96
$[101, \infty)$	1%	39.31%	0.83	0.92
$[2, \infty)$	100%	89.51%	0.96	0.97

by data transformation and find 1,428 AmSits. These AmSits cover 277,236 possible upstream-downstream pairs and 222,621 actual upstream-downstream pairs. The ideal reduction rate is 19.70%.

We aim to determine the right upstream calls for each downstream call in these AmSits. There are 12,084 upstream and 9,323 downstream calls, so 9,323 AmSit cases can be used for evaluation. AmSitor effectively identifies the upstream and downstream correspondences in real-world AmSits, as shown in Table 2. The accuracy of AmSitor achieves 89.51%, significantly outperforming the two baseline methods by over 44%. The actual reduction rate, *i.e.*, the number of upstream-downstream pairs removed by AmSitor divided by the total pair number, is 17.67%, close to the ideal value.

We note that the performance of AmSitor is affected by the upstream call number, as shown in Table 3. As the number increases, the accuracy of AmSitor decreases. But AmSitor still achieves higher precision and recall than baseline methods. We dig into the upstream calls that are mistakenly identified and find that 87% of them occur no more than 20 times during the 6-hour period. We believe that calls of this magnitude would have little impact on RCL, even if misidentified.

5.3 Performance of CMDiagnostor (RQ2)

We evaluate the effectiveness and efficiency of CMDiagnostor compared with four baselines by using the real-world failure dataset $\mathcal{D}_{RF}$. We do not use $\mathcal{D}_{ST}$ for this evaluation because the AmSits in the train-ticket testbed are too simple and not needed for handling, which are not in line with the characteristics of real-world AmSits in large-scale service systems. We do not use $\mathcal{D}_{RT}$ because we do not have a collection spanning many days and any failure case.

5.3.1 Effectiveness Evaluation. Table 4 compares different RCL approaches, where CMDiagnostor achieves HR@5 and MRR of 0.94 and 0.83, both outperforming the strongest baseline MicroHECL [16] by 0.14. Each component of CMDiagnostor contributes to the overall performance, including metric anomaly detection, ambiguity-free call graph construction, root cause exploration, and candidate root cause ranking. The detailed ablation studies can be found in Section 5.4 and Appendix E.

Table 4: Comparison among RCL approaches.

Method	HR@1	HR@3	HR@5	MRR
MonitorRank	0.35	0.60	0.63	0.53
Microscope	0.42	0.66	0.68	0.59
MicroHECL	0.49	0.77	0.80	0.69
AutoMap	0.22	0.74	0.78	0.58
CMDiagnostor	**0.65**	**0.91**	**0.94**	**0.83**

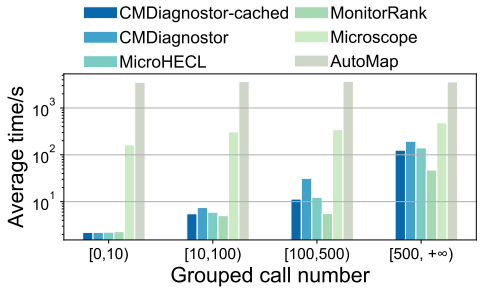


Figure 4: Average execution time for different RCL approaches with respect to the group of the explored call number, which is positively related to the complexity of the task.

5.3.2 Efficiency Evaluation. Figure 4 compares the time taken by different approaches to locate the root causes of failure cases. We group the cases by the actual number of explored calls (after pruning) of CMDiagnostor. Note that the identified correspondences in AmSits can be cached during actual deployment. So we accelerate CMDiagnostor by reading the cached AmSits, denoted as CMDiagnostor-cached in Figure 4. It can be seen that the efficiency of CMDiagnostor-cached is slightly higher than that of MicroHECL as CMDiagnostor-cached prunes some extra calls. Microscope and AutoMap are inefficient as it is time-consuming to build causal graphs in a large service system. The average analysis time of these approaches increases with the number of exploration calls.

5.4 Ablation Study of CMDiagnostor (RQ3)

In this section, we conduct ablation studies to evaluate the contribution of each component of CMDiagnostor to the overall performance, including the ambiguity-free call graph construction, pruning strategies, metric anomaly detection, and candidate root cause ranking. Since the metric anomaly detection and candidate root cause ranking modules can not be completely removed in the ablation study, we evaluate their effectiveness by replacing each method with alternative methods separately.

5.4.1 Contribution of Ambiguity-Free Call Graph Construction and Pruning Strategies. Three pruning strategies are proposed in Section 4.3. Among them, ADP is a basic one because, without it, root cause exploration may indiscriminately explore to the end of each path, which is not in line with root cause localization. Therefore, we focus on analyzing the contribution of ASP and MSP and conduct 3 comparison experiments, *i.e.*, not use ASP and MSP, only use MSP, and only use ASP. The results are shown in Table 5, which show that both ASP and MSP have considerable improvement effects, and the performance can be improved one step further when combining two pruning strategies. Furthermore, from the ablation of ASP, we

Table 5: Ablation analysis by removing each pruning strategy.

Method	HR@1	HR@3	HR@5	MRR
CMDiagnostor	**0.65**	**0.91**	**0.94**	**0.83**
w/o ASP	0.63	0.89	0.91	0.81
w/o MSP	0.60	0.88	0.89	0.79
w/o ASP, MSP	0.57	0.85	0.85	0.76

Table 6: Comparisons of anomaly detection methods.

No.	RT	EC	HR@1	HR@3	HR@5	MRR
1	iForest	3σ-1CW	0.62	0.88	0.91	0.80
2	iForest	3σ-3CW	0.60	0.91	0.94	0.82
3	iForest	iForest	0.58	0.89	0.92	0.80
4	iForest	OC-SVM	0.54	0.88	0.91	0.78
5	iForest	SPOT	0.58	0.85	0.89	0.77
6	3σ-1CW	95-3CW	0.55	0.82	0.83	0.73
7	3σ-3CW	95-3CW	0.52	0.85	0.86	0.74
8	95-3CW	95-3CW	0.55	0.91	0.94	0.80
9	OC-SVM	95-3CW	0.58	0.86	0.91	0.78
10	SPOT	95-3CW	0.52	0.80	0.82	0.71
Ours	**iForest**	**95-3CW**	**0.65**	**0.91**	**0.94**	**0.83**

Table 7: Comparisons of ranking strategies.

Method	Primary Key	Secondary Key	HR@1	HR@3	HR@5	MRR
1	RCNC	-	0.62	0.88	0.91	0.80
2	AER	-	0.51	0.86	0.89	0.75
3	MMS	-	0.11	0.65	0.8	0.52
4	AMS	-	0.15	0.63	0.78	0.52
5	MMS	AER	0.46	0.83	0.89	0.73
6	AMS	AER	0.38	0.71	0.80	0.63
Ours	**RCNC**	**AER**	**0.65**	**0.91**	**0.94**	**0.83**

also conclude that ambiguity-free call graph construction has a significant contribution to the overall performance.

5.4.2 Performance of Anomaly Detection Methods. To investigate the performance of our anomaly detection methods, we perform ablation studies by replacing one of them with alternative methods. Several common unsupervised methods are selected, including 3σ-based methods that respectively use the one hour before the alert window as the comparison window (CW) and the three comparison windows we use, one class support vector machine (OC-SVM) [12] using 12 features the same as our performance anomaly detection, and the SPOT algorithm [17]. Table 6 shows the experimental results, which indicate that our methods are the most effective.

5.4.3 Performance of Ranking Method. The ablation studies for different ranking strategies are shown in Table 7. According to Section 4.4, various indicators, *i.e.*, RCNC, AER, and MMS, are used for ranking. We also compare the service's average metric similarity (AMS). Considering there are the same values for MMS, AMS, and RCNC, a secondary key is used to improve their performance. The results show our method achieves the best performance with RCNC as the primary key and AER as the secondary key.

5.5 Performance of Baselines enhanced by AmSitor (RQ4)

We are the first to investigate the AmSit problem and propose AmSitor to eliminate its side effects. AmSitor can also be used by existing RCL methods to improve their performance. In this section, we conduct experiments to see whether AmSitor can improve their performance. As shown in Table 8, AmSitor improves four baselines.

Table 8: Performance of baselines with/without AmSitor, where + indicates equipping AmSitor.

Approach	HR@1	HR@3	HR@5	MRR
MonitorRank	0.35	0.60	0.63	0.53
MonitorRank(+)	**0.37**	**0.62**	**0.66**	**0.55**
Microscope	0.42	0.66	0.68	0.59
Microscope(+)	**0.45**	**0.66**	**0.69**	**0.60**
MicroHECL	0.46	0.77	0.80	0.69
MicroHECL(+)	**0.46**	**0.83**	**0.89**	**0.73**
AutoMap	0.22	0.74	0.78	0.58
AutoMap(+)	**0.31**	**0.82**	**0.82**	**0.65**

6 RELATED WORK

There are mainly two categories of root cause service localization approaches based on call metrics, *i.e.*, causal graph-based and topological graph-based. The causal graph-based approaches use causal discovery algorithms to identify the relationships among service components. For example, PC algorithm [9, 25] is widely used [3, 14, 20, 22]. Then, they apply various inference methods to identify the root causes, such as the random walk [22] or DFS [3, 14]. We refer readers to a recent survey [6] for more details. However, they are usually limited by the low efficiency and accuracy of causal discovery. Therefore, many approaches use call graphs, which can be obtained by sensors deployed on service components [10], to represent the dependencies among the components. However, these methods suffer from the AmSit problem, limiting their performance.

Root cause service localization based on traces [13, 28] and logs [7, 15] are also extensively studied. However, traces are collected by distributed tracing systems and consume much computation and storage resources, and thus, are unavailable in many online services. Logs are arbitrary and ad-hoc and contain much useless information, which limits the accuracy of root cause localization. Therefore, call metrics-based approaches are much more practical.

MicroHECL [16] is a state-of-the-art topological graph-based DFS method using CMD and the MSP pruning strategy to improve efficiency. Compared with MicroHECL, CMDiagnostor uses AmSitor to disambiguate the call graph and only explores abnormal RT and EC metrics downstream from the entry node instead of upstream. It uses unsupervised anomaly detection methods instead of supervised ones. In addition, it ranks services based on the number of root cause nodes in each service.

7 CONCLUSIONS

This paper proposes CMDiagnostor, an ambiguity-aware root cause localization approach based on call metric data. Though call metric data are widely used, the ambiguity existing in the call graph is ignored, which may lead to misjudgments on failure root causes. A simple but effective method, AmSitor, is proposed to address the ambiguity problem and integrated into CMDiagnostor. Significant improvements on a large-scale real-world dataset verify the effectiveness and efficiency of CMDiagnostor.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China (No.2019YFE0105500) and the Research Council of Norway (No.309494), the State Key Program of National Natural Science of China under Grant 62072264, and the National Natural Science Foundation of China (No.62272249 and No.62202445). In addition, we thank Chenyuan Hu, Hucheng Xie, and other relevant Tencent employees for their help in this work.

REFERENCES

[1] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site Reliability Engineering* (first ed.). O'Reilly Media, Inc.

[2] Mohammad Braei and Sebastian Wagner. 2020. Anomaly detection in univariate time-series: A survey on the state-of-the-art. *arXiv preprint arXiv:2004.00433* (2020).

[3] Pengfei Chen, Yong Qi, Pengfei Zheng, and Di Hou. 2014. CauseInfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. In *INFOCOM*. 1887–1895.

[4] SciPy community. 2022. scipy.optimize.dual_annealing. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.dual_annealing.html, Last accessed on 2022-10-11.

[5] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. 2007. X-trace: A pervasive network tracing framework. In *NSDI*.

[6] Ruocheng Guo, Lu Cheng, Jundong Li, P. Richard Hahn, and Huan Liu. 2020. A Survey of Learning Causality with Data: Problems and Methods. *ACM Comput. Surv.* 53, 4 (jul 2020), 37 pages.

[7] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2018. Identifying Impactful Service System Problems via Log Analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. https://doi.org/10.1145/3236024.3236083

[8] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. 2017. Canopy: An end-to-end performance tracing and analysis system. In *SOSP*. 34–50.

[9] Markus Kalisch and Peter Bühlman. 2007. Estimating high-dimensional directed acyclic graphs with the PC-algorithm. *Journal of Machine Learning Research* 8, 3 (2007).

[10] Myunghwan Kim, Roshan Sumbaly, and Sam Shah. 2013. Root cause detection in a service-oriented architecture. *ACM SIGMETRICS Performance Evaluation Review* 41, 1 (2013), 93–104.

[11] Scikit learn community. 2022. IsolationForest. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html, Last accessed on 2022-10-11.

[12] Scikit learn community. 2022. OneClassSVM. https://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html, Last accessed on 2022-10-11.

[13] Zeyan Li, Junjie Chen, Rui Jiao, Nengwen Zhao, Zhijun Wang, Shuwei Zhang, Yanjun Wu, Long Jiang, Leiqin Yan, Zikai Wang, et al. 2021. Practical Root Cause Localization for Microservice Systems via Trace Analysis. In *IWQoS*.

[14] JinJin Lin, Pengfei Chen, and Zibin Zheng. 2018. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In *ICSOC*. 3–20.

[15] Q. Lin, H. Zhang, J. Lou, Y. Zhang, and X. Chen. 2016. Log Clustering Based Problem Identification for Online Service Systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. https://doi.org/10.1145/2889160.2889232

[16] Dewei Liu, Chuan He, Xin Peng, Fan Lin, Chenxi Zhang, Shengfang Gong, Ziang Li, Jiayu Ou, and Zheshun Wu. 2021. MicroHECL: High-Efficient Root Cause Localization in Large-Scale Microservice Systems. In *ICSE-SEIP*. 338–347.

[17] Fengrui Liu. 2022. SpotDetector. https://streamad.readthedocs.io/en/latest/api/streamad.model.html#spotdetector, Last accessed on 2022-10-11.

[18] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation forest. In *2008 eighth ieee international conference on data mining*. IEEE, 413–422.

[19] Ping Liu, Yu Chen, Xiaohui Nie, Jing Zhu, Shenglin Zhang, Kaixin Sui, Ming Zhang, and Dan Pei. 2019. FluxRank: A Widely-Deployable Framework to Automatically Localizing Root Cause Machines for Software Service Failure Mitigation. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. https://doi.org/10.1109/ISSRE.2019.00014

[20] Xianglin Lu, Zhe Xie, Zeyan Li, Mingjie Li, Xiaohui Nie, Nengwen Zhao, Qingyang Yu, Shenglin Zhang, Kaixin Sui, Lin Zhu, and Dan Pei. 2022. Generic and Robust Performance Diagnosis via Causal Inference for OLTP Database Systems. In *2022 22th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*.

[21] Meng Ma, Weilan Lin, Disheng Pan, and Ping Wang. 2019. Ms-rank: Multi-metric and self-adaptive root cause diagnosis for microservice applications. In *ICWS*. 60–67.

[22] Meng Ma, Jingmin Xu, Yuan Wang, Pengfei Chen, Zonghua Zhang, and Ping Wang. 2020. Automap: Diagnose your microservice-based web applications automatically. In *WWW*. 246–258.

[23] Guansong Pang, Chunhua Shen, Longbing Cao, and Anton Van Den Hengel. 2021. Deep learning for anomaly detection: A review. *ACM Computing Surveys (CSUR)* 54, 2 (2021), 1–38.

[24] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).

[25] Peter Spirtes, Clark N Glymour, Richard Scheines, and David Heckerman. 2000. *Causation, prediction, and search*. MIT press.

[26] Sean Wolfe. 2018. Amazon's one hour of downtime on prime day may have cost it up to $100 million in lost sales. *Business Insider (July 2018). URL: https://www.businessinsider.com/amazon-prime-day-website-issues-cost-it-millions-in-lost-sales-2018-7* (2018).

[27] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. Microrca: Root cause localization of performance issues in microservices. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 1–9.

[28] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. 2021. MicroRank: End-to-End Latency Issue Localization with Extended Spectrum Analysis in Microservice Environments. In *WWW*. 3087–3098.

[29] Xiang Zhou, Xin Peng, Tao Xie, et al. 2018. Train Ticket: A Benchmark Microservice System. https://github.com/FudanSELab/train-ticket, Last accessed on 2022-10-11.

A MATHEMATICAL ANALYSIS OF TRAFFIC

Each time the user request is processed by a given caller A, the number of calls from A to B is determined by the business logic of the system, denoted as a random variable $n_{A\to B}$. Within each time slicing t, denote the number of calls from A to B as $N^{(t)}_{A\to B} = \sum_i n^{(t)}_{A\to B,i}$, where i is the index for different requests. Let θ_i be the context of i^{th} request, *e.g.*, the distribution of user input and the effect of middle-wares like caching. Hence, the probability distribution of $n_{A\to B,i}$ can be taken as a conditional one of $n_{A\to B}$, *i.e.*, $n_{A\to B,i} \sim P(n_{A\to B} \mid \theta_i)$. By grouping the contexts of all calls in the same time slicing, we obtain $N^{(t)}_{A\to B} = \sum_\theta \left[\sum_{j\in\{i\mid\theta_i^{(t)}=\theta\}} n^{(t)}_{A\to B,j}\right]$. Notice that elements within the brackets come from the same distribution $P(n_{A\to B} \mid \theta)$. With the Law of Large Numbers, we have (2) where $N^{(t)}_{A,\theta} = |\{i \mid \theta_i^{(t)} = \theta\}|$ is the number of requests starting from A at the given time t with the same context θ.

$$N^{(t)}_{A\to B} \to \sum_\theta N^{(t)}_{A,\theta}\mathbb{E}\left(n_{A\to B} \mid \theta\right) \tag{2}$$

In this work, we approximate θ in $P(n_{A\to B} \mid \theta)$ with A's upstream calls, as the structure of a trace, *i.e.*, how the system processes a user request, has to be consistent with what the user request is. So (2) can be extended into (3), where each U is a caller of A.

$$\begin{aligned} N^{(t)}_{A\to B} &\to \sum_\theta N^{(t)}_{A,\theta}\mathbb{E}\left(n_{A\to B} \mid \theta\right) \\ &\approx \sum_U N^{(t)}_{U\to A}\mathbb{E}\left(n_{A\to B} \mid U \to A\right) \end{aligned} \tag{3}$$

Then, we find the number of calls from A to B can be approximated as the weighted sum of the numbers of A's upstream calls.

A special case of $P(n_{A\to B} \mid \theta)$ is a Bernoulli distribution with a parameter of p_θ, *i.e.*, A will call B once with a probability of p_θ or not with a probability of $1 - p_\theta$. Under this circumstance, (3) can be further deduced to $\sum_U N^{(t)}_{U\to A} p_{U\to A}$. We use the expectation here for an arbitrary distribution.

B METRIC SELECTION

We choose RT and EC for root cause localization instead of RC because we do not think the RC anomaly is the failure root cause. We analyze two types of RC anomalies as follows.

- **Traffic is too high**: If the RC of a call is too high, the downstream call traffic will increase. As a result, some service nodes on the link cannot cope with such a large number of requests, causing an increase in RT or EC. Conversely, if the high RC does not cause an increase in RT or EC, then the high RC is not considered a root cause because it does not affect the service performance and reliability. We prefer to regard service nodes that cannot adapt to the current traffic when the traffic is too high as the root cause in order to solve the current problem through capacity expansion of the nodes.
- **Traffic is too low**: The traffic of a call is too low may be because relevant user requests are few currently (*e.g.*, early morning), which is not the root cause of failure. On the other hand, the related user requests are not few, but those reaching the caller of the call are few due to the reasons of the intermediate link. The EC of intermediate calls increases as the related request does not execute the necessary calls.

Therefore, we do not consider high or low traffic to be the root cause of failure. Because the root cause of high-traffic failure should be that some nodes can not respond correctly, and the root cause of low-traffic failure should be that some upstream calls do not execute properly. So we chose RT and EC instead of RC.

C IMPLEMENTATION DETAILS

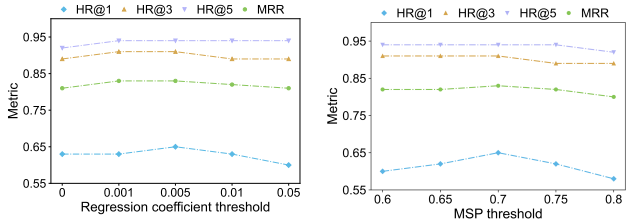
C.1 Performance Anomaly Detection

We use the Python-based machine learning framework *scikit-learn* [11] to implement the iForest model. IForest identifies anomalies by isolating them with shorter path lengths. A large number of samples will reduce the ability of iForest to isolate anomalies because normal samples will interfere with the isolation process. Small datasets tend to achieve better results and efficiency in iForest. Each training case corresponds to a 10-minute period with 12 features, so the dataset contains too many such 10-minute cases. Thus, sampling is necessary. We select a subset (*e.g.*, 200,000 cases in this paper) for training. Training cases are selected from historical data semi-randomly. Specifically, we randomly select a few days from each month, then randomly select some calls from each of those days, and finally, randomly select some 10-minute periods from the calls. The selected cases are used to train the iForest model. The model trained can output the prediction results (*i.e.*, normal or abnormal) after receiving 12 features of a detection window.

C.2 RCL approaches

Five RCL approaches are implemented as follows.

- **MonitorRank** [10]: It determines possible root causes of a failure by running a personalized PageRank algorithm on the topological graph. We adjust MonitorRank to start from each entry node to meet our context and rank services by the average ranking of methods in each service to meet the method level.
- **Microscope** [14]: It determines possible root causes of a failure by recursively visiting the causal graph built based on PC algorithm and enriched by service calls. As PC algorithm is challenging to handle a mass of method-level nodes, we transform the method-level data to service level. We also adjust it to start from each entry service instead of the front-end service.
- **MicroHECL** [16]: It determines possible root causes of a failure by extending anomaly propagation chains starting from the entry service. As its anomaly detection methods need labeled data lacking in our context, we use our methods in Section 4.1 instead. Since our cases have no traffic root causes, exploring upstream is unnecessary. So MicroHECL only explores downstream for RT and EC anomalies, as in our approach. MicroHECL ranks services with MMS as the primary key and AER as the secondary key, as described in Section 4.4, to meet the method level.
- **AutoMap** [22]: It determines possible root causes of a failure by running a heuristic random walk algorithm on the causal graph constructed by combining PC algorithm and multi-metric data. We transform the method-level data to the service level as in Microscope. In addition, we parallelize the original algorithm to make the time consumption acceptable in our scenario.



(a) regression coefficient threshold (b) MSP threshold

Figure 5: Impact of hyperparameters on CMDiagnostor.

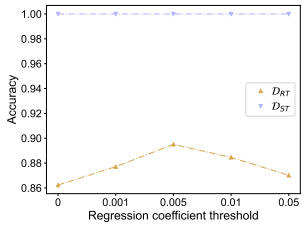


Figure 6: Impact of regression coefficient threshold on AmSitor.

- **CMDiagnostor**: Some default threshold settings are the following: the regression coefficient threshold used by AmSitor is 0.005; the correlation threshold used by the MSP pruning strategy is 0.7. In addition, the traffic interval used by AmSitor is the 24-hour period before the alert is generated.

D HYPERPARAMETER SENSITIVITY

There are two hyper-parameters in CMDiagnostor, i.e., the regression coefficient threshold used by AmSitor (Section 3.3) and the correlation threshold used by the MSP pruning strategy (Section 4.3). Their default settings are 0.005 and 0.7, respectively. The following results are based on the $\mathcal{D}_{RF}$ dataset.

Impact of regression coefficient threshold. Figure 5(a) shows the impacts of different regression coefficient thresholds on the effectiveness of CMDiagnostor. We find that the regression coefficient threshold does not impact the CMDiagnostor effectiveness much. The largest difference in the four metrics is 0.05 of HR@1, and the largest differences in the other three metrics are all 0.02. In addition, our setting of regression coefficient threshold (0.005) shows the best performance in the comparison.

Impact of MSP threshold. Figure 5(b) shows the impacts of different MSP thresholds on the effectiveness of CMDiagnostor. The MSP threshold also does not impact the CMDiagnostor effectiveness much. The largest difference in the four metrics is 0.07 of HR@1, and the largest differences in HR@3, HR@5, and MRR, are 0.02, 0.02, and 0.03, respectively. Our setting of MSP threshold (0.7) shows the best performance in the comparison.

We also discuss the impact of regression coefficient threshold on AmSitor using $\mathcal{D}_{ST}$ and $\mathcal{D}_{RT}$ datasets. Figure 6 shows the results. We find AmSitor is robust and insensitive to the threshold. 0.005 also performs the best compared with the other four values. When the algorithm is applied to datasets beyond this paper, we recommend adopting the optimal value in this paper as a starting point and experimentally searching for better ones.

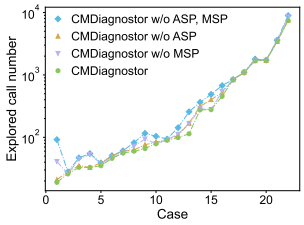


Figure 7: Explored call number after pruning.

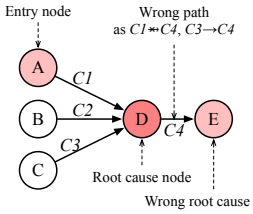


Figure 8: AmSit related to root cause node.

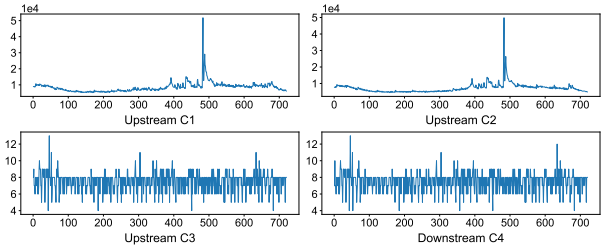


Figure 9: Traffic of related calls of an AmSit.

E CASE STUDY

E.1 Pruning Number Analysis in Failure Cases

As an extension of Section 5.4.1, this section explores the call number pruned in failure cases. According to statistics, 22 out of 65 cases encounter and deal with the AmSit problem. We count the number of calls explored using different pruning strategies for these cases, and the results are shown in Figure 7. More precise statistics are below. In the 22 cases, compared with method CMDiagnostor without ASP and MSP, the number of calls explored by method CMDiagnostor without ASP is reduced by 0.17 on average; the number of calls explored by method CMDiagnostor without MSP is on average a reduction of 0.14; our CMDiagnostor using both ASP and MSP explores an average reduction of 0.23 in the number of calls. This shows our pruning methods can improve efficiency.

E.2 Actual Effect of AmSitor on RCL

We use a real-world failure case to show the effect. For this case, CMDiagnostor can identify the actual root cause by using AmSitor but cannot identify the root cause without AmSitor. Figure 8 shows the main AmSit affecting the identification. When exploring from the entry node A, the method without AmSitor explores the call $C4$ while missing the root cause node D. Fortunately, CMDiagnostor identifies the upstream call of $C4$ is $C3$. Related traffic flows are shown in Figure 9, which intuitively illustrates the correctness of AmSitor. As a result, CMDiagnostor does not explore $C4$ and finds the actual root cause node D. This shows that AmSitor can improve real-world root cause exploration effectiveness.