Check for updates

The Potential of One-Shot Failure Root Cause Analysis: Collaboration of the Large Language Model and Small Classifier

Yongqi Han
2011438@tongji.edu.cn
Tongji University
Shanghai, China

Qingfeng Du[*]
du_cloud@tongji.edu.cn
Tongji University
Shanghai, China

Ying Huang
2231511@tongji.edu.cn
Tongji University
Shanghai, China

Jiaqi Wu
2211241@tongji.edu.cn
Tongji University
Shanghai, China

Fulong Tian[†]
fulong.tian@di-matrix.com
Di-Matrix
Shanghai, China

Cheng He[†]
cheng.he@di-matrix.com
Di-Matrix
Shanghai, China

ABSTRACT

Failure root cause analysis (RCA), which systematically identifies underlying faults, is essential for ensuring the reliability of widely adopted microservice-based applications and cloud-native systems. However, manual analysis by simple rules faces significant burdens due to the heterogeneous nature of resource entities and the massive amount of observability data. Furthermore, existing approaches for automating RCA struggle to perform in-depth fault analysis without extensive fault labels. To address the scarcity of fault labels, we examine an extreme RCA scenario where each fault type has only one example (one-shot). We propose *LasRCA*, a framework for one-hot RCA in cloud-native systems that leverages the collaboration of the large language model (LLM) and the small classifier. In the training stage, *LasRCA* initially trains a small classifier based on one-shot fault examples. The small classifier then iteratively selects high-confusion samples and receives feedback on their fault types from LLM-driven fault labeling. These samples are applied to retrain the small classifier. In the inference stage, *LasRCA* performs a joint RCA through the collaboration of the LLM and small classifier, achieving a trade-off between effectiveness and cost. Experiment results on public datasets with heterogeneous nature and prevalent fault types show the effectiveness of *LasRCA* in one-shot RCA.

CCS CONCEPTS

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

KEYWORDS

root cause analysis, cloud-native systems, multimodal data

ACM Reference Format:
Yongqi Han, Qingfeng Du, Ying Huang, Jiaqi Wu, Fulong Tian, and Cheng He. 2024. The Potential of One-Shot Failure Root Cause Analysis: Collaboration of the Large Language Model and Small Classifier. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695475

1 INTRODUCTION

In large IT enterprises, loosely-coupled and lightweight microservice applications have become the *de facto* standard for delivering core business [26, 40]. The evolving cloud computing technology has further enabled the seamless migration of microservice applications to the cloud-native architecture, which provides superior scalability, resiliency, and elasticity [16]. However, in such systems, failures are inevitable due to the large scale and complexity, which may result in significant costs in terms of customer impact [39, 56].

To minimize financial losses and impacts, accurate root cause analysis (RCA) is crucial for identifying underlying faults. Figure 1 shows the typical process of RCA for failures in cloud-native systems. Cloud-native systems are typically composed of various heterogeneous resource entities. We use three typical entities—*service*, *pod*, and *node*—as examples. For each *service* representing a microservice, there is a corresponding set of *pods*. Each *pods* bundles one or more containers together to handle incoming requests from the *service*. These *pods* are deployed on *nodes*, which are either virtual or physical machines. Once a fault occurs (such as the fault in the *pod* P_{d2} in Figure 1), it may propagate to other resource entities, ultimately resulting in abnormal user requests. To analyze such complex fault characteristics, observability data is systematically collected. It consists of traces, metrics, and logs, each serving a distinct purpose [41]. For RCA in such systems, site reliability engineers (SREs) typically first gain a preliminary understanding of the system's condition based on observability data. Subsequently, based on troubleshooting guides, SREs pinpoint the faulty resource entities, identify the exact fault type, and explain their decisions. However, due to the heterogeneous nature of resource entities and the massive amount of observability data, manual analysis by simple rules struggles to generalize, resulting in significant burdens.

Many efforts have been devoted to automating RCA to reduce such burdens. Existing approaches typically rely on unsupervised assumptions or supervised training [39]. Unsupervised approaches

---

[*] Qingfeng Du is the corresponding author.
[†] Di-Matrix is the Di-Matrix (Shanghai) Information Technology Co., Ltd.

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

ASE '24, Oct. 27 - Nov. 1, 2024, Sacramento, CA, USA

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695475

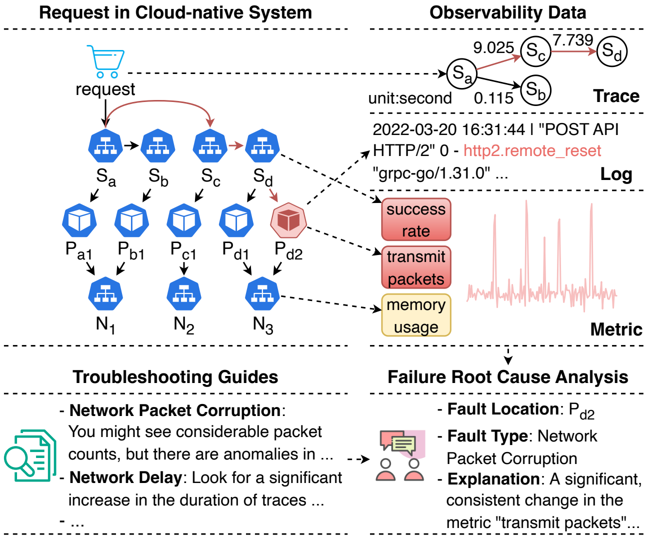

Request in Cloud-native System
Observability Data
request
9.025
7.739
S_a
S_b
S_c
S_d
unit:second
0.115
Trace
2022-03-20 16:31:44 | "POST API
HTTP/2" 0 - http2.remote_reset
"grpc-go/1.31.0" ...
Log
success rate
transmit packets
memory usage
Metric
P_{a1}
P_{b1}
P_{c1}
P_{d1}
P_{d2}
N_1
N_2
N_3
Troubleshooting Guides
Failure Root Cause Analysis
- Network Packet Corruption: You might see considerable packet counts, but there are anomalies in ...
- Network Delay: Look for a significant increase in the duration of traces ...
- ...
- Fault Location: P_{d2}
- Fault Type: Network Packet Corruption
- Explanation: A significant, consistent change in the metric "transmit packets"...

Figure 1: A typical RCA process in cloud-native systems. For $i \in \{a, b, c, d\}$, S_i denotes a *service*. For $j \in \{1, 2\}$, P_{ij} denotes a *pod* that actually receives and executes requests from S_i. And N_1-N_3 are the *nodes* where these *pods* are deployed.

do not need fault labels but are limited by their inability to incorporate fault knowledge for in-depth analysis. They only identify suspect resource entities or observability data. Without advancing to fault types, they often necessitate significant manual intervention. In contrast, supervised approaches offer detailed analysis but rely on sufficient fault labels, which are challenging to produce due to the required domain knowledge and time. Typically, the availability of fault labels is moderate. Troubleshooting guides or fault knowledge bases commonly provide a limited number of examples to aid SREs. The scarcity of fault labels necessitates exploring how to effectively use limited fault examples to automate RCA.

In the context of RCA conducted by SREs, understanding troubleshooting guides and possessing strong reasoning skills are crucial, especially with limited fault examples. The success of large language models (LLMs) like Generative Pretrained Transformer (GPT) [5, 47] indicates a promising pathway for imitating the RCA process. LLMs process natural language inputs, apply knowledge from training for analysis and reasoning, and generate succinct and insightful outputs, similar to the way SREs conduct RCA. However, despite their considerable capabilities, LLMs are costly in terms of both time and financial expense, especially in complex cloud-native systems. Therefore, depending entirely on LLMs to replicate the full scope of RCA is impractical. This motivates us to leverage the collaboration between a low-cost small classifier and a reasoning-capable LLM to achieve a trade-off between effectiveness and cost.

Our Vision and Idea. We consider an extreme scenario where each fault type has only one example (one-shot). In this scenario, we propose *LasRCA*, an approach for fault localization and fault type classification. *LasRCA* is based on the LLM and the small classifier, operating in two stages. In the training stage, *LasRCA* initially trains a small classifier based on one-shot fault examples. The small classifier then iteratively selects high-confusion samples and receives feedback on their fault types from LLM-driven fault labeling. In the inference stage, the small classifier determines the scope and type of the fault. Then the LLM conducts RCA based on the small model's confidence and provides explanations.

Contributions. The main contributions are as follows:
(1) We propose *LasRCA*, a one-shot root cause analysis approach in cloud-native systems based on observability data and troubleshooting guides to overcome limitations in fault labeling.
(2) We examine the division of labor between the large language model, the small classifier, and site reliability engineers, proposing a cost-effective collaborative mechanism that leverages both the LLM and the small classifier for effective root cause analysis.
(3) Experiment results on public datasets with heterogeneous graph structures and prevalent fault types show the effectiveness of *LasRCA* in one-shot root cause analysis.
(4) We release the source code of *LasRCA* with data [12], facilitating practitioners to replicate and extend *LasRCA*.

2 BACKGROUND

2.1 Preliminaries

Resource entities. In cloud-native systems, there are various resources for allocation and management. Those that possess the distinct management and monitoring value are classified under **resource entities**, such as *service*, *pod*, and *node* in Figure 1.

Failures and faults. A **failure** is an event that represents an undesired deviation in service delivery, and its underlying root cause is a **fault** [2, 32]. A fault label indicates the fault type of a resource entity within a time period. A fault example indicates the labels of all resource entities during the occurrence of a fault.

Observability is the ability to monitor system behavior via its external outputs. Cloud-native **observability** relies on 3 elements: **traces**, **metrics**, and **logs**. **Traces** capture call paths of responses from microservice applications to user requests, offering insights into the behavior and performance of microservices. **Metrics** are quantitative measurements that track changes in resource entities over time, commonly organized in a time series format. **Logs** are semi-structured text recording events within cloud-native systems.

Large language models are huge language models pretrained on extensive datasets, without requiring tuning on data for specific tasks [47]. Recent advancements in **LLMs** (such as GPT-4 [33]) have demonstrated superior performance in natural language understanding, generation, and reasoning albeit at high costs.

Small classifiers are low-cost, label-dependent models trained on specific datasets for classification. Neural network classifiers have achieved significant success, with Transformer [43] and Graph Attention Network (GAT) [4] being two widely used components for effectively modeling of sequence and graph structures, respectively.

2.2 Motivation

Given the scarcity of fault examples discussed in Section 1, we make several attempts to reduce the reliance on the scale of fault examples. To ensure attempts only focus on strategies for reducing the reliance on fault examples, all attempts align with the feature engineering process and the small classifier Φ_s in Section 4.1.2, using statistical features from traces, log features extracted by Drain [13] and TF-IDF [37], and metric features as input. We then present results from

Table 1: Mean and standard deviation of MicroF1 and MacroF1 for several one-shot RCA attempts (5 runs for each attempt).

Attempt	MiF1	MaF1
Vanilla Training	0.4062 ± 0.0218	0.3873 ± 0.0189
Pretrain and Fine-tuning	0.0146 ± 0.0014	0.0508 ± 0.0124
Semi-Supervised Learning	0.4573 ± 0.0283	0.4467 ± 0.0297

representative methods on dataset $\mathcal{B}$ as examples. Details of $\mathcal{B}$ and evaluation metrics are provided in Section 5.2.1 and 5.2.3.

2.2.1 Attempt 1: Pretraining and Fine-tuning. Pretraining and fine-tuning have become mainstream paradigms for enhancing performance while reducing label dependence. They aim to learn general representations by pretraining and then optimize model parameters with limited labeled data during fine-tuning. As an example, we pretrain a time-series model for each resource entity type using the widely adopted TS2Vec [51]. For fine-tuning, we replace resource entity representations in Φ_s with the pretrained ones and adjust the remaining parameters in Φ_s to predict fault types. Table 1 shows the experiment results. The performance is worse than using the original features with the same one-shot fault examples. This is mainly due to the mismatch between RCA-relevant features and unsupervised assumptions in pretraining, leading to a loss of crucial information. While pretraining methods like TS2Vec can model multivariate time series, they focus on capturing overall patterns. However, RCA often requires focusing on significant changes in specific subsets of observability features (e.g., CPU or network metrics). This mismatch reduces the effectiveness of these methods, highlighting the challenges of finding unsupervised assumptions to derive RCA-relevant features without prior knowledge.

Findings: Deriving RCA-relevant features based on unsupervised assumptions without prior knowledge is challenging.

2.2.2 Attempt 2: Semi-Supervised Learning. Semi-supervised learning is also popular for training models on large datasets with limited labels. It typically employs consistency regularization [17] to generate pseudo labels for unlabeled data by randomly modifying inputs and leveraging the model's predictions. We select FixMatch [38], a leading method that applies Mixup [53] as its "strong augmentation" and Gaussian noise (mean 0, standard deviation 0.01) as its "weak augmentation". Table 1 shows the experiment results. The performance improvement is limited for two main reasons. First, robust data augmentation for RCA is uncertain. Traditional data augmentation like adding noise may either be too subtle or risk distorting the fault type. Figure 2 shows how Gaussian noise affects different features. Both features are standardized by z-score [18] ($z = \frac{X-\mu}{\sigma}$ where μ is the mean and σ is the standard deviation). Second, limited labels lead to more errors in pseudo labels, increasing the likelihood of introducing erroneous labels in iterative training. Hence, labeling high-quality fault examples remains essential.

Findings: Expanding fault samples by semi-supervised learning is limited by data augmentation challenges and errors in pseudo labels. Labeling high-quality fault examples remains essential.

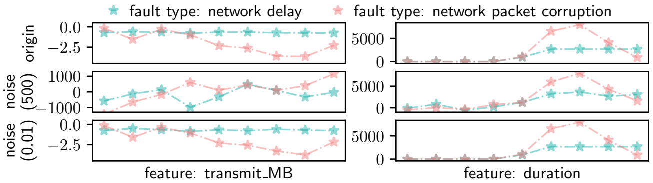


Figure 2: Gaussian noise augmentation at scales of 500 and 0.01. A scale of 500 disrupts the decreasing 'transmit_MB' feature in packet corruption but minimally affects 'duration'.

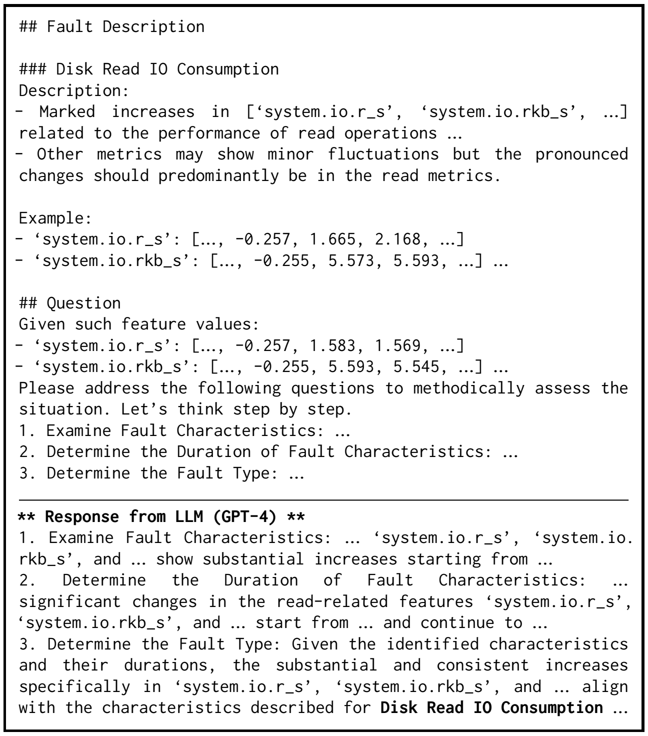
2.2.3 Attempt 3: LLM-driven RCA. Based on previous findings and recent advancements in LLMs, we attempt to leverage LLMs for RCA based on troubleshooting guides and evaluate their ability to identify potential fault types. We slightly modify troubleshooting guides to make them suitable for LLM reading. Then we insert labeled data from one-shot fault examples to enhance reasoning. Figure 3 shows an LLM-driven RCA example for a *node* disk fault using GPT-4 [33] with content condensed for brevity. The example shows that LLMs can identify fault types and provide user-friendly explanations using troubleshooting guides with a designed analysis chain. Thus, LLMs can simulate the SRE process of reading guides, analyzing observability data, and making decisions. However, the costs in terms of both time and financial expense are significant. In our experiments, the average token count for troubleshooting guides is 1386.5. The financial cost for a single input to identify one fault type in a resource entity is approximately \$0.04 [34], with a time cost of around 15 seconds for each complete output. For cloud-native systems containing dozens or hundreds of resource entities and different fault types, relying entirely on LLMs to replicate the full scope of RCA is prohibitively expensive and impractical.

Findings: When appropriately configured, LLMs can effectively perform RCA based on fault characteristics from troubleshooting guides. However, the high financial and time costs make them impractical for replicating the full scope of RCA.

2.2.4 Motivation for LasRCA. Based on the aforementioned findings, we consider reducing LLM decisions to lower costs. We aim to use a small classifier to narrow down the analysis scope. However, as shown in Table 1, the effectiveness of the small classifier in one-shot RCA is insufficient for practical use. To improve this, we propose using LLMs to expand fault labels to assist in training the small classifier, enhancing its performance. Through the collaboration between the LLM and the small classifier, we aim to achieve effective and low-cost RCA. Moreover, since LLMs offer user-friendly explanations, SREs can correct LLM misjudgments during the classifier's iterative training. Compared to other LLM-driven approaches focusing on retrieving and summarizing incidents, such as RCACopilot [7], *LasRCA* targets a complementary direction. *LasRCA* aims to maximize the utilization of LLMs and fault examples in one-shot RCA while reducing unnecessary costs.

3 PROBLEM DEFINITION

We focus on resource entity localization and fault type classification in one-shot RCA. In heterogeneous cloud-native systems, $\mathcal{E} = \{e \mid e \in \mathcal{E}_i, 1 \le i \le \kappa\}$ denotes the set of resource entities, where $\mathcal{E}_i$



```
## Fault Description

### Disk Read IO Consumption
Description:
- Marked increases in ['system.io.r_s', 'system.io.rkb_s', ...]
related to the performance of read operations ...
- Other metrics may show minor fluctuations but the pronounced
changes should predominantly be in the read metrics.

Example:
- 'system.io.r_s': [..., -0.257, 1.665, 2.168, ...]
- 'system.io.rkb_s': [..., -0.255, 5.573, 5.593, ...] ...

## Question
Given such feature values:
- 'system.io.r_s': [..., -0.257, 1.583, 1.569, ...]
- 'system.io.rkb_s': [..., -0.255, 5.593, 5.545, ...] ...
Please address the following questions to methodically assess the
situation. Let's think step by step.
1. Examine Fault Characteristics: ...
2. Determine the Duration of Fault Characteristics: ...
3. Determine the Fault Type: ...
```

```
** Response from LLM (GPT-4) **
1. Examine Fault Characteristics: ... 'system.io.r_s', 'system.io.
rkb_s', and ... show substantial increases starting from ...
2. Determine the Duration of Fault Characteristics: ...
significant changes in the read-related features 'system.io.r_s',
'system.io.rkb_s', and ... start from ... and continue to ...
3. Determine the Fault Type: Given the identified characteristics
and their durations, the substantial and consistent increases
specifically in 'system.io.r_s', 'system.io.rkb_s', and ... align
with the characteristics described for Disk Read IO Consumption ...
```

Figure 3: An LLM-driven RCA example for a *node* disk fault. Feature values are standardized for better comparability.

is the set of resource entities with the ith type and κ is the number of resource entity types. For the ith resource entity type, the set of possible fault types is denoted by $\mathcal{F}_i$. The objectives are as follows:
(1) *Resource entity localization*: Find the subset of resource entities $\mathcal{E}_{sub} \subseteq \mathcal{E}$ where the fault occurs.
(2) *Fault type classification*: For each resource entity e of the ith resource entity type, identify its fault type vector $\boldsymbol{f_e} \in \{0,1\}^{|\mathcal{F}_i|}$ where $|\mathcal{F}_i|$ is the cardinality of $\mathcal{F}_i$. If the fault is classified as the jth type, the jth element of $\boldsymbol{f_e}$ is 1. Otherwise, it is 0.

4 APPROACH

Figure 4 shows the architecture of *LasRCA*. LLMs, small classifiers, and optionally involved SREs collaborate to perform one-shot RCA, with each assuming different responsibilities in the training and inference stages as detailed in Section 4.1 and Section 4.2.

4.1 Training Stage

In the training stage (Figure 4a), the entire process forms a closed loop, with responsibilities divided as follows:

- *Small Classifier:* The small classifier is the primary optimization target in the training stage and is refined through k training iterations. Initially trained on one-shot fault examples, a set of high-confusion data is selected in each iteration. The feedback labels for these data, provided by the LLM and optionally involved SREs, are then used to retrain the small classifier.

Algorithm 1 LLM-driven agglomerative fault labeling

```
Input: entity type set: $\mathcal{I}$; entity subgraph: $\mathcal{G}'$; fault type set: $\mathcal{F}'$
Output: the set of fault type vectors for all vertices in $\mathcal{G}'$: $\mathcal{V}'$
$\mathcal{E}' \leftarrow$ the vertex set based on $\mathcal{G}'$
$\mathcal{R}' \leftarrow$ the edge set based on $\mathcal{G}'$
$store_e \leftarrow HashMap()$    ▷ Store entity fault identifications
for each $e \in \mathcal{E}'$ do
    $i \leftarrow$ the entity type of $e$ in $\mathcal{I}$
    $O_e \leftarrow$ the set of observability features of $e$
    $\mathcal{F}'_i \leftarrow$ the subset of $\mathcal{F}'$ specific to the entity type $i$
    $store_f \leftarrow HashMap()$    ▷ Store fault type identifications
    for each $f \in \mathcal{F}'$ do
        ▷ Determining whether $e$ exhibits characteristics of $f$
        $store_f[f]$ = LLMFaultTypeMatching($O_e$)
    end for
    ▷ Merge fault characteristics to make entity-level decisions
    $store_e[e]$ = LLMEntityLevelAnalysis($store_f$)
end for
▷ Consider propagation and return the set of fault type vectors
return LLMGraphLevelAnalysis($store_e$, $\mathcal{R}'$)
```

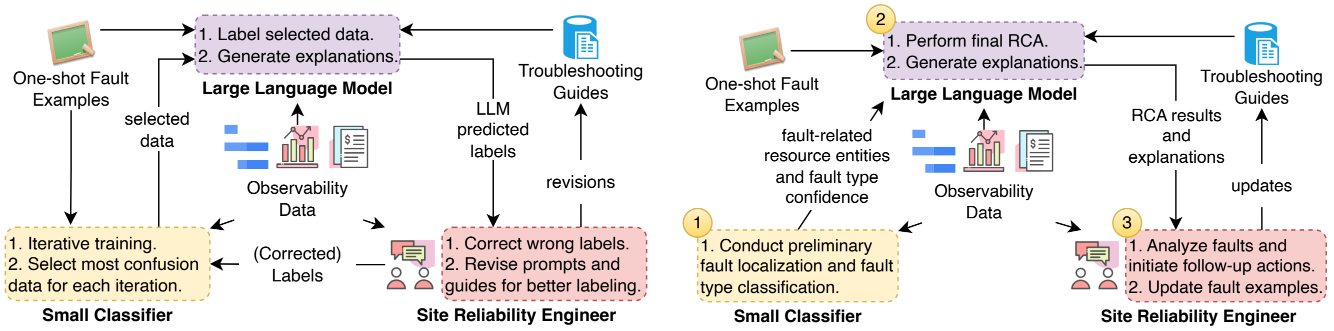
- *LLM:* Based on troubleshooting guides and one-shot fault examples, the LLM is applied to label the selected data and generate explanations to assist with the corrections if SREs are involved.
- *SRE:* Since LLMs cannot guarantee absolute inference accuracy, SREs can enhance the training stage by correcting mislabeled data, analyzing labeling errors, and optimizing prompts and troubleshooting guides for more accurate LLM-driven fault labeling.

Since this stage is inherently driven by LLMs, we will first introduce LLM-driven fault labeling in Section 4.1.1. From this, we will derive the principles and specific design of the supporting small classifier in Section 4.1.2. Then we will introduce the process of the SRE correction to address LLM misjudgments in Section 4.1.3.

4.1.1 LLM-Driven Fault Labeling.

In cloud-native systems, faults can propagate across multiple resource entities and involve various aspects. For example, a "Disk Write IO Load" fault may cause increased CPU usage. Analyzing CPU usage alone may result in misjudgments. To address this, we propose an LLM-driven agglomerative fault labeling method as shown in Algorithm 1. This method starts with basic individual fault types and progressively analyzes fault labels for resource entities within the subgraph, involving the following key functions:

- `LLMFaultTypeMatching`: This function uses an LLM to analyze fault characteristics in various aspects of resource entities (e.g., CPU, Memory, and Network). An example is shown in Figure 3. It identifies whether a resource entity exhibits any fault type.
- `LLMEntityLevelAnalysis`: This function provides a holistic analysis of the resource entity, mitigating misjudgments from faults affecting multiple aspects. Figure 5 shows the LLM input prompt template. The variable "CPU observations from LLMFaultTypeMatching" is from `LLMFaultTypeMatching`. The LLM sequentially considers the duration and significance of fault characteristics to enhance inference. This process ultimately determines the overall fault type of the entity through a reasoning chain [45].


One-shot Fault Examples
1. Label selected data.
2. Generate explanations.
Large Language Model
Troubleshooting Guides
selected data
LLM predicted labels
Observability Data
revisions
1. Iterative training.
2. Select most confusion data for each iteration.
Small Classifier
(Corrected) Labels
1. Correct wrong labels.
2. Revise prompts and guides for better labeling.
Site Reliability Engineer
One-shot Fault Examples
2
1. Perform final RCA.
2. Generate explanations.
Large Language Model
Troubleshooting Guides
fault-related resource entities and fault type confidence
RCA results and explanations
Observability Data
updates
1
1. Conduct preliminary fault localization and fault type classification.
Small Classifier
3
1. Analyze faults and initiate follow-up actions.
2. Update fault examples.
Site Reliability Engineer

(a) Training stage (b) Inference stage

Figure 4: The overall architecture of *LasRCA*.

## Observations

### CPU

{{CPU observations from LLMFaultTypeMatching}}

### ...

## Question

Address the following questions sequentially based on the observations. Let's think step by step.

1. What is the most likely duration of the fault? ...

2. Which fault characteristic is most significant? ...

3. What is the most probable fault type? Or there is 'No fault'? ...

Figure 5: The "`LLMEntityLevelAnalysis`" prompt template.

- `LLMGraphLevelAnalysis`: This function analyzes fault propagation within a subgraph to identify the actual fault. Figure 6 shows the LLM input prompt template. The variable "entity and relation information" is from the system topology, and the variable "currencyservice-0 observations from LLMEntityLevelAnalysis" is from `LLMEntityLevelAnalysis`. Subgraphs are selected since the full system topology is too large, resulting in excessive input. This can reduce the quality of LLM inference and increase time and financial costs. We limit subgraphs to n_g entities (default 10), which may truncate some neighbor information. Hence, we allow the LLM to predict the outcome for an entity as "unknown" if the information is insufficient. Finally, each entity in the subgraph is labeled as a specific fault type, "no fault", or "unknown".

Based on fault labels from Algorithm 1, the corresponding fault type vectors are generated. For each entity, if the fault label is "unknown", its fault type vector is not generated this time. Otherwise, its fault type vector is generated as described in Section 3.

4.1.2 Principles and Design of the Small Classifier.

According to the fault labeling method in Section 4.1.1, the supporting small classifier needs to possess the following capabilities:

- Modeling the mapping between observability features and fault type vectors for heterogeneous resource entities.
- Fast training speed to accommodate multiple iterations.
- Capturing high-confusion resource entity samples and their associated subgraphs for LLM-driven fault labeling and retraining.

## Observations

### currencyservice-0

{{currencyservice-0 observations from LLMEntityLevelAnalysis}}

### ...

## Entities and Relations

{{entity and relation information}}

## Question

Address the following questions sequentially based on the observations. Let's think step by step.

1. What is the exact type of each entity? ...

2. Explain the reasons for making judgments. ...

Figure 6: The "`LLMGraphLevelAnalysis`" prompt template.

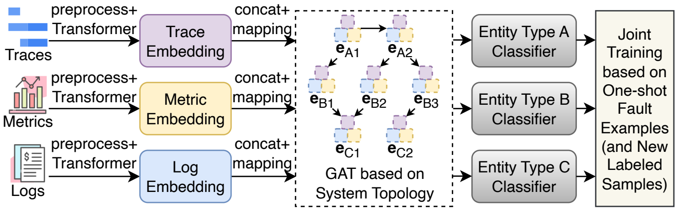


Figure 7: The architecture of the small classifier.

To meet these capabilities, we design a small classifier as shown in Figure 7. It is composed of the following components:

- *Preprocess and Embed:* Raw observability data is preprocessed to compressed observability features to reduce noise and redundancy. Since traces, metrics, and logs each have different data structures, and the time series format of metrics is most concise with intuitive characteristics, both traces and logs are converted to the time series format. This also eliminates the subsequent significant computational time cost associated with vectorizing their original text formats. With the granularity Δt, we partition the analysis interval $[t_s, t_e)$ into buckets $[t_s, t_s + \Delta t)$, $[t_s + \Delta t, t_s + 2\Delta t), \ldots, [t_s + (t-1) \times \Delta t, t_e - (t-1) \times \Delta t - t_s)$ to align the data from a temporal perspective where $t = \lfloor (t_e - t_s)/\Delta t \rfloor + 1$. For traces, we analyze each *service* by calculating the average duration and average number of invocations for the following:

invocations executed on the *service*, parent invocations of the invocations executed on the *service*, and child invocations of the invocations executed on the *service*. This preserves as much structural information from traces as possible. For logs, we first apply Drain [13] to extract lightweight constant context in logs (a.k.a. log event). Then we use the TF-IDF score [37] to select important words and log events, counting their occurrences in each bucket as the log features. All of trace features, log features, and metric features are standardized and are finally denoted by $\boldsymbol{X}^{\mathcal{T}} \in \mathbb{R}^{t \times n_t}$, $\boldsymbol{X}^{\mathcal{L}} \in \mathbb{R}^{t \times n_l}$, $\boldsymbol{X}^{\mathcal{M}} \in \mathbb{R}^{t \times n_m}$ respectively, where n_t, n_l, and n_m are the number of respective feature types.

After preprocessing, the feature matrices need to be embedded into a high-dimensional vector space to learn high-order representations. Due to the varying number of feature types among different heterogeneous entities (e.g., a *node* may have 24 metric feature types, whereas a *service* may only have 8), simple equal-length segmentation is not feasible. Therefore, the generated embeddings must preserve the original feature type dimension for detailed segmentation. Additionally, significant differences in feature characteristics among traces, metrics, and logs (such as "transmit_MB" in metrics and "duration" in traces as shown in Figure 2) necessitate independent modeling modules for each data type. Considering these factors, we apply the Transformer encoder [43] to separately model the features of each data type due to its efficiency and superior performance in sequence modeling. The fixed positional embedding [43] is initially added to each feature matrix to incorporate relative positional information. Then each position-encoded $\boldsymbol{X} \in \mathbb{R}^{t \times n}$ is transposed and mapped into the vector space $\mathbb{R}^{n \times \lambda}$ using a Linear layer to preserve the original feature type dimension. Finally, `TransformerEncoder` is applied to model the features of each data type. Equation 1 shows the embedding process based on $\boldsymbol{X}$. Each data type has independent `TransformerEncoder` and `Linear`.

$$\boldsymbol{H} = \texttt{TransformerEncoder}(\texttt{Linear}(\boldsymbol{X}^{\top})) \tag{1}$$

- *Concat and Map:* To obtain the vector representation of the resource entity for resource entity localization and fault type classification, observability feature embeddings of the resource entity are concatenated. For each resource entity type, we establish a fixed order for concatenating all relevant feature types. Accordingly, for an entity e of the ith type, we select the corresponding rows from the embedding matrix $\boldsymbol{H}$. These rows are concatenated into a new matrix $\boldsymbol{C}_e \in \mathbb{R}^{n_i \times \lambda}$, where n_i denotes the number of all feature types of e across traces, metrics, and logs. To facilitate rapid fusion of feature embeddings in $\boldsymbol{C}_e$ for a compact representation, `TransformerEncoder` is applied as shown in Equation 2.

$$\boldsymbol{C}'_e = \texttt{TransformerEncoder}(\boldsymbol{C}_e) \tag{2}$$

$\boldsymbol{C}'_e$ is then mapped into the vector representation $\boldsymbol{v}_e \in \mathbb{R}^{\mu}$ through `Linear` and `Flatten`. μ is the hyperparameter of the entity representation dimension and `Flatten` reshapes the matrix into a one-dimensional vector. Each resource entity type has its independent `TransformerEncoder` and `Linear`.

- *Model Fault Propagation and Train Classifiers:* Based on the entity vector representations and the system topology, we apply Graph Attention Network (GAT) [4] to model fault propagation due to its efficiency and superior performance in graph modeling. Each vector representation is updated by the weighted average of its neighbor representations. For each entity e with the ith type and the updated representation $\boldsymbol{v}'_e \in \mathbb{R}^{\mu}$, its probability vector $\hat{\boldsymbol{f}}_e \in (0,1)^{|\mathcal{F}_i|}$ for possible fault types $\mathcal{F}_i$ is computed using the activation function `Sigmoid` and `Linear` as shown in Equation 3. Since each resource entity type may have diverse possible fault types, a separate `Linear` is used for each entity type.

$$\hat{\boldsymbol{f}}_e = \texttt{Sigmoid}(\texttt{Linear}(\boldsymbol{v}'_e)) \tag{3}$$

We use Binary Cross Entropy (BCE) [10], widely used in classification problems, to measure the difference between the predicted vectors and true fault type vectors. BCE optimizes probability predictions for each fault type, making it suitable for providing the LLM with confidence levels of specific fault types during inference. For each resource entity type, BCE is calculated on the fault types of entities in one-shot fault examples and LLM-labeled samples. The sum of BCEs for all resource entity types is then optimized as the overall loss function. In this manner, the small classifier models the mapping between observability features and fault type vectors for heterogeneous resource entities. Additionally, it utilizes efficient model architectures to ensure rapid training speeds, thereby supporting multiple iterations.

- *Select High-Confusion Samples:* After training the small classifier in each iteration, it is essential to select samples exhibiting high confusion in the classifier, label these samples using the LLM, and continue with iterative training. We select the most intuitive entropy as the measure of confusion. For the ith fault type in every unlabeled entity e, the quantitative calculation of its confusion con_{ei} is as Equation 4. p_{ei} is the probability of the ith fault type occurring in entity e as predicted by the small classifier.

$$con_{ei} = -p_{ei} \log p_{ei} - (1 - p_{ei}) \log(1 - p_{ei}) \tag{4}$$

The confusion level of an entity is the confusion level of its most confused fault type. We select the top α entities with the highest confusion levels. The β-order neighbors of these α entities from the system topology form subgraphs. Each subgraph includes up to n_g entities, which are provided for LLM fault labeling.

4.1.3 SRE Corrections.

LLMs may generate nonsensical or unfaithful text to the provided source input (a.k.a. hallucination) [14]. Figure 8 shows an example of LLM misjudgments due to hallucination. According to RULE 1, a fault in a single *pod* does not cause the entire *service* to fail, but the LLM incorrectly classified the entire *service* as faulty. To address this, SREs can correct wrong labels from LLM misjudgments during training. Currently, SREs mainly review two aspects, which have clear objectives and reduce the workload of analyzing from scratch:

- Do LLM-generated responses conflict with troubleshooting guides?
- Are the phenomena in LLM-generated responses accurate?

When SREs identify incorrect fault labels in LLM-generated responses, they replace them with one of the following: a specific fault type, "no fault", or "unknown". For example, in Figure 8, the label of "shipping" is corrected to "no fault". These corrected fault labels, along with the accurate ones, are used as new labels. The

```
## Observations
- shipping-0: … Given the nature of metrics and the lack of
correlation to known fault types, it is most accurate to classify
this as 'No Fault' …
- shipping-1: … it is most likely that there are 'No fault' …
- shipping-2: … the most probable assertion … is 'No fault'.
- shipping2-0: … The most probable fault type, based on the
significant rise in disk read operations without comparable
increases in disk write metrics, is 'K8s Disk Read IO Load' …
- …

## Entities and Relations
### Entities
service: shipping, frontend, checkout
pod: shipping-0, shipping-1, shipping-2, shipping2-0
### Relations
shipping - shipping-0; shipping - shipping-1; shipping - shipping-2;
shipping - shipping2-0; …

## Question
RULES:
- RULE 1: Only 1 pod displays fault characteristics. In this
condition, the service MUST be classified as 'No Fault'.
- RULE …: …
Address the following questions sequentially based on the
observations. Let's think step by step.
1. What is the exact type of each entity? The candidates are …,
K8s Disk Read IO Load', …. You MUST only select one candidate.
2. Explain the reasons for making judgments.
------------------------------------------------------------------
** Response from LLM (GPT-4) **
1. Entity classifications:
shipping: K8s Disk Read IO Load; shipping-0: No fault; shipping-1:
No fault; shipping-2: No fault; shipping2-0: K8s Disk Read IO Load
2. Reasons for judgments: … since 1 of 4 pods (shipping2-0) displays
K8s Disk Read IO Load fault characteristics and no other faults
mirror across other pods, the service inherits this fault type. …
```

Figure 8: An example of LLM misjudgments.

small classifier is then retrained on the entirety of valid fault labels (i.e., the labels with the value of a specific fault type or "no fault").

4.2 Inference Stage

The inference stage (Figure 4b) sequentially involves the small classifier and the LLM, ultimately providing RCA results and explanations to SREs. The responsibilities are divided as follows:

- *Small Classifier:* It first predicts fault type probabilities for each resource entity. If a probability exceeds the threshold θ, the fault type is considered present; otherwise, it is absent. Entities are ranked by their highest fault type probability for resource entity localization. The small classifier then provides fault types with probabilities exceeding θ, along with their predicted probabilities (a.k.a. fault type confidence) and original characteristics, to the LLM for further analysis and explanations. For more comprehensive information, `LLMEntityLevelAnalysis` and `LLMGraphLevelAnalysis` can also be applied.
- *LLM:* To better utilize the results of the small classifier, we refer to SuperICL [46], incorporating the fault type confidence into the input for the LLM. The LLM provides detailed judgments based on the troubleshooting guide, one-shot fault examples, and fault type confidence. This combined analysis offers further explanations to support SRE decision-making. An example of the LLM input and judgments is shown in Figure 10.
- *SRE:* SREs can analyze faults using explanations provided by the LLM. Additionally, when a confirmed fault exhibits significant characteristics, SREs may consider updating the fault examples to enhance the capabilities of the LLM and the small classifier. Currently, in the inference stage, SREs only receive fault feedback, which does not impact the performance of *LasRCA*.

5 EVALUATION

5.1 Research Questions

- *RQ1:* How effective is *LasRCA* in performing one-shot RCA?
- *RQ2:* How cost-effective is *LasRCA* compared to manual analysis and solely LLM-based analysis?
- *RQ3:* How beneficial are the analysis results of *LasRCA* for SREs?
- *RQ4:* What is the main component's contribution in *LasRCA*?

5.2 Experiment Setup

5.2.1 Study Data. We evaluate *LasRCA* using public datasets $\mathcal{A}$, $\mathcal{B}$, and C, which exhibit the heterogeneous nature and prevalent fault types. These datasets are from a large-scale AIOps challenge collected from an open-source cloud-native application Hipstershop [11] widely used in AIOps research. Hipstershop comprises 10 microservices written in Java, Python, Go, Node.js, and C#. The entire dataset comprises 7 days of training data with 300 fault injections and 5 days of test data with 241 fault injections. $\mathcal{A}$ contains observability data for all 10 *services* with 9 fault types. Each *service* in $\mathcal{A}$ corresponds 4 *pods* which handle incoming requests from the *service*. $\mathcal{B}$ contains observability data for the total 40 *pods* with 9 fault types. Each *pod* is deployed in one of the six *nodes*, and C contains observability data for all these *nodes* with 6 fault types. For $\mathcal{A}$, $\mathcal{B}$, and C, only one fault example per fault type is selected from the training data during the training stage.

5.2.2 Baselines and Implementation. We compare *LasRCA* with the following state-of-the-art (SOTA) and widely adopted baselines in RCA, encompassing both supervised and unsupervised approaches.

DiagFusion[54]. It is the SOTA multimodal supervised approach for localizing *pods* as faults as well as classifying fault types.

DejaVu [22]. It is the SOTA metric-based supervised approach for localizing resource entities and suspect metric sets.

CIRCA [20]. It is another SOTA metric-based unsupervised approach for localizing resource entities and suspect metric sets.

TraceRCA [21]. It is a widely used trace-based unsupervised approach for resource entity localization in the *service* level.

LightGBM [15]. It is one of the most effective classifiers, and we employ it for supervised fault type classification.

For implementation details, we use GPT-4 [33] as the LLM due to its superior logical reasoning capability. The small classifier is implemented using PyTorch [36] and PyTorch-Geometric [9]. The experiments are conducted on a Linux server with the Intel(R) Xeon(R) Gold 6226R CPU and the NVIDIA GeForce RTX 4090 GPU via Python 3.10. We set the number of iterations $k = 1$ to specifically observe the performance of *LasRCA* under lower cost conditions. The hyperparameters of selecting high-confusion samples are set as follows: $\alpha = 100$, $\beta = 2$, and $n_g = 10$. Then we apply Mixup [53] to augment training data through linear interpolations of input data pairs and their corresponding labels, thereby improving generalization and reducing overfitting. Suppose $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ are feature matrices of a resource entity in two fault samples, with corresponding fault labels $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$. The augmented sample $(\boldsymbol{X}, \boldsymbol{y})$ by Mixup is shown as Equation 5, where η is a random value from the Beta distribution $\mathbf{B}(0.2, 0.2)$ (0.2 is a widely used hyperparameter value).

$$\boldsymbol{X} = \eta \boldsymbol{X}_1 + (1-\eta)\boldsymbol{X}_2,\ \boldsymbol{y} = \eta \boldsymbol{y}_1 + (1-\eta)\boldsymbol{y}_2 \tag{5}$$

For hyperparameter settings of the small classifier and baselines, we use *a fixed set of empirical hyperparameters* due to the lack of a validation set for tuning in one-shot RCA. For example, the probability threshold θ is set to 0.5. We set the initial learning rate to 0.0001 and apply the cosine annealing schedule [25] to gradually reduce the learning rate. Training is stopped after 500 epochs to reduce underfitting since the training loss stabilizes across various experiments. Details can be found in our publicly available code.

5.2.3 Evaluation metrics. We evaluate *LasRCA* in two critical one-shot RCA tasks: resource entity localization (RCL) and fault type classification (FTC). For RCL, we employ the top-u accuracy (AC@u), a generalizable metric in localization tasks [20–22, 54]. AC@u measures the probability of the ground truth resource entity being included in the top-u results. Formally, given a test set $\mathcal{S}$ of fault samples, for each fault sample $a \in \mathcal{S}$, the resource entity with the ground truth fault is denoted by e_a. For the top-u resource entity set $\mathcal{E}_a[u]$ generated by an RCA approach, AC@u is defined as:

$$\mathrm{AC@}u = \frac{1}{|\mathcal{S}|} \sum_{a \in \mathcal{S}} \begin{cases} 1, \text{if } e_a \in \mathcal{E}_a[u] \\ 0, \text{otherwise} \end{cases} \tag{6}$$

AC@u evaluates the model performance by considering not just the top prediction but a set of likely candidates. We set $u = 1, 3, 5$ to help SREs identify the faulty entity within a short list. For FTC, a standard classification task, mature evaluation metrics already exist [10]. We apply standard evaluation metrics considering both the individual fault type and the overall performance: Micro Precision (MiPr), Macro Precision (MaPr), Micro Recall (MiRe), Macro Recall (MaRe), Micro F1-score (MiF1), and Macro F1-score (MaF1), c [54]. The calculation details are as follows:

- **True Positives (TP)**: The number of correct predictions where the model correctly identifies a fault type.
- **False Positives (FP)**: The number of incorrect predictions where the model incorrectly identifies a fault type that is not present.
- **False Negatives (FN)**: The number of incorrect predictions where the model fails to identify a fault type that is present.
- **MiPr and MaPr**: MiPr is calculated by considering the total TP and FP across all w fault types, while MaPr averages the metrics calculated for each fault type regardless of their size.

$$\mathrm{MiPr} = \frac{\sum_{i=1}^{w} \mathrm{TP}_i}{\sum_{i=1}^{w} (\mathrm{TP}_i + \mathrm{FP}_i)},\ \mathrm{MaPr} = \frac{1}{w} \sum_{i=1}^{w} \frac{\mathrm{TP}_i}{\mathrm{TP}_i + \mathrm{FP}_i} \tag{7}$$

- **MiRe and MaRe**: MiRe is calculated by considering the total TP and FN across w all fault types, while MaRe averages the metrics calculated for each fault type regardless of their size.

$$\mathrm{MiRe} = \frac{\sum_{i=1}^{w} \mathrm{TP}_i}{\sum_{i=1}^{w} (\mathrm{TP}_i + \mathrm{FN}_i)},\ \mathrm{MaRe} = \frac{1}{w} \sum_{i=1}^{w} \frac{\mathrm{TP}_i}{\mathrm{TP}_i + \mathrm{FN}_i} \tag{8}$$

- **MiF1 and MaF1**: MiF1 is the harmonic mean of MiPr and MiRe, and MaF1 is the harmonic mean of MaPr and MaRe:

$$\mathrm{MiF1} = 2 \cdot \frac{\mathrm{MiPr} \cdot \mathrm{MiRe}}{\mathrm{MiPr} + \mathrm{MiRe}},\ \mathrm{MaF1} = 2 \cdot \frac{\mathrm{MaPr} \cdot \mathrm{MaRe}}{\mathrm{MaPr} + \mathrm{MaRe}} \tag{9}$$

All quantified results are from small classifiers retrained on LLM feedback (and SRE corrections explicitly noted) for consistency.

5.3 RQ1: Effectiveness in One-Shot RCA

Table 2 shows the one-shot RCA results. To ensure a fair comparison, *LasRCA* in RQ1 excludes the SRE involvement, using small classifiers retrained on LLM feedback for more intuitive comparisons. In the context of limited samples, *LasRCA* shows significant advantages over baselines. This superiority is primarily due to the additional supervision from LLM-driven fault labeling and the small model design that effectively addresses heterogeneity. In one-shot RCA, unsupervised CIRCA and TraceRCA cannot classify fault types as they do not use fault type information. Their applicability is also limited by inherent assumptions, leading to degraded performance or incompatibility in scenarios like dataset $\mathcal{C}$. Supervised DiagFusion, DejaVu, and LightGBM exhibit performance limitations due to insufficient labeled data. Moreover, the baseline performance of fault type classification on $\mathcal{A}$ is generally poor. This is mainly because *service*-level features are user-perceived metrics, with details depending on associated *pods*. Effective fault type classification requires modeling these heterogeneous entity associations. The results highlight the need for more fault labels and modeling of heterogeneous entity associations. *LasRCA* addresses these issues with LLM fault labeling and the small classifier design, reducing SREs' analysis burden and proving effective in one-shot RCA.

5.4 RQ2: Effectiveness in Cost Reduction

We first compare the performance of *LasRCA* to manual labeling. As detailed in Section 5.2.2, we use 100 LLM-labeled subgraphs containing 653 resource entities for retraining the small classifier in *LasRCA*. With no prior knowledge, a single fault analysis involves a complete graph involving 56 entities, so the LLM-labeled data equates to roughly 12 analyses. We randomly select 12, 24, 36, and 48 additional complete analyses (1×, 2×, 3×, and 4× the LLM-labeled amount), provide their ground truth labels, and retrain the small classifier to evaluate the performance of manual labeling. Moreover, we evaluate the small classifier retrained with SRE-corrected LLM labels. We invite three SREs with over two years of experience to correct fault labels according to the process in Section 4.1.3. We use MicroF1 and MacroF1 as the example evaluation metrics shown in Figure 9. The x-axis value of 0 represents the small classifier trained only on one-shot fault examples. The effectiveness of LLM-driven fault labeling is roughly equivalent to 3 to 4 times the number of manual labels without prior knowledge, thanks to the efficient strategy for selecting labeling samples. Additionally, we observe that LLM-driven fault labeling still lags behind SRE-corrected labels, primarily due to LLM hallucinations and its capability limitations. Nonetheless, the automated nature of *LasRCA* shows high usability.

We also evaluate the effectiveness of *LasRCA* in reducing costs. By assessing the number of involved entities, the reduction in input can be evaluated without extra algorithms. For time costs, considering the intrinsic resource constraints limiting LLM concurrency,

Table 2: Results of one-shot RCA. Each result of supervised approaches is the mean of 5 runs.

Data	Approach	Resource Entity Localization			Fault Type Classification					
		AC@1	AC@3	AC@5	MiPr	MaPr	MiRe	MaRe	MiF1	MaF1
$\mathcal{A}$	LasRCA (Small Classifier + LLM)	**79.54%**	**89.77%**	**94.54%**	**0.5585**	**0.5658**	**0.5045**	**0.5481**	**0.5244**	**0.5320**
	DiagFusion	26.81%	53.40%	67.72%	0.0440	0.0932	0.0272	0.0370	0.0336	0.0431
	DejaVu	55.22%	64.77%	71.13%	-	-	-	-	-	-
	CIRCA	56.81%	69.31%	72.72%	-	-	-	-	-	-
	TraceRCA	14.77%	32.95%	43.18%	-	-	-	-	-	-
	LightGBM	-	-	-	0.1971	0.1398	0.1045	0.0975	0.1363	0.1025
$\mathcal{B}$	LasRCA (Small Classifier + LLM)	**61.62%**	**75.81%**	**81.86%**	0.4047	0.5123	**0.6552**	**0.6786**	**0.4973**	**0.5694**
	DiagFusion	8.6%	16.04%	26.04%	-	-	-	-	-	-
	DejaVu	48.37%	55.11%	60.00%	-	-	-	-	-	-
	CIRCA	43.02%	61.62%	63.95%	-	-	-	-	-	-
	TraceRCA	5.81%	9.30%	10.46%	-	-	-	-	-	-
	LightGBM	-	-	-	**0.6927**	**0.6840**	0.3716	0.3813	0.4837	0.4646
$\mathcal{C}$	LasRCA (Small Classifier + LLM)	**61.49%**	**69.25%**	**72.53%**	**0.6586**	**0.7346**	**0.4328**	**0.4134**	**0.5196**	**0.4851**
	DejaVu	54.92%	60.59%	71.04%	-	-	-	-	-	-
	CIRCA	16.41%	19.40%	25.37%	-	-	-	-	-	-
	LightGBM	-	-	-	0.4861	0.3471	0.2626	0.2194	0.3410	0.2653

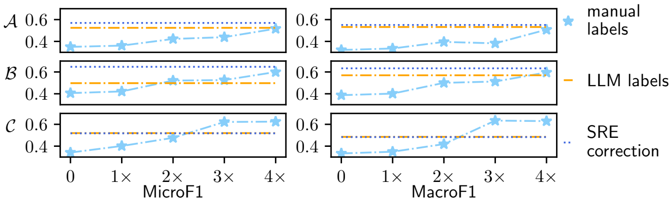


Figure 9: LLM-driven fault labeling vs. manual labeling.

Table 3: Cost analysis of the involved entities and time.

Approach	Involved Entities		Approximate Average Time (s)		
	Train	Test	Train	Each Test	Total Test
LasRCA	653	677	954.19	15.88	3827.08
All by LLM	-	13496	-	42.87	10331.67

we use approximate average time, ignoring concurrency limits for concurrently executable steps. As shown in Table 3, for the total test set, *LasRCA* requires about $\frac{1}{10}$ of resource entities needed for full LLM dependence, with most analysis performed by the LLM-guided small classifier. Training *LasRCA* takes approximately 16 minutes but accelerates each inference by about half a minute. The more inferences performed, the more significant the reduction.

5.5 RQ3: Usability for SREs

The LLM's ability to generate insightful outputs enables *LasRCA* to produce clear explanations. With a designed question chain, *LasRCA* provides feedback on fault characteristics, duration, and potential fault types. Figure 10 shows an example of "Node CPU Failure" fault explanations, exhibiting the following characteristics:

- *LasRCA* can infer the root cause. The explanations can accurately identify the ground truth fault type "Node CPU Failure": "It is reasonable to suspect a **CPU Failure** fault in this instance".
- *LasRCA* provides the detailed fault description rather than just prediction probabilities. The description is "The pattern where the CPU usage suddenly spikes and remains at a high level without a continued increase after stabilizing is more suggestive of CPU failure rather than CPU climb". It clearly describes fault symptoms such as "system.cpu.pct_usage" (a sharp increase from 2.177 to 8.209, and stabilization from 8.209 to 7.427). It provides the basis for judgments, forming a complete fault analysis chain.
- The confidence level from the small classifier can assist the LLM in making judgments ("The model prediction also affirms that it's likely a CPU Failure with 62% confidence.").

The example shows that *LasRCA* provides understandable explanations, alleviating the burden of manual analysis by SRE.

5.6 RQ4: Ablation Study

To evaluate the contributions of the main components in *LasRCA*, we create the following variants, reporting their results in Table 4:

- SRE Correction Ablation (**V1**): Without SRE-corrected fault labels (see Section 5.4), the performance drops by an average of 0.0332 in F1-scores, underscoring the effectiveness of SRE corrections.
- LLM Ablation (**V2**): The LLM notably enhances the small classifier, improving F1-scores by an average of 0.1647. The improvement stems from LLM's rapid expansion of high-confusion labels.
- Transformer Ablation in the Small Classifier (**V3**): Removing Transformer results in a drastic decrease in F1-scores on datasets $\mathcal{A}$ and $\mathcal{B}$, reflecting the necessity of Transformer.
- GAT Ablation in the Small Classifier (**V4**): Removing GAT leads to a significant drop in evaluation metrics on dataset $\mathcal{A}$, similarly demonstrating the essential role of Graph Attention Network.

Table 4: The contributions of the main components in LasRCA. Each result is the mean of 5 runs.

Data	Variants of LasRCA	MiPr	MaPr	MiRe	MaRe	MiF1	MaF1
$\mathcal{A}$	Small Classifier + LLM + SRE Correction	0.6905	**0.6252**	0.4833	0.5397	**0.5661**	**0.5456**
	Small Classifier + LLM (**V1**, also applied in RQ1)	0.5585	0.5658	**0.5045**	**0.5481**	0.5244	0.5320
	Small Classifier (**V2**)	**0.7511**	0.6075	0.2295	0.2465	0.3499	0.3214
	Small Classifier without Transformer (**V3**)	0.0566	0.0800	0.3681	0.3548	0.0979	0.1114
	Small Classifier without GAT (**V4**)	0.0747	0.1232	0.1068	0.1080	0.0872	0.0928
$\mathcal{B}$	Small Classifier + LLM + SRE Correction	0.6259	0.6317	0.5832	0.6187	**0.6017**	**0.6086**
	Small Classifier + LLM (**V1**, also applied in RQ1)	0.4047	0.5123	**0.6552**	**0.6786**	0.4973	0.5694
	Small Classifier (**V2**)	**0.7928**	**0.7604**	0.2735	0.2976	0.4062	0.3873
	Small Classifier without Transformer (**V3**)	0.1224	0.1595	0.4589	0.4601	0.1911	0.2195
	Small Classifier without GAT (**V4**)	0.7648	0.7476	0.3013	0.3241	0.4318	0.4271
$\mathcal{C}$	Small Classifier + LLM + SRE Correction	**0.6627**	**0.7362**	0.4318	0.4116	**0.5202**	0.4850
	Small Classifier + LLM (**V1**, also applied in RQ1)	0.6586	0.7346	0.4328	0.4134	0.5196	**0.4851**
	Small Classifier (**V2**)	0.5460	0.6842	0.2537	0.2703	0.3412	0.3332
	Small Classifier without Transformer (**V3**)	0.4128	0.3746	**0.5313**	**0.5270**	0.4638	0.3905
	Small Classifier without GAT (**V4**)	0.6171	0.7144	0.2268	0.2377	0.3309	0.3226

In summary, the above ablation experiment results demonstrate the necessity of each main component within *LasRCA*.

6 DISCUSSION

6.1 Limitations and Future Explorations

Since *LasRCA* is an LLM-driven approach, its effectiveness is highly dependent on the LLM capabilities. Currently, we choose GPT-4 for its superior reasoning capabilities. Within our NVIDIA GeForce RTX 4090 GPU, we test some small-scale LLMs such as Mistral-7B-Instruct-v0.3 (Mistral-7B) and Gemma-7B-Instruct. They exhibit noticeable performance degradation and even errors in numerical comparisons. Figure 11 shows judgment differences between GPT-4 and Mistral-7B on a pod without the "CPU Load" fault. Mistral-7B exhibits hallucinations, failing to detect the upward trend in numbers and leading to incorrect identification of the "CPU load" fault. GPT-4 demonstrates accurate perception and makes correct judgments. Consequently, *LasRCA* still requires large-scale LLMs with substantial parameter counts, leading to computational constraints.

In future work, we plan to use the fault labels and explanations from GPT-4 to fine-tune small-scale LLMs. This will expand the applicability of *LasRCA* and further reduce the cost of LLMs.

6.2 Threats to Validity

The main threat to *internal* validity arises from the implementation of *LasRCA* and baselines. To mitigate this, *LasRCA* is implemented using mature frameworks (Section 5.2.2). Baselines are implemented using the publicly available codes and the meticulous reproduction of the methodology in original papers. To minimize the impact of randomness, we conduct multiple runs for supervised approaches.

The main threat to *external* validity arises from study objects. To mitigate this, *LasRCA* is evaluated on three representative resource entity types with common faults. Since homogeneity is the subset of heterogeneity, the heterogeneity of these resource entities shows that *LasRCA* can also be extended to other homogeneous systems.

The main threat to *construct* validity arises from evaluation metrics. To mitigate this, we utilize widely accepted evaluation metrics in two key RCA tasks and verify the usability through case analysis.

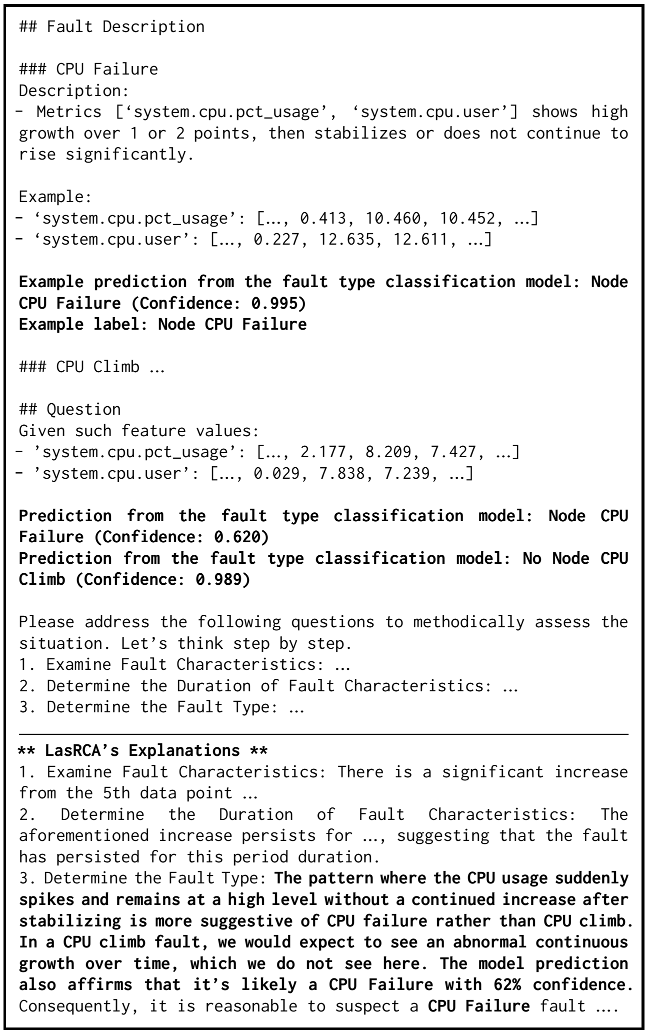
The main threat to *conclusion* validity arises from potential overfitting and underfitting due to the lack of a validation set in one-shot RCA. To mitigate overfitting, we use Mixup [53] to augment training data (Section 5.2.2). To mitigate underfitting, we first use *a fixed set of empirical hyperparameters* for all experiments, similar to the strategy in TS2Vec [51]. Then we set the initial learning rate to 0.0001 and apply the cosine annealing schedule. Observing that training loss stabilizes after 500 epochs across various experiments, we set 500 training epochs to reduce underfitting (Section 5.2.2).

7 RELATED WORK

Many approaches have been devoted to automating root cause analysis based on traces, metrics, logs, or their combinations [39]. These approaches can be classified by the adopted fault labels.

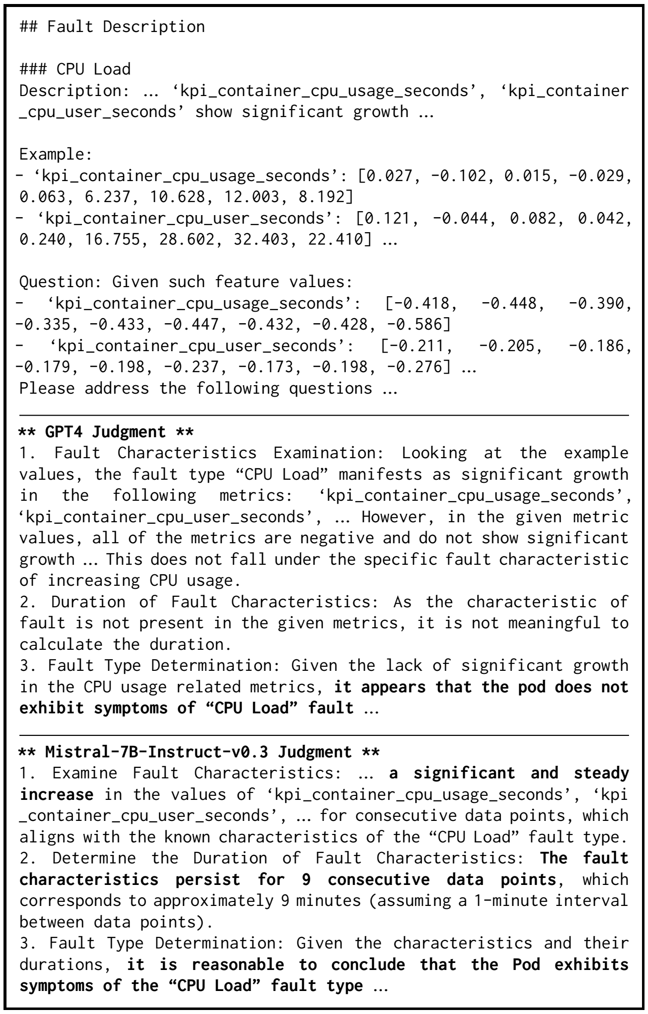
Unsupervised approaches rely on unsupervised assumptions rather than fault labels. A common assumption is that causal modeling reflects true causal relationships in fault propagation. Many studies [6, 8, 23, 24, 27–30, 35, 44] use causality discovery techniques to model causal graphs, potentially revealing fault propagation patterns. CIRCA [20] further defines intervention recognition for causality and identifies suspect metrics by observing changes in conditioned probability distributions. Another assumption comes from association rule mining, mainly applied to traces. It assumes that a microservice with frequent abnormal traces and fewer normal traces is likely the root cause [21, 31, 48, 49]. While unsupervised approaches do not require labels, they lack the depth for detailed fault analysis, such as fault type classification. Their applicability is also limited by their inherent assumptions as shown in experiments.

Supervised approaches heavily rely on fault labels. Many studies [1, 3, 42, 50, 52, 55] use machine learning models to map observability data to fault labels. Recently, graph neural network-based approaches have emerged, mainly for homogeneous systems [19, 54].



```
## Fault Description

### CPU Failure
Description:
- Metrics [‘system.cpu.pct_usage’, ‘system.cpu.user’] shows high
growth over 1 or 2 points, then stabilizes or does not continue to
rise significantly.

Example:
- ‘system.cpu.pct_usage’: [..., 0.413, 10.460, 10.452, ...]
- ‘system.cpu.user’: [..., 0.227, 12.635, 12.611, ...]

Example prediction from the fault type classification model: Node
CPU Failure (Confidence: 0.995)
Example label: Node CPU Failure

### CPU Climb ...

## Question
Given such feature values:
- ’system.cpu.pct_usage’: [..., 2.177, 8.209, 7.427, ...]
- ’system.cpu.user’: [..., 0.029, 7.838, 7.239, ...]

Prediction from the fault type classification model: Node CPU
Failure (Confidence: 0.620)
Prediction from the fault type classification model: No Node CPU
Climb (Confidence: 0.989)

Please address the following questions to methodically assess the
situation. Let’s think step by step.
1. Examine Fault Characteristics: ...
2. Determine the Duration of Fault Characteristics: ...
3. Determine the Fault Type: ...
------------------------------------------------------------------
** LasRCA’s Explanations **
1. Examine Fault Characteristics: There is a significant increase
from the 5th data point ...
2. Determine the Duration of Fault Characteristics: The
aforementioned increase persists for ..., suggesting that the fault
has persisted for this period duration.
3. Determine the Fault Type: The pattern where the CPU usage suddenly
spikes and remains at a high level without a continued increase after
stabilizing is more suggestive of CPU failure rather than CPU climb.
In a CPU climb fault, we would expect to see an abnormal continuous
growth over time, which we do not see here. The model prediction
also affirms that it’s likely a CPU Failure with 62% confidence.
Consequently, it is reasonable to suspect a CPU Failure fault ....
```

Figure 10: "Node CPU Failure" fault analysis using *LasRCA*.



```
## Fault Description

### CPU Load
Description: ... ‘kpi_container_cpu_usage_seconds’, ‘kpi_container
_cpu_user_seconds’ show significant growth ...

Example:
- ‘kpi_container_cpu_usage_seconds’: [0.027, -0.102, 0.015, -0.029,
0.063, 6.237, 10.628, 12.003, 8.192]
- ‘kpi_container_cpu_user_seconds’: [0.121, -0.044, 0.082, 0.042,
0.240, 16.755, 28.602, 32.403, 22.410] ...

Question: Given such feature values:
-  ‘kpi_container_cpu_usage_seconds’:  [-0.418,  -0.448,  -0.390,
-0.335, -0.433, -0.447, -0.432, -0.428, -0.586]
-  ‘kpi_container_cpu_user_seconds’:  [-0.211,  -0.205,  -0.186,
-0.179, -0.198, -0.237, -0.173, -0.198, -0.276] ...
Please address the following questions ...
------------------------------------------------------------------
** GPT4 Judgment **
1. Fault Characteristics Examination: Looking at the example
values, the fault type “CPU Load” manifests as significant growth
in the following metrics: ‘kpi_container_cpu_usage_seconds’,
‘kpi_container_cpu_user_seconds’, ... However, in the given metric
values, all of the metrics are negative and do not show significant
growth ... This does not fall under the specific fault characteristic
of increasing CPU usage.
2. Duration of Fault Characteristics: As the characteristic of
fault is not present in the given metrics, it is not meaningful to
calculate the duration.
3. Fault Type Determination: Given the lack of significant growth
in the CPU usage related metrics, it appears that the pod does not
exhibit symptoms of “CPU Load” fault ...
------------------------------------------------------------------
** Mistral-7B-Instruct-v0.3 Judgment **
1. Examine Fault Characteristics: ... a significant and steady
increase in the values of ‘kpi_container_cpu_usage_seconds’, ‘kpi
_container_cpu_user_seconds’, ... for consecutive data points, which
aligns with the known characteristics of the “CPU Load” fault type.
2. Determine the Duration of Fault Characteristics: The fault
characteristics persist for 9 consecutive data points, which
corresponds to approximately 9 minutes (assuming a 1-minute interval
between data points).
3. Fault Type Determination: Given the characteristics and their
durations, it is reasonable to conclude that the Pod exhibits
symptoms of the “CPU Load” fault type ...
```

Figure 11: Judgment differences between GPT-4 and Mistral-7B-Instruct-v0.3 on a pod without the "CPU Load" fault.

DejaVu [22] builds failure dependency graphs linking each metric to its resource entity, using graph neural networks to identify suspect metrics and entities. While these approaches offer detailed fault analysis, they require sufficient fault labels. This is challenging in cloud-native systems where such labels are scarce.

With the success of LLMs [47], some LLM-driven approaches, such as RCACopilot [7], focus on retrieving and summarizing well-organized incidents for root cause analysis. Unlike prior work, we aim to maximize the utilization of LLMs and fault examples. The proposed *LasRCA* simultaneously overcomes the label scarcity challenge of traditional approaches and the excessive cost challenge of LLMs. It is a complementary direction to existing approaches.

8 CONCLUSION

This paper addresses the challenge of fault label scarcity in root cause analysis (RCA) by focusing on an extreme one-shot scenario, where each fault type has only one example. We propose *LasRCA*, a framework that combines the strengths of the large language model and the small classifier to handle this scenario effectively. *LasRCA* consists of two stages. In the training stage, the small classifier is initially trained on one-shot fault examples and iteratively selects high-confusion samples. The LLM then provides fault type feedback for these samples, and the small classifier is retrained based on this feedback. In the inference stage, the small classifier first conducts preliminary RCA. Then the LLM provides detailed judgments based on troubleshooting guides, one-shot fault examples, and fault type confidence from the small classifier. Experiment results on public datasets show the effectiveness of *LasRCA* in one-shot RCA, achieving a trade-off between effectiveness and cost.

9 ACKNOWLEDGMENTS

This work was supported by National Science and Technology Major Project of China (Grant No. 2022ZD0120303).

REFERENCES

[1] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. 2016. Taking the blame game out of data centers operations with NetPoirot. In *Proceedings of the ACM Web Conference 2023 (SIGCOMM '16)*. 440–453. https://doi.org/10.1145/2934872.2934884

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (2004), 11–33. https://doi.org/10.1109/TDSC.2004.2

[3] Jasmin Bogatinovski, Sasho Nedelkoski, Li Wu, Jorge Cardoso, and Odej Kao. 2022. Failure identification from unstable log data using deep learning. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 346–355. https://doi.org/10.1109/CCGrid54584.2022.00044

[4] Shaked Brody, Uri Alon, and Eran Yahav. 2022. How attentive are graph attention networks?. In *International Conference on Learning Representations*. https://openreview.net/forum?id=F72ximsx7C1

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*. 1877–1901.

[6] 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 *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. 1887–1895. https://doi.org/10.1109/INFOCOM.2014.6848128

[7] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang, Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. 2024. Automatic Root Cause Analysis via Large Language Models for Cloud Incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. 674–688. https://doi.org/10.1145/3627703.3629553

[8] Yujun Chen, Xian Yang, Qingwei Lin, Hongyu Zhang, Feng Gao, Zhangwei Xu, Yingnong Dang, Dongmei Zhang, Hang Dong, Yong Xu, Hao Li, and Yu Kang. 2019. Outage prediction and diagnosis for cloud service systems. In *The World Wide Web Conference (WWW '19)*. 2659–2665. https://doi.org/10.1145/3308558.3313501

[9] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. arXiv:1903.02428 [cs.LG] https://arxiv.org/abs/1903.02428

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[11] Google. 2024. Online Boutique. https://github.com/GoogleCloudPlatform/microservices-demo.

[12] Yongqi Han, Qingfeng Du, Ying Huang, Jiaqi Wu, Fulong Tian, and Cheng He. 2024. Artifacts accompanying LasRCA. https://github.com/AmanecerTrio/LasRCA.

[13] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*. 33–40. https://doi.org/10.1109/ICWS.2017.13

[14] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* 55, 12 (2023). https://doi.org/10.1145/3571730

[15] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A highly efficient gradient boosting decision tree. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. 3149–3157. https://dl.acm.org/doi/abs/10.5555/3294996.3295074

[16] Nane Kratzke and Peter-Christian Quint. 2017. Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study. *Journal of Systems and Software* 126 (2017), 1–16. https://doi.org/10.1016/j.jss.2017.01.001

[17] Samuli Laine and Timo Aila. 2017. Temporal Ensembling for Semi-Supervised Learning. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BJ6oOfqge

[18] D. N. Lawley. 1938. A Generalization of Fisher's z Test. *Biometrika* 30 (1938), 180–187. http://www.jstor.org/stable/2332232

[19] Cheryl Lee, Tianyi Yang, Zhuangbin Chen, Yuxin Su, and Michael R. Lyu. 2023. Eadro: An end-to-end troubleshooting framework for microservices on multi-source data. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1750–1762. https://doi.org/10.1109/ICSE48619.2023.00150

[20] Mingjie Li, Zeyan Li, Kanglin Yin, Xiaohui Nie, Wenchi Zhang, Kaixin Sui, and Dan Pei. 2022. Causal Inference-Based Root Cause Analysis for Online Service Systems with Intervention Recognition. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22)*. 3230–3240. https://doi.org/10.1145/3534678.3539041

[21] Zeyan Li, Junjie Chen, Rui Jiao, Nengwen Zhao, Zhijun Wang, Shuwei Zhang, Yanjun Wu, Long Jiang, Leiqin Yan, Zikai Wang, Zhekang Chen, Wenchi Zhang, Xiaohui Nie, Kaixin Sui, and Dan Pei. 2021. Practical root cause localization for microservice systems via trace analysis. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. 1–10. https://doi.org/10.1109/IWQOS52092.2021.9521340

[22] Zeyan Li, Nengwen Zhao, Mingjie Li, Xianglin Lu, Lixin Wang, Dongdong Chang, Xiaohui Nie, Li Cao, Wenchi Zhang, Kaixin Sui, Yanhua Wang, Xu Du, Guoqiang Duan, and Dan Pei. 2022. Actionable and interpretable fault localization for recurring failures in online service systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. 996–1008. https://doi.org/10.1145/3540250.3549092

[23] Jinjin Lin, Pengfei Chen, and Zibin Zheng. 2018. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In *Service-Oriented Computing*. 3–20. https://doi.org/10.1007/978-3-030-03596-9_1

[24] Weilan Lin, Meng Ma, Disheng Pan, and Ping Wang. 2018. FacGraph: Frequent anomaly correlation graph mining for root cause diagnose in micro-service architecture. In *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*. 1–8. https://doi.org/10.1109/PCCC.2018.8711092

[25] Ilya Loshchilov and Frank Hutter. 2017. SGDR: Stochastic Gradient Descent with Warm Restarts. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Skq89Scxx

[26] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*. Association for Computing Machinery, New York, NY, USA, 412–426. https://doi.org/10.1145/3472883.3487003

[27] Meng Ma, Weilan Lin, Disheng Pan, and Ping Wang. 2019. MS-Rank: Multi-metric and self-adaptive root cause diagnosis for microservice applications. In *2019 IEEE International Conference on Web Services (ICWS)*. 60–67. https://doi.org/10.1109/ICWS.2019.00022

[28] Meng Ma, Weilan Lin, Disheng Pan, and Ping Wang. 2022. ServiceRank: Root cause identification of anomaly in large-scale microservice architectures. *IEEE Transactions on Dependable and Secure Computing* 19, 5 (2022), 3087–3100. https://doi.org/10.1109/TDSC.2021.3083671

[29] Leonardo Mariani, Cristina Monni, Mauro Pezzé, Oliviero Riganelli, and Rui Xin. 2018. Localizing faults in cloud systems. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 262–273. https://doi.org/10.1109/ICST.2018.00034

[30] Yuan Meng, Shenglin Zhang, Yongqian Sun, Ruru Zhang, Zhilong Hu, Yiyin Zhang, Chenyang Jia, Zhaogang Wang, and Dan Pei. 2020. Localizing failure root causes in a microservice through causality inference. In *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. 1–10. https://doi.org/10.1109/IWQoS49365.2020.9213058

[31] Vijayaraghavan Murali, Edward Yao, Umang Mathur, and Satish Chandra. 2021. Scalable statistical root cause analysis on app telemetry. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 288–297. https://doi.org/10.1109/ICSE-SEIP52600.2021.00038

[32] Paolo Notaro, Jorge Cardoso, and Michael Gerndt. 2021. A survey of AIOps methods for failure management. *ACM Trans. Intell. Syst. Technol.* 12, 6 (2021), 1–45. https://doi.org/10.1145/3483424

[33] OpenAI. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.LG]

[34] OpenAI. 2024. Pricing. https://openai.com/api/pricing/.

[35] Yicheng Pan, Meng Ma, Xinrui Jiang, and Ping Wang. 2021. Faster, deeper, easier: Crowdsourcing diagnosis of microservice kernel failure from user space. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. 646–657. https://doi.org/10.1145/3460319.3464805

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. 8026–8037. https://dl.acm.org/doi/abs/10.5555/3454287.3455008

[37] Gerard Salton and Christopher Buckley. 1988. Term-weighting approaches in automatic text retrieval. *Information Processing & Management* 24, 5 (1988), 513–523. https://doi.org/10.1016/0306-4573(88)90021-0

[38] Kihyuk Sohn, David Berthelot, Nicholas Carlini, Zizhao Zhang, Han Zhang, Colin A Raffel, Ekin Dogus Cubuk, Alexey Kurakin, and Chun-Liang Li. 2017. FixMatch: Simplifying Semi-Supervised Learning with Consistency and Confidence. In *Advances in Neural Information Processing Systems (NIPS '20)*. 596–608. https://doi.org/10.5555/3495724.3495775

[39] Jacopo Soldani and Antonio Brogi. 2022. Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. *ACM Comput. Surv.* 55, 3 (2022), 1–39. https://doi.org/10.1145/3501297

[40] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. 2018. The pains and gains of microservices: A Systematic grey literature review.

Journal of Systems and Software 146 (2018), 215–232. https://doi.org/10.1016/j.jss.2018.09.082

[41] Cindy Sridharan. 2018. *Distributed systems observability: A guide to building robust systems.* O'Reilly Media.

[42] Ozan Tuncer, Emre Ates, Yijia Zhang, Ata Turk, Jim Brandt, Vitus J. Leung, Manuel Egele, and Ayse K. Coskun. 2019. Online diagnosis of performance variation in HPC systems using machine learning. *IEEE Transactions on Parallel and Distributed Systems* 30, 4 (2019), 883–896. https://doi.org/10.1109/TPDS.2018.2870403

[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. 6000–6010. https://doi.org/10.5555/3295222.3295349

[44] Ping Wang, Jingmin Xu, Meng Ma, Weilan Lin, Disheng Pan, Yuan Wang, and Pengfei Chen. 2018. CloudRanger: Root cause identification for cloud Native systems. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 492–502. https://doi.org/10.1109/CCGRID.2018.00076

[45] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Model. In *Advances in Neural Information Processing Systems*. 24824–24837.

[46] Canwen Xu, Yichong Xu, Shuohang Wang, Yang Liu, Chenguang Zhu, and Julian McAuley. 2023. Small Models are Valuable Plug-ins for Large Language Models. arXiv:2305.08848 [cs.LG] https://arxiv.org/abs/2305.08848

[47] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Shaochen Zhong, Bing Yin, and Xia Hu. 2024. Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond. *ACM Trans. Knowl. Discov. Data* 18 (2024). https://doi.org/10.1145/3649506

[48] Zihao Ye, Pengfei Chen, and Guangba Yu. 2021. T-Rank:A lightweight spectrum based fault localization approach for microservice systems. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 416–425. https://doi.org/10.1109/CCGrid51090.2021.00051

[49] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. 2021. MicroRank: End-to-end latency issue localization with extended spectrum analysis in microservice environments. In *Proceedings of the Web Conference 2021 (WWW '21)*. 3087–3098. https://doi.org/10.1145/3442381.3449905

[50] Yue Yuan, Wenchang Shi, Bin Liang, and Bo Qin. 2019. An Approach to Cloud Execution Failure Diagnosis Based on Exception Logs in OpenStack. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 124–131. https://doi.org/10.1109/CLOUD.2019.00031

[51] Zhihan Yue, Yujing Wang, Juanyong Duan, Tianmeng Yang, Congrui Huang, Yunhai Tong, and Bixiong Xu. 2022. TS2Vec: Towards Universal Representation of Time Series. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 8980–8987. https://doi.org/10.1609/aaai.v36i8.20881

[52] Chaoli Zhang, Zhiqiang Zhou, Yingying Zhang, Linxiao Yang, Kai He, Qingsong Wen, and Liang Sun. 2022. Netrca: An effective network fault cause localization algorithm. In *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 9316–9320. https://doi.org/10.1109/ICASSP43922.2022.9747882

[53] Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. 2017. Mixup: Beyond Empirical Risk Minimization. In *International Conference on Learning Representations*. 596–608. https://openreview.net/forum?id=r1Ddp1-Rb

[54] Shenglin Zhang, Pengxiang Jin, Zihan Lin, Yongqian Sun, Bicheng Zhang, Sibo Xia, Zhengdan Li, Zhenyu Zhong, Minghua Ma, Wa Jin, Dai Zhang, Zhenyu Zhu, and Dan Pei. 2023. Robust failure diagnosis of microservice system through multimodal data. *IEEE Transactions on Services Computing* (2023), 1–14. https://doi.org/10.1109/TSC.2023.3290018

[55] Xuan Zhang, Longxiang Xiong, Ningyuan Sun, Mingxia Wang, Hao Tang, and Yanxing Zhao. 2022. Accurate inference of unseen combinations of multiple root causes with classifier ensemble. In *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 9306–9310. https://doi.org/10.1109/ICASSP43922.2022.9746416

[56] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark System, and empirical study. *IEEE Transactions on Software Engineering* 47, 2 (2018), 243–260. https://doi.org/10.1109/TSE.2018.2887384