

# COCA: Generative Root Cause Analysis for Distributed Systems with Code Knowledge

Yichen Li<sup>†</sup>, Yulun Wu<sup>†</sup>, Jinyang Liu<sup>†</sup>, Zhihan Jiang<sup>†</sup>, Zhuangbin Chen<sup>‡</sup>, Guangba Yu<sup>§</sup>, Michael R. Lyu<sup>†</sup>

<sup>†</sup>The Chinese University of Hong Kong, Hong Kong SAR, China.

Email: {ycli21, ylwu24, jyliu, zhjiang22, lyu}@cse.cuhk.edu.hk, yugb5@mail2.sysu.edu.cn

<sup>‡</sup>School of Software Engineering, Sun Yat-sen University, Guangzhou, China.

Email: chenzhb36@mail.sysu.edu.cn

**Abstract**—Runtime failures are commonplace in modern distributed systems. When such issues arise, users often turn to platforms such as Github or JIRA to report them and request assistance. Automatically identifying the root cause of these failures is critical for ensuring high reliability and availability. However, prevailing automatic root cause analysis (RCA) approaches rely significantly on comprehensive runtime monitoring data, which is often not fully available in issue platforms. Recent methods leverage large language models (LLMs) to analyze issue reports, but their effectiveness is limited by incomplete or ambiguous user-provided information.

To obtain more accurate and comprehensive RCA results, the core idea of this work is to extract additional diagnostic clues from code to supplement data-limited issue reports. Specifically, we propose COCA, a code knowledge enhanced root cause analysis approach for issue reports. Based on the data within issue reports, COCA intelligently extracts relevant code snippets and reconstructs execution paths, providing a comprehensive execution context for further RCA. Subsequently, COCA constructs a prompt combining historical issue reports along with profiled code knowledge, enabling the LLMs to generate detailed root cause summaries and localize responsible components. Our evaluation on datasets from five real-world distributed systems demonstrates that COCA significantly outperforms existing methods, achieving a 28.3% improvement in root cause localization and a 22.0% improvement in root cause summarization. Furthermore, COCA’s performance consistency across various LLMs underscores its robust generalizability.

## I. INTRODUCTION

The reliability of complex distributed systems is critical for ensuring seamless operation and user satisfaction. However, despite best efforts, users may still encounter various issues and report them through issue tracking systems (such as JIRA [1]). These reports often contain essential information such as issue descriptions, runtime logs, and stack traces. It is crucial for developers to identify the root causes and mitigate them promptly to maintain system reliability. However, manually understanding these reports and identifying the underlying root causes can be labor-intensive and error-prone, given the complexity of modern distributed systems.

Significant research has been devoted to developing automatic root cause analysis (RCA) approaches [2]–[7]. Extensive approaches focus on using comprehensive run-time monitoring data (*e.g.*, logs, metrics, and traces) to locate root causes [4], [8]. They typically compare the fault-suffering data collected

from the faulty states with the fault-free data collected from the normal state of the system to identify root causes [2], [4], [5]. Nevertheless, this monitoring data is often unavailable in practice because it requires extensive infrastructure to collect the required data. For instance, commonly used issue tracking systems, such as JIRA [1], generally contain only issue descriptions and fault-suffering data in reports submitted by users [9]. Another related work is fault localization, which pinpoints specific locations in the code responsible for software failures. However, bug reproduction-based fault localization techniques [10]–[12] face challenges for localizing the faulty code in distributed systems, which is extremely challenging with only issue reports available [9], [13]. Furthermore, fault localization techniques are limited to addressing code-related issues and cannot diagnose the non-code-related issues, such as network failures.

To address the limitations of monitor data-driven RCA approaches, recent approaches leverage the semantic comprehension and reasoning capabilities of Natural Language Processing (LLMs) techniques to identify root causes from issue reports, particularly when only limited run-time monitoring data is available [7], [14]–[17]. Unfortunately, the information contained in user-submitted issue reports is often incomplete or ambiguous [9], [14]. Users typically can only provide symptom-related details, such as brief issue descriptions and snippets of log messages, as the complexity of the distributed systems makes it difficult for them to detail the underlying execution logic directly responsible for the issue. This lack of detailed execution information, including the execution logic proceeding to failure, architectural and interaction patterns embedded in system code limits the ability of LLMs to recommend the accurate root causes.

To bridge this gap, the core idea of this work is to accurately retrieve diagnostic clues (*e.g.*, execution logic) from the complicated source code of a distributed system to supplement RCA in data-limited issue reports. Given the limited information in an issue report (*e.g.*, issue description, logs, and stack traces), our approach deals with the problem by reconstructing the execution logic preceding the issue by identifying and analyzing the relevant code. This allows us to infer root causes even with incomplete textual data in the issue report. Inspired by the capabilities of LLMs to comprehend both code and natural language, our intuition is to leverage LLMs to reason

<sup>§</sup>Guangba Yu is the corresponding author.

the execution paths prior to issue and understand the logic from relevant code.

However, it is non-trivial to integrate knowledge from codebase for diagnosing root causes in distributed systems without execution. We have identified three major challenges. (1) Linking logs directly to specific code positions is difficult because detailed metadata, like line numbers, is often missing, and runtime log messages are usually different from the original logging statements [18], [19]. (2) The complexity of distributed systems, with various invocation mechanisms such as static calls and Remote Procedure Calls (RPCs)<sup>1</sup>, makes it hard to piece together execution paths from code positions. (3) Due to the context window limitations, LLMs struggle to understand long codebases [20], [21], making it difficult to use LLMs to analyze complex and extensive codebases of distributed systems during RCA.

**Our work.** To address these challenges, we propose COCA, the first code knowledge enhanced root cause analysis approach. To address the first challenge, the *logging source retrieval* phase employs a static analysis-based backtracking approach to accurately identify the source logging statements, even when the actual log messages differ significantly from the original logging statements. These logging statements then provide code snippets directly related to the issue reports, such as the lines of code executed immediately before the issue occurs. Next, we design an *execution path reconstruction* step to incorporate method invocations, thereby providing a more comprehensive execution context prior to the occurrence of an issue. Beyond analyzing call graphs between methods, we propose an RPC bridging method, aiming to reconstruct interactions prior to failure via RPCs. Then, the *failure-related code profiling* phase employs code snippet indexing and retrieval methods to analyze all obtained code snippets, which aims to profile the snippets into a more compact form. This allows them to fit within a reasonably sized window suitable for processing by LLMs. Finally, during the *root cause inference* phase, we construct a prompt based on historical issue reports, the target issue report, and all profiled code knowledge. This prompt is then used to query the LLM, which generates a thorough root cause summary and localized responsible components.

To evaluate COCA, we conduct a comprehensive evaluation based on the dataset collected from five real-world distributed systems. Our results demonstrate that COCA achieves the best performance overall metrics in both root cause summarization and localization. More precisely, COCA surpasses the current leading method by 28.3% in root cause localization (as measured by *Exact Match*) and by 22.0% in root cause summarization (e.g., *BLEU-4* [22]). Additionally, COCA consistently enhances performance across various underlying LLMs (e.g., LLaMa-3.1 [23]), thereby affirming its robust generalizability. Moreover, we explore the individual contributions of each phase and provide the corresponding reasoning.

<sup>1</sup>a widely-used protocol in distributed systems that allows a program to run a procedure on another machine as if it were local.

Bug ID	JobClient fails, removes files and in turn crashes MR AM	
Closed	Version: 0.23.0	Assignee: Maintainer
Major	Resolution: Fixed	Reporter: Reporter Name
<b>Description</b>		
We ran into this multiple times. MR JobClient crashes immediately. the application was created and added to the list of apps. So if the client contacts the RM, it returns the application which perhaps is still in NEW state.		
<b>Logs</b>		
10:52:35 INFO mapreduce.JobSubmitter: number of splits: xxx		
10:52:36 INFO mapred.YARNRunner: AppMaster capability = memory,		
10:52:36 INFO mapred.YARNRunner: Command to launch container for AM		
10:52:36 INFO mapred.ResourceMgrDelegate: Submitted application ID to RM		
10:52:36 INFO mapreduce.JobSubmitter: Cleaning up the staging area		
<b>Stack Traces</b>		
at Local Trace:		
org.apache...YarnRemoteExceptionPBImpl: failed to run job		
at org.apache...createYarnRemoteException		
at org.apache...getRemoteException		
at org.apache...submitJob		
at org.apache...submitJobInternal		
...		

Fig. 1: Example of an issue report.

This paper’s contributions are summarized as follows:

- To our best knowledge, COCA<sup>2</sup> is the first work incorporating code knowledge into the automatic root cause analysis framework for issue reports in distributed systems.
- We proposed and implemented COCA with multiple generalized novel tools (e.g., RPCBridge) which solve several critical challenges in RCA field.
- We collected and labeled a real-world issue dataset of distributed systems for RCA, which includes 106 real-world issues covering five widely used distributed systems.
- We extensively evaluate the performance of COCA on real-world datasets. The results demonstrate the effectiveness of COCA and the adaptability of COCA with different backbone models.

## II. BACKGROUND AND MOTIVATION

### A. Background

**Issue Reports in Distributed Systems.** Issue reports, including bug reports, GitHub issues, and cloud incident tickets [24], [25], are reported by users to track and manage various types of system problems. We unify the terms throughout the paper to issue reports for distributed systems.

As shown in Figure 1, when users encounter a system issue (e.g., JobClient fails), they typically report the problem on the system’s issue tracking platform as an issue report. An issue report usually contains attributes such as the *issue title*, *description*, *logs*, *stack trace*, and *meta information* (including version, severity, and status). The *description* is a free text written by users that describes the problem, aiding in understanding the issue. *Logs* offer a detailed record of system events and actions preceding the failure. Each log message typically includes a timestamp indicating when the event occurred, a text template, and dynamic variables that provide context-specific information (e.g., request IDs). Log sequences can reflect the execution path of the system within a specific time frame [26]–[28]. For instance, the provided logs in the issue report suggest that the JobClient failure occurred

<sup>2</sup><https://github.com/YichenLi00/COCA>

after posting the application to RM and the staging area clean up. *Stack traces* act as a snapshot of the call stack at the point where a failure occurred to pinpoint the context of the failure.

Addressing an issue report manually can be a time-consuming and tedious procedure [9], [13], [29], [30]. It requires rich domain knowledge from maintainers to understand, reproduce, and eventually fix the problem. This difficulty is further amplified in complex distributed systems.

### B. Motivating Study

In this section, we present a motivating example to demonstrate our key insight: using code knowledge to enhance system failure diagnosis. Specifically, we present a real-world issue report, MAPREDUCE-2953<sup>3</sup>, from the widely-used distributed system MapReduce [31].

A MapReduce job fails immediately after the user submits the job from the client machine. To diagnose this issue, a common practice is to investigate the error log messages to understand what went wrong. For example, the stack trace indicates that this job failed after submission. Once this error is identified, the next step is to trace back through the logs to find preceding events that might lead to the issue. In this case, line 8 is particularly suspicious because a `cleaning up` action is performed after the job is submitted.

While log analysis sheds some light, it falls short of revealing the underlying cause of the failure. To find out the rationale behind this problem, we need to understand the code execution logic that occurs before the issue arises. Specifically, ① the client submits an application to the Resource Manager (RM), and ② immediately requests an application report. However, at the time of the request, ③ the RM has not completed setting up the application in the staging area (*i.e.*, a temporary workspace to organize the necessary resources to run an application), returns the NULL to the client. Due to the incomplete setup, ④ the client then mistakenly perceives the application as removed and initiates a cleanup process, while printing the log message “Cleaning up the staging area...”. This action occurs while the RM is still reading the staging area after receiving the application submitted from the client. This race condition results in a failed job run.

In this case, existing solutions [3], [14], [16], which solely rely on the analysis of reports, exhibit a significant limitation: they can only confirm job failure and pinpoint the suspicious cleanup action. However, accurately determining the root causes without a comprehensive understanding of the job’s execution is hard, without the involvement of the relevant code.

To address this limitation, we introduce an innovative approach for a more thorough RCA. Our approach synergies runtime log data with the corresponding executed code, thereby providing a deeper comprehension of job failures. Specifically, we plan to leverage the capabilities of LLMs, which have demonstrated proficiency in comprehending both incident tickets [14], [16] and code [32], [33], to enhance the issue diagnosis process.

<sup>3</sup><https://issues.apache.org/jira/browse/MAPREDUCE-2953>

### C. Challenges

While the integration of code knowledge into the RCA procedure seems intuitive, it poses three main challenges:

**Challenge 1: Sourcing Log Messages.** While stack traces provide specific *class names*, *file names*, and *line numbers* for precise location, sourcing the original code lines of log messages is complex [19]. Ideally, we could use the above mentioned metadata (*e.g.*, *line numbers*) provided by logging libraries for precise localization. However, obtaining such detailed metadata is usually not feasible in real-world scenarios [34]–[36]. Furthermore, runtime log messages often differ significantly from the logging statements in the code [19], [37], making it difficult to for a precise match. In particular, the variable in the logging statement may consist of multiple dynamic variables across different branches [26], [29].

**Challenge 2: Reconstructing Execution Paths.** The complexity of execution paths in distributed systems makes their reconstruction from logs and stack traces in issue reports a formidable task [27], [38]. Building the dependencies between these code pieces is highly complex due to the diverse nature of dependencies in distributed systems, including static calls, gRPC calls, and others. These complexities block the accurate determination of the code executed prior to failures.

**Challenge 3: Profiling Failure-Related Code Snippets.** Assuming the ability to reconstruct the execution paths prior to failures, this would involve a vast amount of code. Failure-related code knowledge comes not only from the executed code but also from its dependencies [39]. The absence of these dependencies could render code slices hard to comprehend. However, including all executed and dependent code significantly expands the context length, potentially exceeding the model’s input limit, leading to high costs and model confusion.

## III. METHODOLOGY

### A. Overview

To address the above challenges, we propose COCA, a framework that enhances RCA of issue reports in distributed systems through code knowledge.

As illustrated in Figure 3, COCA takes the issue report along with the associated project code of the corresponding version as input and generates the diagnostic result through a four-phase process. (I) In **Logging Source Retrieval** phase, COCA localizes the code positions (*i.e.*, the line numbers of the corresponding logging statements) of log messages in issue reports and maps out the code points spreading across the systems. (II) In **Execution Path Reconstruction** phase, the identified code points are used to reconstruct the execution paths leading up to the failure with a call graph patched with RPC edges and inter-method analysis. (III) In **Failure-related Code Profiling** phase, COCA analyzes and indexes the extracted method-level code snippets with execution orders with signatures and documents. The LLM can retrieve the full method code of the code snippet based on indexes and initial comprehension of the issue. (IV) During **Root Cause Inference** phase, in conjunction with similar historical issue reports,

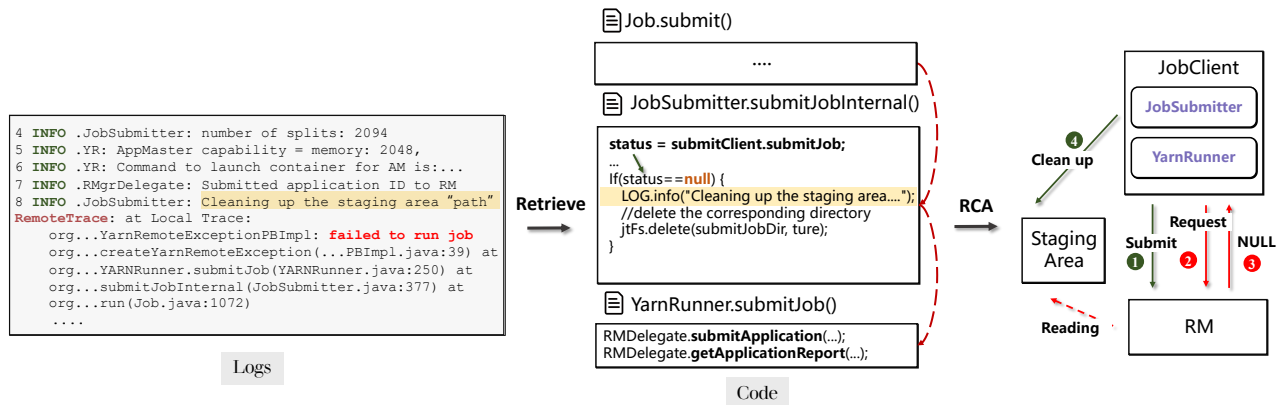


Fig. 2: A motivating example: MapReduce-2953.

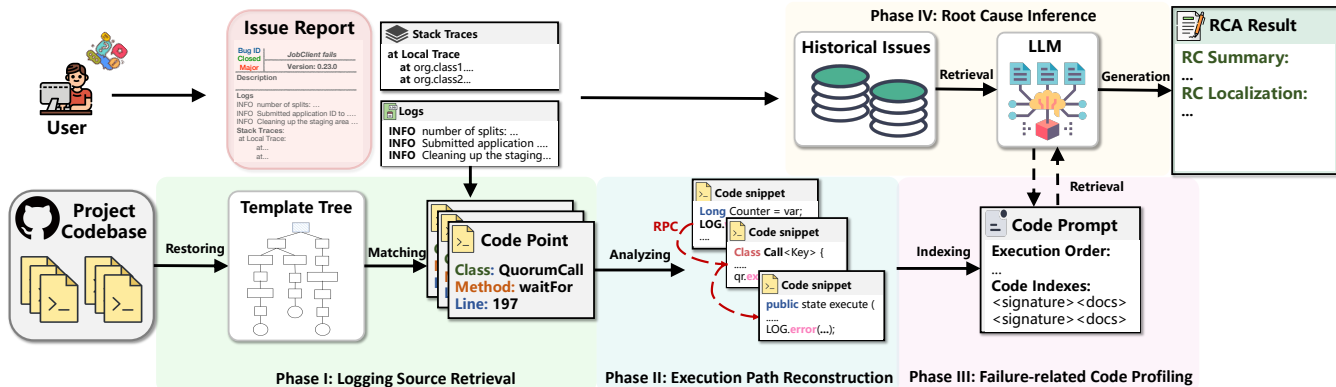


Fig. 3: The workflow of COCA.

```
String msg = String.format(
    "Waited %s ms (timeout=%s ms) for a response for %s", waited, millis, operator);
if (!successes.isEmpty()) {
    msg += ". Succeeded so far: " + String.join(", ", successes.keySet());
}
if (successes.isEmpty() && exceptions.isEmpty()) {
    msg += ". No responses yet.";
}
LOG.warn(msg);
```

hdfs/qjournal/client/QuorumCall.java

Fig. 4: An example of logging statement restoring.

COCA diagnoses the issue using all acquired knowledge and outputs a comprehensive root cause summary and components.

### B. Logging Source Retrieval

To address the **Challenge 1** and identify the precise logging statement sources from the code of log messages in the given issue report, COCA employs a two-step approach after executing all logging statements and their corresponding code positions, which is achieved by analyzing the project’s logging library [40]. The first step involves restoring the logging statements that contain constructed variables [26] back to primitive. For the second step, we introduce a template match algorithm to match log messages with restored logging statements of the corresponding system version.

1) *Restoring Logging Statements*: This step involves restoring logging statements by resolving constructed variables (e.g., `Log.warn(msg)`), which is further used for matching log messages to their corresponding logging statements. As

shown in Figure 4, the raw logging statement only indicates the constructed variable `msg`, instead of the primitive template with constant strings and original variables.

To tackle this issue, we analyze data dependency and control flow to obtain the primitive template set. Inspired by previous work [37], COCA employs a backtracking approach based on static analysis to identify and restore variables’ paths, allowing COCA to restore logging statements and obtain uncompressed string constants and variables from a simple `msg`. Specifically, variable `msg` is constructed through several steps via conditional branches. COCA restores `msg` back to its original constant strings and variables for further matching. Specifically, after restoring, the logging statement in Figure 4 is reinstated to four different primitive templates (e.g., `Waited <*> ms (timeout=<*>)` for a response for `<*>`. No responses yet.) for each control branch.

2) *Log Template Matching*: Upon restoring all the primitive log templates, the subsequent step involves matching log messages to templates for determining the code positions.

During the process of log message matching, a single log message could potentially be matched to multiple logging templates [35], [41]. For instance, there are two templates  $T_1$ : (“Job created, executing ” + `<*>`) and  $T_2$ : (“Job created, executing ” + `<*>` + “ on node ” + `<*>`). Every log message generated from template  $T_2$  could be potentially matched by  $T_1$ . This might be due to the fact

---

**Algorithm 1: Template Matching Procedure**

---

```
1 Input: Template prefix tree; Log message L;  
2 Output: Matched template  $t_m$ ;  
3 Function match(root, L, results) :  
4   if len(L) = 0 and root is a leaf node then  
5     add template of root to results; return;  
6   if L[0] in root.children then  
7     match(root[L[0]], L[1:], results) ;  
8   if <*> in root.children then  
9     for i  $\leftarrow$  0 to len(L)-1 do  
10      if L[i] in root[<*>].children then  
11        match(root[<*>], L[i:], results) ;  
12 root  $\leftarrow$  root node of the template tree;  
13 L, results  $\leftarrow$  split_tokens(log message), [ ];  
14 match(root, L, results) ;  
15  $t_m \leftarrow$  most_static_template(results);  
16 return  $t_m$ ;
```

---

that for some complexly constructed variables (*e.g.*, *msg*), it is challenging to restore them thoroughly, leaving some unsolved variables to have such a general template.

To address this issue, we proposed a template matching Algorithm 1 to match log messages back to logging statements accurately. Different from general logging parsing problems [42], [43], COCA performs a non-pruneable recursive matching to review all templates within the prefix tree, maintaining a global perspective. When a log message can match several log templates, COCA prioritizes the template with the most static parts (*i.e.*, non-`<*>` characters). Noted that COCA currently only analyzes logs from the system being diagnosed without analyzing logs from third-party libraries, as COCA focuses on helping maintainers of target systems.

### C. Execution Path Reconstruction

After successfully matching the code positions of log messages in a given issue report, the subsequent task is to figure out how to incorporate these code points into the RCA process. An intuitive approach is extracting the surrounding lines of logging statements to profile the logs. However, this method presents challenges when applied to distributed systems, where logs are not densely populated [9], [13], [44]. For two adjacent log messages, numerous methods might execute across different components. Merely extracting surrounding lines without reconstructing and incorporating the executed code may miss important system runtime information [9], [26], [45], [46].

In order to comprehend the pre-failure execution state of the system, we need to address the **Challenge 2**: reconstructing the execution paths prior to failure based on identified code points. Following the previous works [26], [27], [46], [47], COCA constructs the *inter-procedure control flow graph* (ICFG) for connecting the code points with execution code paths. Typically, the construction of the ICFG involves both *inter-method*

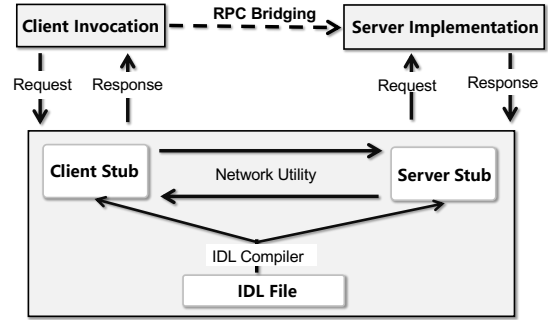


Fig. 5: A common RPC call invocation process.

and *intra-method* analysis. *Inter-method* analysis primarily focuses on identifying the call graph of the system, which includes determining the relationships of invocations between different methods, both direct and indirect. Conversely, *intra-method* analysis resolves the control flow within each individual method to identify its potential execution paths.

Additionally, in distributed systems, simply analyzing the call graph to model the invocation relationships is insufficient, given the widespread use of RPC [48]. RPC is a prevalent mechanism for facilitating interactions between individual instances [49]. These calls are made dynamically (*i.e.*, via network utilities), making it hard to statically capture the call edges between instances. Worse still, existing approaches leave this issue unsolved [26], [27]. To address this, we propose an RPC bridging method suitable for common RPC frameworks such as gRPC [50] and Thrift [51], to improve the precision of constructing call graphs in distributed systems.

Figure 5 shows the typical invocation process for a remote call under common RPC frameworks. Initially, RPC functions defined by the Interface Description Language (IDL) are transformed into *client* and *server* stubs. These stubs provide an interface for handling remote calls, including serialization and message sending. Subsequently, the *client* invokes the *client stub*, which ultimately connects to the server stub to call the server’s service functional implementation. In this process, the client stub invokes the server implementation via network utilities. However, as the actual call targets are determined at runtime, this dynamic binding introduces significant uncertainty for static analysis [52]–[54], thus breaking the call stack between client and server.

To address this uncertainty, we leverage the implementation patterns in the call process to bridge the call stack: COCA first identifies client invocation to RPC functions by parsing the IDL files (*e.g.*, proto files for gRPC and thrift files for Thrift) to obtain the RPC function names and interface names. COCA then searches for matched callees in the call graph, confirming them as client-to-server RPCs. Second, COCA extracts the server-side implementation classes, which typically have names similar to their interface counterparts (*e.g.*, the interface *ResourceTrack* and its implementation *ResourceTrackService*). We use substring matching for fuzzy identification. To prevent false positives, we further examine the declarations of the matched classes to ensure the implementation of corresponding interface classes. Finally, we replace the callee



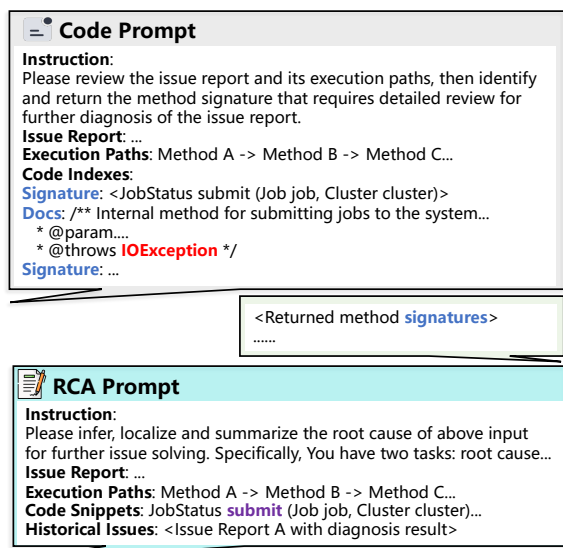


Fig. 6: Example prompt in COCA.

of client-to-server RPCs with the corresponding server-side implementation functions, completing the client-to-server call stack bridging, as shown by the dashed line in Figure 5.

With this patched call graph and inter-method analysis, COCA identifies multiple execution paths based on specified code points for the subsequent step. Note that while COCA currently supports gRPC [50] framework, it can be readily adapted to other RPC (e.g., Thrift [51]) frameworks by replacing the code that parses the RPC definition files (i.e., IDL files)

#### D. Failure-Related Code Profiling

Upon reconstructing the execution paths based on code points the failure, **Challenge 3** emerges: *How to deal with thousands of lines of executed code?*

The code profiling phase aims at profiling the code information while concurrently reducing redundant failure-unrelated code snippets and minimizing the cost (i.e., context length) of COCA. To retrieve the code snippets that contribute to failure comprehension or are prone to failure, and forward them to the backbone LLM, COCA utilizes a two-staged approach that involves indexing and retrieval.

1) *Code Snippet Indexing*: To streamline the process of retrieving relevant code snippets, we implement a method-level indexing framework for executed paths. The indexing framework creates individual indexes for each method in the execution paths, equipping the backbone model with sufficient key information to make selections without the need to process the entire method code. Given the experiment datasets (seen Section IV-A) were collected from well-maintained systems, we use standard method docs along with the method signature generated by Soot [55] as indexes, as shown in code prompt part of Figure 6. If docs are unavailable, we can generate standard docs using LLM-based methods [39], [56], [57].

2) *Code Snippet Retrieval*: We provide the LLM with a comprehensive understanding of the issue report and analyzed execution paths, along with indexes for subsequent retrieval.

With the initial comprehension of the target issue report and execution paths preceding to the failure, COCA will allow LLM to identify code snippets that contribute to failure comprehension or are prone to failure. To accomplish this, we input both the issue report and the code snippet indexes of all analyzed execution paths into the backbone LLM for retrieval from the codebase. As shown in Figure 6, it will return the method signatures that require detailed review. Based on these generated method signatures, the corresponding method code is then extracted, preparing for the next phase.

#### E. Root Cause Analysis

1) *In-context Learning*: To learn the historical experiences from previous issue reports of current system, COCA employs the in-context learning (ICL) strategy [33], which has proved its effectiveness in bug diagnosis task [14], [16], [58]. The ICL strategy improves the effectiveness of LLMs based on historical experience. Specifically, this strategy provides a few historical issue reports sampled from the collected and labeled dataset, which was detailed in Section IV-A

However, the inherent heterogeneity of issue reports presents the challenge [9] of finding similar and useful historical examples. Embedding-based similarity metrics, though proved effective for certain applications [14], [16], may not perform optimally in the context of diverse issue reports [33].

We use the BM25 [59] similarity to select these examples. The BM25 function, based on the Term Frequency-Inverse Document Frequency (TF-IDF) method, places greater emphasis on keywords, making it particularly suitable for component matching among issue reports. We specifically select the top five historical issue reports with the highest similarity scores.

2) *Diagnosis Result Inference*: As illustrated in Figure 6, after comprehensively analyzing the issue report, reconstructed execution paths, and retrieved code snippets, COCA is well-equipped to infer the root cause of the issue.

COCA structures the prompt by beginning with the issue report, followed by the relevant knowledge obtained from the analysis that contributes to the root causes analysis. Specifically, COCA focuses on two key tasks: root cause summarization and localization. The goal of root cause summarization is to provide a clear and concise guide for maintainers in their subsequent mitigation and fixing efforts. In contrast, root cause localization aims to provide the analyzed scope for the actual root cause, facilitating further fixes by enumerating the potential root cause components ranked by likelihood.

## IV. EXPERIMENT SETUP

We want to answer the following research questions (RQs).

- RQ1: How effective is COCA in RCA compared with existing methods?
- RQ2: What are the impacts of different phases in COCA?
- RQ3: How generalizable is COCA with different backbone models?

**TABLE I:** Details of the datasets.

System	SLOC*	Failure Types (#)	# Issues
MapReduce	721K	Hangs (7), Inconsistent state (5), Incorrect result (5), Resource exhaustion (2), Resource leak (1), Unexpected termination (6)	26
HDFS	706K	Data loss (1), Hangs (1), Inconsistent state (1), Incorrect result (8), Resource leak (1), Unexpected termination (3)	15
HBase	912K	Data loss (4), Hangs (8), Inconsistent state (2), Incorrect result (2), Unexpected termination (8)	24
Cassandra	1.1M	Hangs (6), Inconsistent state (5), Incorrect result (20), Unexpected termination (2)	33
ZooKeeper	184K	Hangs (4), Incorrect result (2), Resource leak (2)	8
Total	-	Data loss (5), Hangs (26), Inconsistent state (13), Incorrect result (37), Resource exhaustion (2), Resource leak (4), Unexpected termination (19)	106

\*SLOC is calculated based on the latest version. The actual SLOC may vary depending on the version in which the issue occurred.

### A. Dataset

1) *Collection*: Our experiments utilize datasets from a previous study [9], collected from the JIRA [1], a public issue-tracking platform. As shown in Table I, the systems from which these issues were collected range from 144K to 3M lines of code, with 11 to 15 years of development. These systems are distributed and include computing, storage, and configuration frameworks. All the issues we studied are distributed issues, meaning the failure propagation involves multiple nodes or components, which makes RCA particularly challenging. To make our analysis more rigorous, we excluded issues that lacked run-time logs and filtered out those that lacked a detailed root cause analysis upon the closure.

2) *Labelling*: All the issues in our study have been fixed and thoroughly investigated [9], [60], [61]. The discussions and patches related to these issues are publicly available, which facilitates the labeling process. Two experienced developers, both are contributors of at least one above system, undertook the labeling procedure jointly.

To label *root cause summary*, annotators read and summarize the root cause identified from the discussions in the issue report and previous studies [9]. They then distill the diagnostic procedure from the discussion, refining the summary based on mitigation strategies and fixing items. The goal of the summary is to provide a clear and concise guide for maintainers to diagnose the issue. For *root cause localization*, due to differences in the granularity [2], [4], [62], we standardized the process by identifying all components mentioned within the discussion and postmortem studies, allowing all approaches to select from them. These components include code-related elements such as specific classes (e.g., JobClient), as well as non-code-related components including system resources (e.g., network, RPC) and abstract concepts (e.g., race conditions, deadlocks). The ground truth for root cause localization is derived from the studied root cause and the corresponding fixing patches, and it consists of one or two components.

### B. Evaluation Metrics

1) *Root Cause Summarization*: We use **BLEU** [22], **ME-TEOR** [63], and **ROUGE** [64] metrics, consistent with previous research [14], [16], [58], to evaluate the quality of the generated root cause summarization from the aspect of text similarity with ground truth. We also leverage widely-used

embedding models, OpenAI embedding [65], to embed the root cause summary and calculate the **Semantics** similarity.

Following previous works [10], [66], [67], we also employ human evaluation **Usefulness** indicates whether the summary accurately explains why the issue occurred and how to fix it. The same experts who annotated the datasets conducted this evaluation, ensuring consistency and a high level of expertise. They are offered the summary from all baselines alongside the corresponding ground truth for each issue. The evaluators are instructed to compare these results against the ground truth and rate the usefulness for diagnosing the given issue of each summary on a scale from 0 to 1. We then calculate the average score among evaluators for certain baseline.

2) *Root Cause Localization*: We use the following two metrics to evaluate the effectiveness of COCA and baselines.

- **Exact Match** measures whether the generated root cause components exactly match the ground truth set of components. Any mismatch, either fewer or more components than the ground truth, results in a score of 0.
- **Top-k** measures whether the ground truth components are completely included within the top-k (k=3, 5) most-likely root cause components generated by models. For this metric, we prompt the model to generate the k most likely root cause components.

### C. Comparative Methods

We deliberately refrained from comparing COCA with traditional [4], [29] and AIOps methodologies [2], [6], [68] that utilize fault-free data during the procedure. Typically, when issues are reported and tracked using platforms such as JIRA [1], obtaining fault-free comparative data is not feasible [9], [14]. Additionally, we excluded AutoFL [10] because solely based on ticket, it was hard to reproduce the bugs [13] of distributed system and obtain the runtime coverage.

We selected RCACopilot [14] and ReAct [16] as our primary baselines because of their similar RCA procedures and capabilities in handling both localization and summarization tasks. Both models will retrieve the historical issue reports as knowledge sources to enhance the performance of root cause analysis. Since neither model is open-source, we **reimplemented** their methods based on the corresponding papers. We also included the base model with five fixed examples to demonstrate the task for comparison. All methods utilize the **same backbone LLM** to ensure a fair comparison.

#### D. Implementation Details

The static analysis module in COCA has been implemented using approximately 4,652 lines of code, written in both Java and Python. This development employs Soot [55] and Eclipse JDT Core [69], facilitating joint analysis of both Java bytecode and source code. The experiments of COCA and all baselines were conducted on a Linux machine (Ubuntu LTS 20.04).

For GPT-4o, GPT-3.5 [70], LLaMa-3.1 [23], Gemini-1.5-Pro [71] and Claude-3.5 [72], we use the public APIs provided by OpenAI [70], DeepInfra [73], Google and Anthropic [72], corresponding to the model *GPT-4o-0513*, *GPT-3.5turbo-0125*, *LLaMa-3.1-405b*, *Gemini-1.5-Pro* [71] and *Claude-3.5-Sonne*, respectively. We set the *temperature* of all models as zero to ensure the reproduction. The default number of sampled examples is set to 5 for all approaches. The depth of analyzed constructed execution path is 2.

### V. EVALUATION RESULTS

#### A. RQ1: Effectiveness of COCA

To evaluate the performance of COCA in RCA tasks, we conduct a thorough evaluation with comparison to other baselines. The results of this evaluation are detailed in Table II and all approaches employ *GPT-4o-0513* as the uniform backbone model. We analyze the evaluation results from two dimensions: *Root Cause Summarization* and *Root Cause Localization*.

1) *Root Cause Summarization*: We compare COCA with baselines in terms of root cause summarization from the aspects of syntax similarity, semantic similarity, and human-evaluated metrics. Regarding syntax similarity, we observe that COCA outperforms the best performing approach, RCA-Copilot, showing improvements of 22.0% in *BLEU-4*, 6.8% in *ROUGE-1*, and 10.6% in *METEOR* scores, on average. These enhancements are even more pronounced in specific system: for instance, when analyzing issues from ZooKeeper, COCA surpasses the best-performing approach by 28.5% in *BLEU-4*. For semantic similarity, COCA consistently outperforms the best-performing baseline with improvements ranging from 2.3% to 8.7% across all evaluated systems, providing another dimension of effectiveness against the baselines. In terms of *Usefulness*, which could directly measure the help for the maintainers, COCA exhibits substantial enhancements. This underscores the practical impact of integrating code knowledge in real-world distributed system maintenance.

2) *Root Cause Localization*: According to the evaluation results, it is clear that COCA outperforms all baselines in faulty components localization. COCA outperforms the best performing baseline in all evaluated systems across almost all metrics. Specifically, when averaged across all systems, COCA outperforms RCACopilot by 28.3%, 20.7% and 4.3% for metric *Exact Match*, *Top-3* and *Top-5*, respectively. Notably, despite utilizing the same foundational model (*i.e.*, GPT-4o), COCA achieves a remarkable 56.3% improvement of the base model in *Exact Match*, the most challenging metric, underscoring the effectiveness in the task of root cause localization. These results underscore the importance of integrating internal

system knowledge (*i.e.*, code) over relying solely on historical data (*e.g.*, ReAct) in RCA tasks.

**Answer to RQ1.** By incorporating the code knowledge into the procedure of RCA, COCA demonstrates superior performance in both dimensions of root cause summarization and localization, significantly outperforms all baselines and especially the base model.

#### B. RQ2: Impact of different phases in COCA

To evaluate the individual contribution and parameter setting of each phase within the COCA, we conducted an ablation study by creating three variants of COCA, each missing a different phase. Specifically, to evaluate the framework without the logging source retrieval phase, we adopted a new code-enhanced method that retrieves the top 5 methods (calculated by OpenAI embedding [65] similarity) from the project code-base for comparison. Additionally, to estimate the impact of analyzed execution depth, we also investigated execution depth parameters ranging from 1 to 3.

Table III demonstrates the experimental results. The results indicate that without conducting the logging source retrieval phase, the overall performance of COCA generally declines across all metrics. There is a decrease of 18% in *Exact Match*, while *METEOR* and *Usefulness* fall by 22.9% and 14.9% respectively. More critically, some metrics (*e.g.*, *METEOR*, *Top-5*) even fall below those of the raw LLM. This suggests that when the retrieved code snippets are irrelevant to the failure at hand, they can potentially mislead the model. Additionally, when execution path reconstruction is not conducted, the overall performance of COCA also deteriorates significantly. The *BLEU-4* score drops from 0.205 to 0.171 and the *Usefulness* score from 0.776 to 0.685, representing decreases ranging from 16.6% to 11.7%, respectively. It is noteworthy that the *Exact Match* performance is even lower than search-based method. This indicates that without analyzing the execution paths and relying solely on the log surrounding code snippets, it is challenging for the LLM to accurately identify the root cause. As shown by our experiments with different execution depths, where each invocation is treated as one degree, 1-degree paths may miss critical code information, while 3-degree paths may introduce too much irrelevant code. The experiment results suggest that 2-degree paths (default setting of COCA) provide optimal performance in this scenario. Additionally, regarding RPC invocations, approximately 15.7% of the invocations in a reconstructed execution path are RPC invocations, demonstrating the effectiveness of proposed RPCBridge.

Furthermore, when the phase of failure-related code profiling is omitted, the performance of COCA for root cause localization drops obviously (16.1%, reflected by *Exact Match*), while the performance of root cause summarization remains relatively high. Furthermore, it is puzzling to observe that the *Top-5* accuracy remains at a similar level with *Top-3* accuracy. Upon investigating this phenomenon, we discovered that it is due to extremely long code contexts. These results underscore the critical role of the failure-related code profiling phase.



**TABLE II:** Root cause analysis results from both *summarization* and *localization* dimensions for each system.

System	Model	Root Cause Summarization					Root Cause Localization		
		BLEU-4	ROUGE-1	METEOR	Semantics	Usefulness	Exact Match	Top-3	Top-5
MapReduce	Base model	0.147	0.451	0.354	0.781	0.645	0.346	0.615	0.692
	RCACopilot	0.174	0.503	0.396	0.812	0.740	0.346	0.692	0.808
	ReAct	0.172	0.494	0.384	0.804	0.683	0.385	0.654	0.808
	COCA	<b>0.203</b>	<b>0.525</b>	<b>0.445</b>	<b>0.867</b>	<b>0.795</b>	<b>0.423</b>	<b>0.731</b>	<b>0.923</b>
HDFS	Base model	0.145	0.430	0.346	0.787	0.627	0.200	0.400	0.600
	RCACopilot	0.196	0.479	0.407	0.835	0.650	0.400	0.667	0.800
	ReAct	0.192	0.486	0.413	0.843	0.733	0.400	0.533	0.733
	COCA	<b>0.219</b>	<b>0.501</b>	<b>0.454</b>	<b>0.883</b>	<b>0.754</b>	<b>0.467</b>	<b>0.800</b>	<b>0.800</b>
HBase	Base model	0.167	0.487	0.385	0.817	0.683	0.250	0.542	0.917
	RCACopilot	0.171	0.505	0.408	0.873	0.734	0.292	0.458	0.958
	ReAct	0.185	0.496	0.416	0.844	0.729	0.375	0.458	0.917
	COCA	<b>0.200</b>	<b>0.526</b>	<b>0.435</b>	<b>0.893</b>	<b>0.745</b>	<b>0.417</b>	<b>0.667</b>	<b>0.958</b>
Cassandra	Base model	0.149	0.451	0.340	0.784	0.648	0.364	0.576	0.697
	RCACopilot	0.153	0.466	0.392	0.807	0.661	0.424	0.697	0.879
	ReAct	0.165	0.484	0.407	0.796	0.691	0.515	0.727	0.848
	COCA	<b>0.208</b>	<b>0.522</b>	<b>0.441</b>	<b>0.877</b>	<b>0.830</b>	<b>0.606</b>	<b>0.848</b>	<b>0.909</b>
ZooKeeper	Base model	0.129	0.429	0.333	0.756	0.588	0.250	0.500	0.875
	RCACopilot	0.144	0.468	0.357	0.863	0.613	<b>0.375</b>	0.750	1.000
	ReAct	0.136	0.438	0.342	0.828	0.594	0.250	0.625	0.875
	COCA	<b>0.185</b>	<b>0.497</b>	<b>0.381</b>	<b>0.894</b>	<b>0.625</b>	0.250	<b>0.875</b>	<b>1.000</b>

**TABLE III:** Ablation Study of COCA.

System	Setting	Root Cause Summarization					Root Cause Localization		
		BLEU-4	ROUGE-1	METEOR	Semantics	Usefulness	Exact Match	Top-3	Top-5
All	COCA	<b>0.205</b>	<b>0.519</b>	<b>0.438</b>	<b>0.880</b>	<b>0.776</b>	<b>0.472</b>	<b>0.774</b>	<b>0.915</b>
	w/o Logging Source Retrieval	0.145	0.462	0.338	0.815	0.660	0.387	0.557	0.660
	w/o Execution Path Reconstruction	0.171	0.472	0.403	0.827	0.685	0.349	0.632	0.821
	w/ 1-Degree Execution Path Reconstruction	0.189	0.507	0.373	0.872	0.723	0.443	0.698	0.842
	w/ 3-Degree Execution Path Reconstruction	0.201	0.524	0.419	0.891	0.693	0.349	0.745	0.906
	w/o Failure-Related Code Profiling	0.175	0.507	0.399	0.847	0.734	0.396	0.679	0.792
	only w/ Full JIRA Discussion	0.495	0.712	0.775	0.914	0.895	0.849	0.906	0.934

**Answer to RQ2.** The ablation study indicates that the elimination of any component notably reduces the overall performance. Particularly, the results demonstrate that merely searching the codebase without the COCA framework falls short of expectations, emphasizing that each component contributes to the overall performance.

### C. RQ3: Generalizability of COCA

In this RQ, we evaluate the performance of COCA by utilizing various LLMs in conjunction with our framework. We have selected three representative LLMs that are frequently used in research, specifically *GPT-3.5*, *GPT-4o*, *Llama-3.1-405b*, *Gemini-1.5-Pro*, and *Claude-3.5-Sonnet*. We selected versions of these models with enough parameter size to ensure the capability of complex prompt understanding [33], [74].

The experimental results are shown in Table. IV. Our observations indicate that the COCA framework can consistently improve the performance of all utilized base models in terms of all metrics by a large margin. On average, all models have been improved by 43.3%, 29.7% and 15.3% in localizing the root causes based on the metrics of *Exact Match*, *Top-3*, and *Top-5* respectively. Furthermore, when it comes to summarizing the root causes, we observed an average improvement of 33.2% (as reflected by *BLEU-4*),

20.2% (as reflected by *METEOR*), and 14.4% (as reflected by *Usefulness*). Particularly noteworthy is the performance of the *Claude-3.5-Sonnet* model within the COCA framework, which surpasses the raw model in *Top-3*. The experiments compared the performance of the latest and most powerful LLMs in root cause analysis tasks. The results indicate that although *GPT-4o* is not the most advanced raw model from some aspects (e.g., *Exact Match*), the enhancements provided by COCA are still the most significant.

The results not only demonstrate the advantage of COCA’s design but also demonstrate the generalizability for different backbone models. We believe that the performance of COCA can be further improved with the development of LLMs.

Furthermore, the experiments with full JIRA discussion in Table. III further demonstrate COCA’s generalizability. The results show dramatic improvements after incorporating detailed JIRA issue discussions, indicating that those LLMs have limited memorization of JIRA issue results. Moreover, Table. IV shows that all five LLMs perform moderately when provided with only the raw issues, which more closely resembles their training data format [75]. However, after incorporating the knowledge provided by COCA, all models exhibit consistent and significant improvements in performance.

TABLE IV: The performance of COCA with different backbone models.

System	Framework	Root Cause Summarization					Root Cause Localization		
		BLEU-4	ROUGE-1	METEOR	Semantics	Usefulness	Exact Match	Top-3	Top-5
GPT-4o	Base	0.151	0.455	0.354	0.789	0.648	0.302	0.547	0.745
	COCA	0.205	0.519	0.438	0.880	0.776	0.472	0.774	0.915
	Δ	↑ 35.8%	↑ 14.1%	↑ 23.7%	↑ 11.5%	↑ 19.8%	↑ 56.3%	↑ 41.4%	↑ 22.8%
GPT-3.5	Base	0.124	0.408	0.309	0.755	0.541	0.264	0.472	0.632
	COCA	0.157	0.461	0.361	0.802	0.592	0.368	0.557	0.764
	Δ	↑ 26.6%	↑ 13.0%	↑ 16.8%	↑ 6.22%	↑ 9.4%	↑ 39.4%	↑ 18.0%	↑ 20.9%
LLaMa-3.1-405b	Base	0.150	0.456	0.348	0.764	0.626	0.283	0.557	0.811
	COCA	0.189	0.514	0.427	0.838	0.702	0.434	0.745	0.877
	Δ	↑ 26.0%	↑ 12.7%	↑ 22.7%	↑ 9.7%	↑ 12.1%	↑ 53.3%	↑ 33.9%	↑ 8.1%
Claude-3.5-Sonnet	Base	0.107	0.441	0.337	0.800	0.652	0.349	0.660	0.868
	COCA	0.148	0.470	0.419	0.854	0.744	0.453	0.764	0.943
	Δ	↑ 38.3%	↑ 6.6%	↑ 24.3%	↑ 6.8%	↑ 14.1%	↑ 29.7%	↑ 15.7%	↑ 8.7%
Gemini-1.5-Pro	Base	0.161	0.460	0.361	0.794	0.633	0.321	0.528	0.764
	COCA	0.224	0.523	0.408	0.859	0.737	0.443	0.736	0.887
	Δ	↑ 39.1%	↑ 13.7%	↑ 13.3%	↑ 8.2%	↑ 16.4%	↑ 38.0%	↑ 39.4%	↑ 16.1%

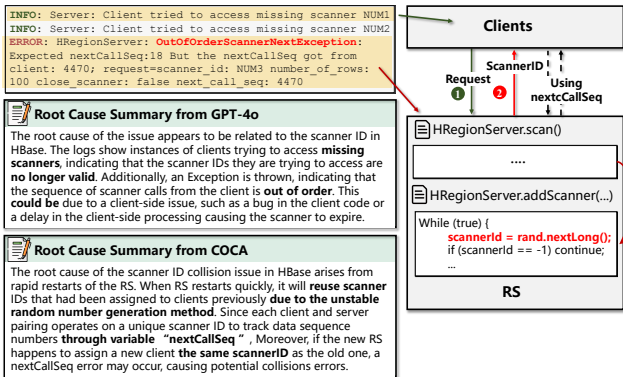


Fig. 7: Case study of HBASE-9821<sup>4</sup>.

**Answer to RQ3.** COCA consistently enhances the performance of both root cause summarization and localization, regardless of the type of backbone models employed. Moreover, our experiments demonstrate the limited data leakage, further highlighting the generalizability of COCA.

## VI. DISCUSSION

### A. Case Study

To demonstrate the practical value of COCA for system maintainers, we analyze HBASE-982<sup>4</sup> as a practical example. This case illustrates how COCA significantly improves the efficiency and accuracy of root cause analysis for system maintainers. The issue involves scanner ID collisions in the HBASE system, where multiple clients attempt to access non-existent scanner IDs, resulting in *nextCallSeq* mismatch errors.

While GPT-4o provides a surface-level diagnosis based solely on the issue report and logs, identifying only that the scanner IDs are invalid and clients are unaware of this condition, COCA delivers substantially more actionable insights for maintainers by leveraging code knowledge extracted from logs and traces. COCA determines the root cause of the issue is

the reuse of scanner IDs, then concludes that scanner IDs are generated through random numbers by reviewing the code. When an HRegionServer (RS) quickly restarts, it is treated as a new RS and is assigned the same scanner ID as before due to the unreliable random number generation logic without dynamic seeds. Consequently, this leads to inconsistencies of scanner IDs for clients, resulting in a system crash.

Compared to solely relying on issue reports, the integration of code knowledge can substantially enhance the information available to the LLM prior to making an inference, thereby generating a more comprehensive and detailed RCA result.

To demonstrate COCA’s benefits for maintainers, we conducted a controlled experiment of this case with an experienced annotator, who is also the contributor to HBase. When presented with GPT-4o’s RCA summarization, the annotator spent over 3 hours analyzing the system’s execution path and related code based on the traces and logs available in the issue report, which COCA had already automatically summarized. Notably, this issue took a full day of active discussion on the JIRA forum before reaching the conclusion. This real-world example demonstrates how COCA accelerates the diagnosis process by automatically reconstructing and analyzing system execution paths with LLM that would otherwise require hours of manual investigation by skilled maintainers.

### B. Practicality of COCA

COCA is designed to facilitate the diagnosis of issues and incidents [14], [24] in real-world distributed systems, such as modern cloud systems (e.g., Azure and AWS). These mature systems have employed issue-tracking platforms (such as ICM [24] in Microsoft, JIRA used by Apache) to streamline issue management and enhance system reliability.

When an incident or issue is reported, COCA will outperform current methods in three aspects. In terms of **effectiveness**, our extensive experiments on real-world JIRA issues demonstrate COCA’s strong performance in production environments. The evaluation results show high accuracy in root cause analysis, consistent performance across different

<sup>4</sup><https://issues.apache.org/jira/browse/HBASE-9821>

types of issues, which is the strong evidence of COCA’s practical capability in real-world systems. Additionally, COCA’s **extensibility** allows for seamless adaptation to various issue tracking systems without the need to reproduce the issue. These platforms share similar settings with user-reported issues, including descriptions, logs, and traces. Besides, for **efficiency**, it typically takes maintainers several hours to days (based on average response time from JIRA issue maintenance records). However, COCA provides RCA results with an average response time of 19.4 seconds while automatically identifying and analyzing an average of 26.9 methods from execution paths in hundreds of lines of logs and stack traces.

### C. Threats to Validity

**Potential Data Leakage.** One of the main concerns of this work is the potential data leakage issue due to the utilization of public issue reports. Specifically, the target issue report may be trained during the pretraining or retrieved by our sampling algorithm, leading it to memorize rather than infer [32], [40], [75] the result. However, this risk may be mitigated by several factors. Firstly, many of the root causes are not available on JIRA, which reduces the chances of data leakage. Secondly, our complex prompt format is unlikely to appear directly in the training dataset, especially compared to raw issue reports, as the experiment results shown in Table IV for 5 representative LLMs. Thirdly, as shown in Table III, the performance improves significantly after feeding the raw JIRA issue discussions, demonstrating that those LLMs have limited memorization of RCA results. Moreover, during the dataset collection phase, we manually reviewed each issue’s discussion and linking relationships to ensure that there were no duplicate issue reports with the same root cause.

**The Precision of Executed Path Reconstruction.** Static analysis has inherent limitations in its analysis boundary [76]–[78] and struggles with dynamic bindings [52]–[54]. This affects its accuracy in constructing call graphs, especially when dealing with network interactions and the use of reflection, proxies and multi-threading in distributed systems. To mitigate this, we propose and implement a novel RPC bridging method to integrate RPCs within COCA to enhance its analytical capabilities. In addition, ablation studies also show that compared with similarity-based code base retrieval methods, reconstructing execution paths can maximize the introduction of fault-related code fragments and restore the system execution state as much as possible.

## VII. RELATED WORK

**Monitor Data-driven RCA.** Most of the existing RCA approaches [2], [4]–[6], [38], [68], [79]–[81] only use monitor data (e.g., logs or traces) as input. They typically compare whether the runtime data pattern changes under normal and abnormal system states to figure out the root cause. For instance, LogCluster [4] utilizes a clustering algorithm to collect log clusters, which are then employed to discern log abnormal patterns by comparing them with test log sequences.

NeZha [2] contrasts event graphs from fault-free and fault-affected systems to deduce the root causes. However, this type of approach has three limitations: (1) they rely on the normal runtime data pattern of the system, which is not readily available in practice. (2) they usually only output the root cause module and lack a root cause summarization. (3) they do not incorporate related code as contextual information, leading to underperforming diagnostic results.

**LLM-based RCA.** The recent advent of LLMs has facilitated the proposal of several LLM-based RCA methodologies [10], [14], [16], [82], since LLMs inherently possess domain knowledge of systems and corresponding failures. RCACopilot [14], which was introduced as an LLM-based RCA approach equipped with retrieval tools. By analyzing historical incidents, ReAct [16] can understand complex system issues, thereby continuously enhancing its performance.

Compared to existing LLM-based RCA approaches, COCA extends its diagnostic capabilities beyond runtime information and historical bug reports by incorporating code knowledge. COCA can retrieve and analyze internal system knowledge (i.e., code) without requiring issue reproduction to obtain code coverage during execution, which fault localization methods did [10] but is impractical in distributed systems [13], [29]. From the perspective of fault types, COCA address a variety of issues, including both code bugs and non-code problems (e.g., resource limitations, misconfigurations, or environmental factors) by providing code knowledge to enhance the comprehension of the system, thereby facilitating the RCA process, while fault localization methods focus solely on localizing reproducible code bugs. Regarding analysis scope, COCA performs root cause analysis by summarizing and localizing runtime problems. In contrast, fault localization techniques are restricted to localizing faults within the code itself, without addressing non-code or environmental factors. Thus, COCA overcomes the limitations of previous LLM-based approaches by providing a more thorough and contextualized RCA method.

## VIII. CONCLUSION

In this paper, we propose COCA, the first work incorporating code knowledge into an automatic root cause analysis framework for issue reports in distributed systems. Experimental results show that COCA outperforms all baselines and can be generalized to various LLMs. COCA demonstrates its promising future in the context of the evolving LLM techniques, elevating the upper limit of LLM’s capability in RCA task and benefiting both researchers and maintainers.

## ACKNOWLEDGMENT

The work described in this paper was supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14206921 of the General Research Fund), and RGC Grant for Theme-based Research Scheme Project (RGC Ref. No. T43-513/23-N), and the National Nature Science Foundation of China under Grant (No. 62402536).

## REFERENCES

- [1] Apache, “Jira,” Mar 2024. [Online]. Available: <https://issues.apache.org/jira/secure/Dashboard.jspa>
- [2] G. Yu, P. Chen, Y. Li, H. Chen, X. Li, and Z. Zheng, “Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 553–565.
- [3] X. Li, P. Chen, L. Jing, Z. He, and G. Yu, “Swisslog: Robust and unified deep learning based log anomaly detection for diverse faults,” in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 92–103.
- [4] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, “Log clustering based problem identification for online service systems,” in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 102–111.
- [5] C. M. Rosenberg and L. Moonen, “Spectrum-based log diagnosis,” in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–12.
- [6] F. Lin, K. Muzumdar, N. P. Laptev, M.-V. Curelea, S. Lee, and S. Sankar, “Fast dimensional analysis for root cause investigation in a large-scale service environment,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 2, pp. 1–23, 2020.
- [7] T. Hirsch and B. Hofer, “Root cause prediction based on bug reports,” in *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2020, pp. 171–176.
- [8] J. Huang, Z. Jiang, J. Liu, Y. Huo, J. Gu, Z. Chen, C. Feng, H. Dong, Z. Yang, and M. R. Lyu, “Demystifying and extracting fault-indicating information from logs for failure diagnosis,” in *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2024, pp. 511–522.
- [9] W. Yuan, S. Lu, H. Sun, and X. Liu, “How are distributed bugs diagnosed and fixed through system logs?” *Information and Software Technology*, vol. 119, p. 106234, 2020.
- [10] S. KANG, G. AN, and S. YOO, “A quantitative and qualitative evaluation of llm-based explainable fault localization,” *FSE*, 2024.
- [11] S. Kang, B. Chen, S. Yoo, and J.-G. Lou, “Explainable automated debugging via large language model-driven scientific debugging,” *arXiv preprint arXiv:2304.02195*, 2023.
- [12] R. Just, D. Jalali, and M. D. Ernst, “Defects4j: A database of existing faults to enable controlled testing studies for java programs,” in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440.
- [13] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, “Simple testing can prevent most critical failures: An analysis of production failures in distributed {Data-Intensive} systems,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 249–265.
- [14] Y. Chen, H. Xie, M. Ma, Y. Kang, X. Gao, L. Shi, Y. Cao, X. Gao, H. Fan, M. Wen *et al.*, “Automatic root cause analysis via large language models for cloud incidents,” in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 674–688.
- [15] K. An, F. Yang, L. Li, Z. Ren, H. Huang, L. Wang, P. Zhao, Y. Kang, H. Ding, Q. Lin *et al.*, “Nissist: An incident mitigation copilot based on troubleshooting guides,” *arXiv preprint arXiv:2402.17531*, 2024.
- [16] D. Roy, X. Zhang, R. Bhave, C. Bansal, P. Las-Casas, R. Fonseca, and S. Rajmohan, “Exploring llm-based agents for root cause analysis,” *arXiv preprint arXiv:2403.04123*, 2024.
- [17] T. Ahmed, S. Ghosh, C. Bansal, T. Zimmermann, X. Zhang, and S. Rajmohan, “Recommending root-cause and mitigation steps for cloud incidents using large language models,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1737–1749.
- [18] Z. Jiang, J. Liu, J. Huang, Y. Li, Y. Huo, J. Gu, Z. Chen, J. Zhu, and M. R. Lyu, “A large-scale benchmark for log parsing,” *arXiv preprint arXiv:2308.10828*, 2023.
- [19] V. Bushong, R. Sanders, J. Curtis, M. Du, T. Cerny, K. Frajtak, M. Bures, P. Tisnovsky, and D. Shin, “On matching log analysis to source code: A systematic mapping study,” in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, 2020, pp. 181–187.
- [20] Y. Peng, C. Wang, W. Wang, C. Gao, and M. R. Lyu, “Generative type inference for python,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 988–999.
- [21] Y. Ding, Z. Wang, W. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth *et al.*, “Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.
- [22] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: a method for automatic evaluation of machine translation,” in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics (ACL)*, 2002, pp. 311–318.
- [23] meta, “Llama 3.1,” 2024. [Online]. Available: <https://ai.meta.com/blog/meta-llama-3-1/>
- [24] Z. Chen, Y. Kang, L. Li, X. Zhang, H. Zhang, H. Xu, Y. Zhou, L. Yang, J. Sun, Z. Xu *et al.*, “Towards intelligent incident management: why we need it and how we make it,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 1487–1497.
- [25] J. Liu, S. He, Z. Chen, L. Li, Y. Kang, X. Zhang, P. He, H. Zhang, Q. Lin, Z. Xu *et al.*, “Incident-aware duplicate ticket aggregation for cloud systems,” *arXiv preprint arXiv:2302.09520*, 2023.
- [26] Y. Huo, Y. Li, Y. Su, P. He, Z. Xie, and M. R. Lyu, “Autolog: A log sequence synthesis framework for anomaly detection,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 497–509.
- [27] B. Chen, J. Song, P. Xu, X. Hu, and Z. M. Jiang, “An automated approach to estimating code coverage measures via execution logs,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 305–316.
- [28] J. Liu, J. Huang, Y. Huo, Z. Jiang, J. Gu, Z. Chen, C. Feng, M. Yan, and M. R. Lyu, “Scalable and adaptive log-based anomaly detection with expert in the loop,” *arXiv preprint arXiv:2306.05032*, 2023.
- [29] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, “Sherlog: error diagnosis by connecting clues from run-time logs,” in *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, 2010, pp. 143–154.
- [30] J. Huang, J. Liu, Z. Chen, Z. Jiang, Y. Li, J. Gu, C. Feng, Z. Yang, Y. Yang, and M. R. Lyu, “Faultprofit: Hierarchical fault profiling of incident tickets in large-scale cloud systems,” in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, 2024, pp. 392–404.
- [31] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [32] Z. Yang, Z. Zhao, C. Wang, J. Shi, D. Kim, D. Han, and D. Lo, “What do code models memorize? an empirical study on large language models of code,” *arXiv preprint arXiv:2308.09932*, 2023.
- [33] S. Gao, X.-C. Wen, C. Gao, W. Wang, H. Zhang, and M. R. Lyu, “What makes good in-context demonstrations for code intelligence tasks with llms?” in *Proceedings of the 38th International Conference on Automated Software Engineering (ASE)*, 2023.
- [34] S. He, J. Zhu, P. He, and M. R. Lyu, “Loghub: a large collection of system log datasets towards automated log analytics,” *arXiv preprint arXiv:2008.06448*, 2020.
- [35] Z. Jiang, J. Liu, J. Huang, Y. Li, Y. Huo, J. Gu, Z. Chen, J. Zhu, and M. R. Lyu, “A large-scale evaluation for log parsing techniques: How far are we?” in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024.
- [36] X. Wang, X. Zhang, L. Li, S. He, H. Zhang, Y. Liu, L. Zheng, Y. Kang, Q. Lin, Y. Dang *et al.*, “Spine: a scalable log parser with feedback guidance,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1198–1208.
- [37] D. Schipper, M. Aniche, and A. van Deursen, “Tracing back log data to its log statement: from research to practice,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 545–549.
- [38] Y. Li, X. Zhang, S. He, Z. Chen, Y. Kang, J. Liu, L. Li, Y. Dang, F. Gao, Z. Xu *et al.*, “An intelligent framework for timely, accurate, and comprehensive cloud incident detection,” *ACM SIGOPS Operating Systems Review*, vol. 56, no. 1, pp. 1–7, 2022.
- [39] Y. Li, Y. Peng, Y. Huo, and M. R. Lyu, “Enhancing llm-based coding tools through native integration of ide-derived static context,” *arXiv preprint arXiv:2402.03630*, 2024.

- [40] Y. Li, Y. Huo, Z. Jiang, R. Zhong, P. He, Y. Su, and M. R. Lyu, "Exploring the effectiveness of llms in automated logging generation: An empirical study," *arXiv preprint arXiv:2307.05950*, 2023.
- [41] Z. Jiang, J. Liu, Z. Chen, Y. Li, J. Huang, Y. Huo, P. He, J. Gu, and M. R. Lyu, "Lilac: Log parsing using llms with adaptive parsing cache," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 137–160, 2024.
- [42] J. Huang, Z. Jiang, Z. Chen, and M. R. Lyu, "Ulog: Unsupervised log parsing with large language models through log contrastive units," *arXiv preprint arXiv:2406.07174*, 2024.
- [43] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *2017 IEEE international conference on web services (ICWS)*. IEEE, 2017, pp. 33–40.
- [44] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm, "lprof: A non-intrusive request flow profiler for distributed systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 629–644.
- [45] T. Jia, L. Yang, P. Chen, Y. Li, F. Meng, and J. Xu, "Logged: Anomaly diagnosis through mining time-weighted control flow graph in logs," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. IEEE, 2017, pp. 447–455.
- [46] Y. Li, Y. Huo, R. Zhong, Z. Jiang, J. Liu, J. Huang, J. Gu, P. He, and M. R. Lyu, "Go static: Contextualized logging statement generation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 609–630, 2024.
- [47] Y. Wu, Z. Yu, M. Wen, Q. Li, D. Zou, and H. Jin, "Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1046–1058.
- [48] Z. Zhang, M. K. Ramanathan, P. Raj, A. Parwal, T. Sherwood, and M. Chabbi, "CRISP: critical path analysis of large-scale microservice architectures," in *Proceedings of the 2022 USENIX Annual Technical Conference*, J. Schindler and N. Zilberman, Eds., 2022, pp. 655–672.
- [49] A. Fang, R. Zhou, X. Tang, and P. He, "Rpcover: Recovering grpc dependency in multilingual projects," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1930–1939.
- [50] gRPC, "grpc," Jul 2024. [Online]. Available: <https://grpc.io/>
- [51] Thrift, "Thrift," Jul 2024. [Online]. Available: <https://thrift.apache.org/>
- [52] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 2, pp. 7:1–7:50, 2019.
- [53] N. Grech and Y. Smaragdakis, "P'taint: unified points-to and taint analysis," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, 2017.
- [54] B. R. Bruce, T. Zhang, J. Arora, G. H. Xu, and M. Kim, "Jshrink: in-depth investigation into debloating modern java applications," in *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 135–146.
- [55] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.
- [56] D. Nam, A. Macvean, V. J. Hellendoorn, B. Vasilescu, and B. A. Myers, "Using an LLM to help with code understanding," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 97:1–97:13.
- [57] Y. Wang, H. Le, A. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, "Codet5+: Open code large language models for code understanding and generation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds., 2023, pp. 1069–1088.
- [58] Z. Wang, Z. Liu, Y. Zhang, A. Zhong, L. Fan, L. Wu, and Q. Wen, "Rcagent: Cloud root cause analysis by autonomous agents with tool-augmented large language models," *arXiv preprint arXiv:2310.16340*, 2023.
- [59] S. Robertson, H. Zaragoza, and M. Taylor, "Simple bm25 extension to multiple weighted fields," in *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, 2004, pp. 42–49.
- [60] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, "Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems," in *Proceedings of the twenty-first international conference on architectural support for programming languages and operating systems*, 2016, pp. 517–530.
- [61] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-Anake, T. Do, H. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin *et al.*, "What bugs live in the cloud? a study of 3000+ issues in cloud systems," in *Proceedings of the ACM symposium on cloud computing*, 2014, pp. 1–14.
- [62] Z. Li, J. Chen, R. Jiao, N. Zhao, Z. Wang, S. Zhang, Y. Wu, L. Jiang, L. Yan, Z. Wang *et al.*, "Practical root cause localization for microservice systems via trace analysis," in *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. IEEE, 2021, pp. 1–10.
- [63] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005, pp. 65–72.
- [64] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.
- [65] OpenAI. Openai embeddings. [Online]. Available: <https://platform.openai.com/docs/guides/embeddings>
- [66] H. Wang, X. Xia, D. Lo, J. Grundy, and X. Wang, "Automatic solution summarization for crash bugs," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1286–1297.
- [67] A. Saha and S. C. Hoi, "Mining root cause knowledge from cloud service incident investigations for aiops," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, 2022, pp. 197–206.
- [68] C. Lee, T. Yang, Z. Chen, Y. Su, and M. R. Lyu, "Eadro: An end-to-end troubleshooting framework for microservices on multi-source data," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1750–1762.
- [69] E. Foundation, "Eclipse java development tools (jdt) core," 2024. [Online]. Available: <https://www.eclipse.org/jdt/core/>
- [70] OpenAI. (2022) Introducing chatgpt. [Online]. Available: <https://openai.com/blog/chatgpt>
- [71] G. DeepMind, "Gemini 1.5 pro," 2024. [Online]. Available: <https://deepmind.google/technologies/gemini/pro/>
- [72] Anthropic, "Anthropic," 2024. [Online]. Available: <https://www.anthropic.com/>
- [73] Deepinfra, "Deepinfra," 2024. [Online]. Available: <https://deepinfra.com/>
- [74] J. Xu, R. Yang, Y. Huo, C. Zhang, and P. He, "Prompting for automatic log template extraction," *arXiv preprint arXiv:2307.09950*, 2023.
- [75] Y. Huang, Y. Li, W. Wu, J. Zhang, and M. R. Lyu, "Do not give away my secrets: Uncovering the privacy issue of neural code completion tools," *arXiv preprint arXiv:2309.07639*, 2023.
- [76] W. Li, J. Ming, X. Luo, and H. Cai, "Polycruise: A cross-language dynamic information flow analysis," in *31st USENIX Security Symposium*, 2022, pp. 2513–2530.
- [77] J. Wang and H. Wang, "Nativesummary: Summarizing native binary code for inter-language static analysis of android apps," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024.
- [78] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyandé, and J. Klein, "Jucify: A step towards android code unification for enhanced static analysis," in *44th IEEE/ACM 44th International Conference on Software Engineering*, 2022, pp. 1232–1244.
- [79] P. Chen, Y. Qi, P. Zheng, and D. Hou, "Causeinfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems," in *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 2014, pp. 1887–1895.
- [80] A. Amar and P. C. Rigby, "Mining historical test logs to predict bugs and localize faults in the test logs," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 140–151.
- [81] Z. Jiang, J. Huang, Z. Chen, Y. Li, G. Yu, C. Feng, Y. Yang, Z. Yang, and M. R. Lyu, "L4: Diagnosing large-scale llm training failures via automated log analysis," *arXiv preprint arXiv:2503.20263*, 2025.
- [82] S. Shan, Y. Huo, Y. Su, Y. Li, D. Li, and Z. Zheng, "Face it yourselves: An llm-based two-stage strategy to localize configuration errors via logs," *arXiv preprint arXiv:2404.00640*, 2024.