LLMRCA: Multilevel Root Cause Analysis for LLM Applications Using Multimodal Observability Data

GOU TAN, School of Systems Science and Engineering, Sun Yat-sen University, China
ZILONG HE, School of Computer Science and Engineering, Sun Yat-sen University, China
MIN LI, School of Systems Science and Engineering, Sun Yat-sen University, China
HAIYU HUANG, School of Computer Science and Engineering, Sun Yat-sen University, China
YILUN WANG, School of Systems Science and Engineering, Sun Yat-sen University, China
PENGFEI CHEN*, School of Computer Science and Engineering, Sun Yat-sen University, China
GIULIANO CASALE, Department of Computing, Imperial College London, UK
CHUANFU ZHANG, School of Systems Science and Engineering, Sun Yat-sen University, China

Ensuring the reliability of large language model (LLM) applications in production environments is critical as LLMs are widely integrated into software systems. However, fault diagnosis in these applications is challenging due to distributed deployment and complex component interactions. Existing root cause analysis (RCA) methods fall short when applied to LLM applications because they ignore two key challenges: unstable response times and response quality-related silent faults. To address these challenges, we propose LLMRCA, the first multilevel and unsupervised RCA framework that leverages multimodal observability data to identify root causes in LLM applications. LLMRCA constructs a heterogeneous causal graph from metrics, logs, and traces, and employs a Residual Graph Attention autoencoder to calculate reconstruction errors for RCA. LLMRCA also introduces request classification to handle unstable response times and verification to reduce false positives. Experiments on a retrieval-augmented generation LLM application show that LLMRCA outperforms eight baseline methods, achieving up to 5.1× more accurate results than baselines for performance problems and 92.86% top-1 ranking accuracy for response-quality problems. Furthermore, the ablation studies confirm the contributions of each data modality and different framework modules to effectiveness. Generally, our framework provides a practical approach for diagnosing both performance and quality faults in LLM applications.

CCS Concepts: • **Software and its engineering → Software reliability**; **Software performance**.

Additional Key Words and Phrases: Large Language Models, Root Cause Analysis, Multimodal Observability Data, Graph Neural Network

*Pengfei Chen is the corresponding author.

Authors' Contact Information: Gou Tan, tang29@mail2.sysu.edu.cn, School of Systems Science and Engineering, Sun Yat-sen University, Guangzhou, China; Zilong He, hezlong@mail2.sysu.edu.cn, School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China; Min Li, limin258@mail2.sysu.edu.cn, School of Systems Science and Engineering, Sun Yat-sen University, Guangzhou, China; Haiyu Huang, huanghy95@mail2.sysu.edu.cn, School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China; Yilun Wang, wangylun6@mail2.sysu.edu.cn, School of Systems Science and Engineering, Sun Yat-sen University, Guangzhou, China; Pengfei Chen, chenpf7@mail.sysu.edu.cn, School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China; Giuliano Casale, g.casale@imperial.ac.uk, Department of Computing, Imperial College London, London, UK; Chuanfu Zhang, zhangcf9@mail.sysu.edu.cn, School of Systems Science and Engineering, Sun Yat-sen University, Guangzhou, China.

ACM 1557-7392/2025/8-ART
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Gou Tan, Zilong He, Min Li, Haiyu Huang, Yilun Wang, Pengfei Chen, Giuliano Casale, and Chuanfu Zhang. 2025. LLMRCA: Multilevel Root Cause Analysis for LLM Applications Using Multimodal Observability Data. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (August 2025), 41 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Since the release of ChatGPT in 2022, large language models (LLMs) have shown a promising future. Various frameworks, such as LlamaIndex [23] and LangChain [20], simplify the development process by removing developers of the management of LLM models, knowledge bases, and other tools [15]. As a result, a large number of LLM applications have been developed recently [40], which could contribute an additional $2.6 to $4.4 trillion annually to the global economy [41].

However, in production environments, these LLM applications are often deployed across different hosts and rely on complex software components, such as Pytorch [1], CUDA Toolkit [28], and Transformers [47]. Factors such as dynamic resource allocations, complex service invocations, and stochastic request arrival patterns can affect application performance ($\mathcal{AP}$), as is typically manifested in response time and error rate metrics [55]. Moreover, novel factors that arise in connection with LLM technology, such as model precision and hallucinations, can instead significantly impact the response quality ($\mathcal{RQ}$), i.e., the experience quality is decreased from the point of view of the end user, ultimately affecting business performance [42]. Such technical and business performance degradations can lead to poor user experience and financial losses [29, 37, 41, 50, 51]. Therefore, it is important to take into account *both* $\mathcal{AP}$ and $\mathcal{RQ}$ to ensure LLM application quality.

To ensure the reliability and robustness of applications, the *observability* paradigm is widely adopted, which seeks to provide continuous and comprehensive visibility into the execution of the application [39]. For LLM applications, the observability data available to engineers to assess these performance dimensions typically includes metrics, logs, and traces [51] that mix both traditional services and LLM-specific components, such as the software services responsible for retrieval augmented generation (RAG), as shown in Fig. 1. Access to heterogeneous observability data can assist people in performing *root cause analysis* (RCA) to diagnose faults at multiple levels of LLM applications. However, it is always a process that is manual, time-consuming and error-prone. Thus, the research can support this activity by defining novel automated methods to analyze LLM application data automatically, going beyond canonical approaches through the integration of $\mathcal{RQ}$ metrics in the RCA phase. This paper addresses the software engineering problem of root cause analysis for complex LLM applications, proposing a framework to enhance their operational reliability.

With this goal in mind, traditional observability approaches for canonical software systems are incomplete when applied to LLM applications as they only focus on $\mathcal{AP}$ aspects such as tracing the performance of LLM generation services [2, 43] and monitoring the underpinning resources and deployment environment [17] (e.g., using solutions such as DCGM Exporter [7], Prometheus Metric Exporter [33], or OpenTelemetry Collector [31]). At present, existing methods therefore do not analyze *both* $\mathcal{AP}$ and $\mathcal{RQ}$ in the process of seeking root causes of faults and performance degradations, which is an important limitation that we seek to address in this paper.

Traditional RCA approaches face key challenges when applied to LLM applications. These challenges include the dual nature of faults affecting both performance and quality, the inherent instability of response times, and the lack of tailored RCA methods capable of analyzing semantic context. We discuss these challenges in detail in Section 3.2.

To overcome these challenges, we propose LLMRCA, the first multilevel and unsupervised RCA framework designed for LLM applications through multimodal data analysis. LLMRCA builds on traditional RCA approaches but introduces key innovations to address LLM-specific challenges:

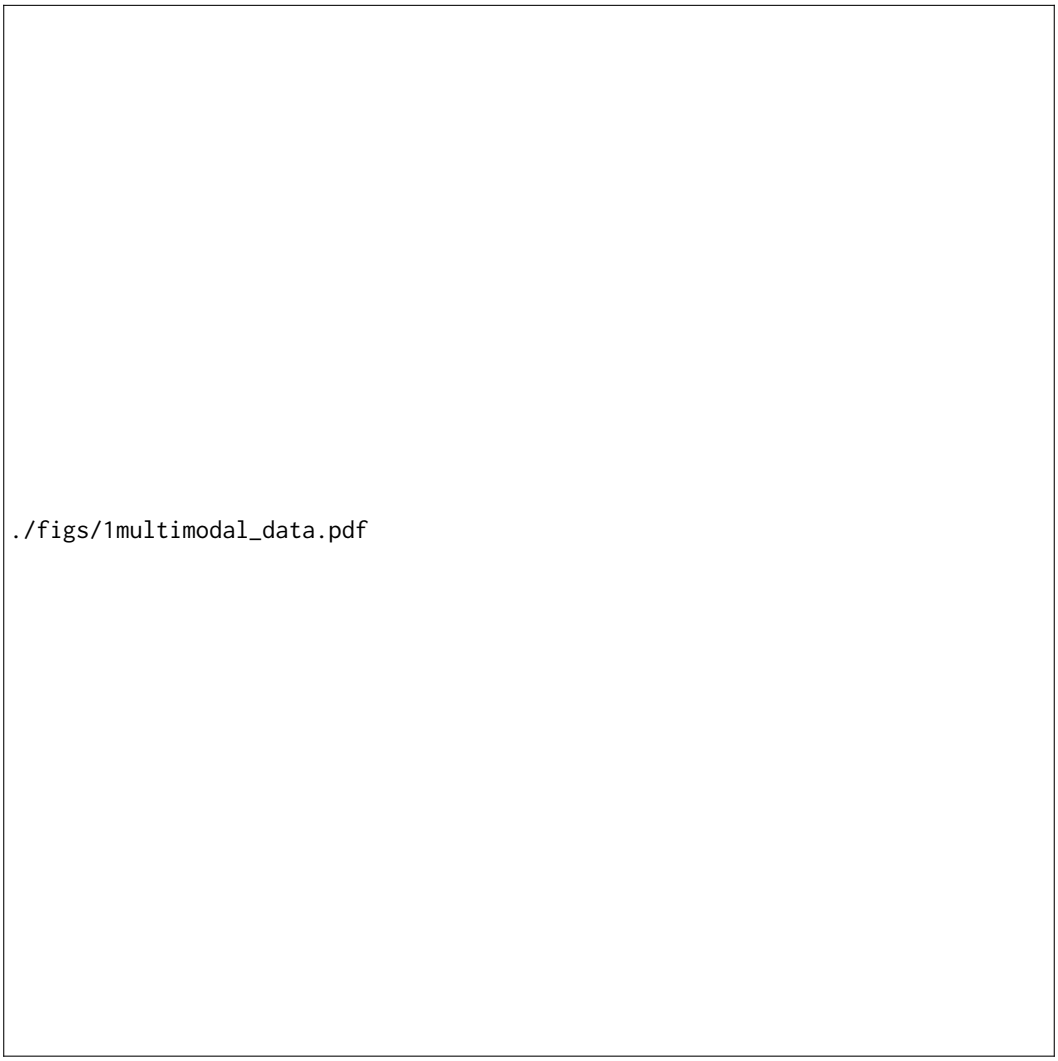


Fig. 1. RCA at multiple levels with different observability data.

(1) it enriches observability data with LLM-specific context (Fig. 8), and constructs a personalized heterogeneous causal graph that enables LLMRCA to diagnose novel performance and quality faults, (2) it classifies requests based on fitted response time (a regression-predicted theoretical normal response time) to address the instability issue, and (3) it uses advanced graph neural networks with verification principles to improve fault localization accuracy. During training, the response time of each normal request is used to fit an XGBoost regression model, and then we classify these requests into categories based on the fitted response time. The class labels are encoded using a one-hot representation and are integrated as additional node attributes into the causal graph to train the Residual Graph Attention (ResGAT) autoencoder model. When faults occur, LLMRCA classifies requests, uses the trained ResGAT model to calculate reconstruction errors, and incorporates a hierarchical verifier to identify the root cause.

Table 1. Multilevel faults in LLM applications [40, 53].

Level	Faults	Fault Description	Possible Causes
Application	Low-Quality Responses	Hallucinations, incorrect or irrelevant answers.	Model limitations, unclear prompts, poor context, outdated knowledge base.
	Response Latency	High delays in response time.	Computational bottlenecks (CPU/GPU), network latency, large model size, complex queries.
Component	Component Unreachable	Instances or components inaccessible.	Infrastructure fault, service crashes, overload, network issues.
	Request Timeout	Requests take too long and fail.	Network congestion, slow backend processing, service overload.
Code	Format Mismatch	LLM Input or output format mismatch.	Incorrect data preprocessing, missing format validation, incompatibilities with external APIs.
	Bugs in Code	Functional errors in code execution.	Logic issues, unhandled exceptions, incorrect algorithm implementation.
	Exceeding Context Limit	Input exceeds the model's token limit.	Too much data in the request, improper truncation of input, model constraints.
Host	Resource Exhaustion	System resources depleted (CPU, GPU, memory).	High demand, poor resource management, resource contention.
	Network Issues	Connectivity issues affecting service access.	Network instability, misconfiguration, infrastructure issues.
Context	Unclear Prompt Context	Insufficient prompt information leading to inaccurate responses.	Poor prompt design, missing details, incomplete user inputs.
	Improper Retrieval Results	Irrelevant or inaccurate search results.	Poor retrieval algorithms, inaccurate embeddings, outdated indexing.
	Knowledge Misalignment	Misaligned, outdated, or incorrect knowledge.	Knowledge base errors, missing updates in external data sources.
	Low-Quality Embeddings	Embeddings fail to accurately represent the input.	Suboptimal embedding model, lack of domain-specific optimization.

We perform extensive studies to evaluate LLMRCA in an RAG-enhanced LLM application. The results demonstrate that LLMRCA significantly outperforms eight baseline RCA methods. For $\mathcal{AP}$, at the inner-component level, LLMRCA improves HR@1 (ranking accuracy of top-1) by up to 5.1× over baselines; for $\mathcal{RQ}$, it achieves HR@1 of 92.86% and NDCG@3 (ranking quality of top-3 results) of 97.36% for single-type requests. Moreover, ablation studies confirm the critical contributions of different modalities of data, and key modules (e.g., classifier, hierarchical verifier, ResGAT).

Contributions. In summary, the main contributions of this paper are listed as follows.

- We present LLMRCA, the first multilevel and unsupervised RCA framework designed for LLM applications. LLMRCA addresses both application performance and response quality faults by constructing a heterogeneous causal graph with LLM-specific contextual data.
- We address the unstable response time issue in LLM applications through regression fitting and classification to effectively reduce false positives of fault diagnosis.

- We establish an RCA benchmark for LLM applications by constructing a typical RAG-enhanced LLM application. We add personalized logs and traces to collect contextual data throughout the entire request lifecycle, enabling more comprehensive performance analysis.
- We conduct extensive experiments on the RAG-enhanced LLM application to validate the effectiveness. We compare LLMRCA with eight baselines and evaluate multilevel RCA capabilities at both the component level and the inner-component level. Ablation studies demonstrate the contributions of different modalities of data and the impact of different configurations.

The rest of the paper is organized as follows. Section 2 introduces related work. Section 3 presents background and motivation. Section 4 overviews the LLMRCA framework. Experimental results are given in Section 5. Section 6 provides a root cause analysis case and an end-to-end case study. Section 7 discusses limitations, future work, and threats to validity, followed by conclusions in Section 8. The source code of LLMRCA is available at https://github.com/IntelligentDDS/LLMRCA.

2 RELATED WORK

Metric-based RCA. Metric-based RCA approaches analyze relationships between various metrics to diagnose faults. However, when anomalies occur, indirect anomaly propagation often affects related metrics, making root cause identification challenging. Most existing methods focus on constructing causal graphs for metrics but typically provide operation-level or system-level root causes rather than code-level insights. For example, CauseInfer [5] and Microscope [21] use the PC algorithm to build causal graphs and identify root causes along causal paths. MicroRCA [49] applies the PageRank algorithm on anomaly sub-graphs to localize suspicious services. MicroDiag [48] builds causal graphs with Granger causality tests, assigns edge weights using Pearson correlation, and ranks root causes through PageRank. Similarly, MicroCause [25] uses the path condition time series algorithm for graph construction and a temporal cause-oriented random walk for root cause ranking. GROOT [44] constructs causal graphs of monitoring events and ranks root causes using a personalized PageRank algorithm. These approaches typically offer system-level analysis, limiting their ability to identify root causes within systems.

Log-based RCA. Log-based RCA approaches primarily analyze critical log events to diagnose faults. LogCluster [22] uses log clustering to extract representative log sequences and compares sequences from production and test environments to find unseen logs. LogRCA [46] uses Transformer models to assign a root cause score to each log, automatically identifying the most relevant logs for troubleshooting. However, these approaches fail to correlate logs with requests and do not fully consider the contextual information of the application.

Trace-based RCA. Trace-based RCA approaches can be categorized into three ways. The first way relies on comparing traces directly. TraceAnomaly [24] uses deep learning to model normal trace patterns offline and detect anomalous traces online. The second way uses statistical analysis. MicroRank [50] and TraceRank [52] combine PageRank and spectral analysis algorithms to detect which service instances or operations lead to latency issues. The third way uses causal graphs. MonitorRank [18] applies the personalised PageRank algorithm to analyze anomaly propagation.

Multimodal data-based RCA. Multimodal data-based RCA approaches integrate various data sources, including metrics, logs, and traces, into a unified representation for analysis. Nezha [51] constructs an event graph from multimodal data to compare execution patterns before and after faults and identify root causes. TrinityRCL [9] models multimodal data as time series entities and applies causal graph and correlation analysis to identify root causes. DiagFusion [54] encodes multimodal data into vectors and uses graph neural networks for root cause localization. FaasRCA [16] integrates multimodal data and employs a graph attention network-based autoencoder to diagnose

faults across serverless lifecycle stages. While these methods outperform single-modal approaches in accuracy and efficiency, they are not yet adapted for LLM applications, particularly in diagnosing response quality faults.

3 PRELIMINARIES

3.1 Background

LLM applications. LLM applications refer to complete software systems that integrate large language models to provide end-user services, such as chatbots, question-answering systems, and content generation platforms. LLM applications integrate various components to handle complex tasks, including prompt engineering, RAG, agents, etc. Prompt engineering involves designing prompts that guide LLMs to generate contextually relevant and accurate responses without modifying the model [38]. RAG enhances the factual accuracy of LLM generations by retrieving pre-constructed knowledge from external sources [40, 41]. Agents, incorporating multiple modules (e.g., planning, memory, tools), allow LLMs to interact with external APIs, databases, and other systems [45]. This approach enables LLMs to think and make decisions autonomously for complex tasks. Although various frameworks [20, 23] simplify the development process by relieving developers from managing components, the interactions of these components are still complex. And faults can occur on both $\mathcal{AP}$ and $\mathcal{RQ}$. As a result, it is difficult to conduct fault diagnosis and RCA for LLM applications.

Observability. Observability refers to the ability to estimate the internal states of a system by examining its external performance. Typically, observability always utilizes three types of telemetry data, namely metrics, logs, and traces for RCA [51].

Metrics are time series collected at predefined intervals and used to describe the state of systems. Generally, metrics can be classified into system-level metrics and application-level metrics. System-level metrics (e.g., CPU usage, memory usage) are used to diagnose faults caused by resource anomalies. However, they are insufficient for identifying code-level or context-level faults. Application-level metrics (e.g., average response time, request latency, error rates) help identify user-perceived service faults. However, these metrics only reflect the surface of faults and not the root causes. Further analysis of system-level metrics, logs, and traces is required to identify root causes.

Logs are textual records of operations during the runtime of an application and offer an internal view of a service instance. Logs typically consist of three fields, including timestamp, verbosity level, and raw message. Moreover, logs help gather contextual information about the application. Currently, various parsing techniques (e.g., DRAIN [12]) have been proposed to extract templates and parameters from logs. Although logs provide detailed information about individual service instances, they fail to demonstrate the whole system states across different services. Thus, log analysis on different services independently cannot characterize the behavior of the overall system.

Traces record the execution paths of user requests. The entire lifecycle of a request is represented as a trace, while the invocation and response of a component are recorded as a span. Every trace has a unique trace ID, and spans within the same trace share the same trace ID but have unique span IDs [10]. Traces contain detailed information about invocations, such as start time, caller, callee, response time, status code, and other attributes. Moreover, traces can be viewed as trees, with services as nodes and invocations as edges. Conventional traces only record coarse-grained service invocation information, limiting their contribution to RCA. However, traces are enhanced with their attributes in this paper to capture LLM-specific contextual information.

Observability of LLM applications. LLM applications face various problems, including uncertain responses, unstable performance, hallucinations, prompt hacking, etc. Uncertain responses

arise from the probabilistic nature of the models, where the same inputs produce different outputs. The unstable performance results from the reliance of LLM applications on external APIs, making them susceptible to external factors. Hallucinations refer to situations where LLMs generate incorrect or fabricated information, which may lead to the propagation of false information. Whereas, the prompt hacking occurs when users exploit prompt injection techniques, causing the LLM to generate inappropriate content.

To address these issues, observability is critical for developers to identify performance bottlenecks and optimize system operations to ensure the reliability and robustness of LLM applications. Current observability approaches can be categorized into two levels: application level and host level. At the application level, tools like Phoenix [2] and OpenLLMetry [43] use OpenTelemetry [31] to track end-to-end interactions. They collect key metrics such as latency, throughput, error rates, and the input/output of LLM APIs. At the host level, DCGM [7] collects GPU metrics like usage, memory consumption, temperature, and power usage. Prometheus [33] collects CPU usage, memory, disk I/O, and network traffic. ENOVA [17] integrates these tools to provide comprehensive observability. Although these methods collect so much data, they often lack fine-grained analysis to identify the root causes of faults.

3.2 Motivation

Although many single-modal (metric-based [5, 13, 21, 25, 44, 48, 49], log-based [22, 46], trace-based [18, 24, 50, 52]) and multimodal [9, 16, 51, 54] RCA approaches have been proposed to localize root causes in canonical software systems, often based on microservice-based architecture, applying these methods in LLM applications faces several key challenges.

- **Two aspects of LLM application faults.** As shown in Table 1, beyond traditional performance faults (server crashes, network delays), LLM applications introduce unique response quality faults (hallucinations, incorrect knowledge, irrelevant retrievals). The study in [40] shows nearly 46.9% of LLM application faults are directly related to response quality (§3.2.2). Yet existing RCA methods focus primarily on application performance rather than response quality. These two aspects require a comprehensive framework that can analyze both performance and quality faults and track fault propagation across multiple levels.
- **Unstable response times in LLM applications.** LLM response times are highly correlated with the number of generated tokens and affected by interactions between multiple components, such as retrievers (embedding computation, similarity matching), rerankers, and generators (scheduling, prefill, decoding) (§3.2.1). This introduces significant instability where the same requests may have completely different normal response time distributions, making it difficult to compare response times across different request types and detect anomalies.
- **Lack of RCA methods and benchmarks for LLM applications.** Current methods focus on LLM training and output explanations while neglecting production-environment fault diagnosis. LLM applications face multilevel faults but lack three critical components: (1) standardized observation data collection mechanisms that capture LLM-specific contextual data (prompts, embeddings, retrieved documents); (2) context-aware causal analysis methods that can process heterogeneous data types; (3) reliable evaluation benchmarks with comprehensive fault scenarios.

In this section, we analyze the performance and quality challenges in LLM applications through empirical studies and introduce our motivations. Our experiments are conducted using the benchmark described in section 5.1.

Fig. 2. Response time distribution.

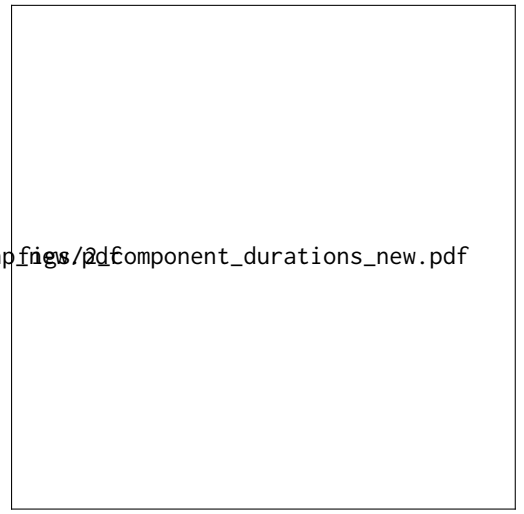


Fig. 3. Component response time.

3.2.1 Application Performance Faults in LLM Applications. **Unstable response time can lead to false positives.** The stochastic nature of LLM generation creates significant instability in the application response time. As shown in Fig. 2, through experiments on 100 different request types (§5.1 benchmark), we observe significant fluctuations in response time. Notably, the distributions of normal and anomalous requests overlap, and anomalies are caused by CPU resource contention. This overlap poses a significant challenge for traditional diagnostic methods that heavily rely on latency, throughput, or QPS metrics and potentially leads to false positives when assessing requests within these overlapping regions. For example, a normal request generating many tokens naturally takes 4s, while an anomalous request that should take only 2s (due to fewer tokens) also takes 4s because of CPU resource contention. Since both requests have the same response times, the latency cannot distinguish them. This instability presents a critical challenge: how to distinguish between "normal slowness" and "abnormal slowness". Traditional RCA methods that assume the performance baseline is stable cannot distinguish them effectively.

Why is response time unstable? Further analysis of the composition of response time reveals that response time is primarily determined by LLM component interactions. As shown in Fig. 3, while retrieval and reranking times remain relatively stable, the LLM component's execution time dominates the overall response time. The LLM processing time comprises primarily the encoding and decoding phases. To understand the relationship between response time and these phases, we conducted correlation analysis as illustrated in Fig. 4. We find that response time is strongly correlated with the number of tokens generated in the decoding phase, with a Pearson correlation coefficient of 0.962. This phenomenon is intuitive and becomes even more noticeable in reasoning models (e.g., OpenAI o1/o3 [30], DeepSeek-R1 [8]). In summary, the instability of response time stems from two key factors: the variable number of generated tokens (for both the same and different requests due to the stochastic nature of LLM generation) and inconsistent token-generation speed (varying based on backend inference optimizations, GPU hardware resources, and other factors).

Novel LLM-specific performance faults evade traditional monitoring. The LLM input, which is composed of the query, context, and prompt, introduces novel performance faults. As shown in Fig. 6, an abnormal prompt can mislead the LLM into generating 300 words when only a sentence answer is expected (normal). This results in an abnormally long response time. However,

Fig. 4. Correlation between response time and contextual data.

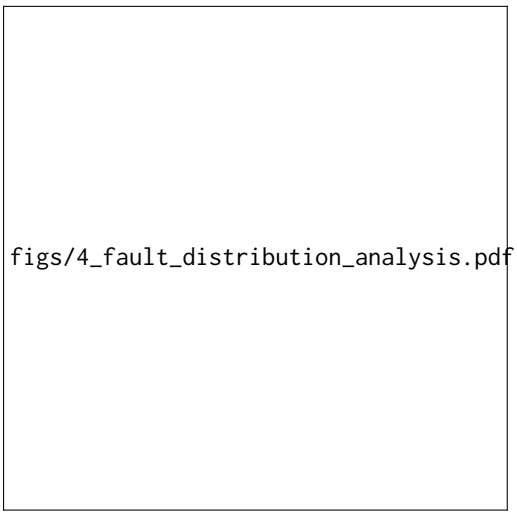


Fig. 5. LLM application faults (both quality-related and non-quality-related) [40].

Motivation 1: Response times naturally vary based on output length, making it difficult to clarify the normal and abnormal requests.

traditional resource metrics, such as CPU usage or memory usage, appear normal because the inference process operates as intended. Without analyzing LLM inputs and outputs, diagnosing such faults becomes infeasible.

Motivation 2: LLM-specific faults related to prompts and contexts cause performance faults while showing normal system metrics, creating detection blind spots that conventional monitoring can't address.

3.2.2 Response Quality Faults in LLM Applications. **Prevalence of response quality faults.** Based on existing research [27, 40], we have summarized common faults in LLM applications, as shown in Table 1. Response quality faults represent a significant portion of these faults. And as shown in Fig. 5, Shao et al. [40] manually tested 100 GitHub repositories of LLM applications and discovered 552 LLM-specific faults in total. Their analysis revealed that 46.9% of all discovered faults were directly related to response quality, highlighting the significance of this problem category.

Conventional observability data misses quality faults. Despite existing tools like Phoenix [2] and OpenLLMetry [43] for tracing LLM generation performance, and ENOVA [17] for monitoring deployment environments, they fail to detect quality faults. This happens because these tools focus on general metrics (e.g., response time, resource usage) rather than semantic content. For example, as shown in Fig. 6, when prompts are poorly constructed, LLMs still execute the inference process normally, producing low-quality outputs without triggering any conventional monitoring alerts since general metrics appear normal. There is a critical need for comprehensive observability.

Lack of evaluation methods for quality assessment. While we can divide response quality evaluation into distinct modules (query, context, prompt, and answer assessment), we currently lack effective methodologies to evaluate the quality of each module. This creates two problems: (1)

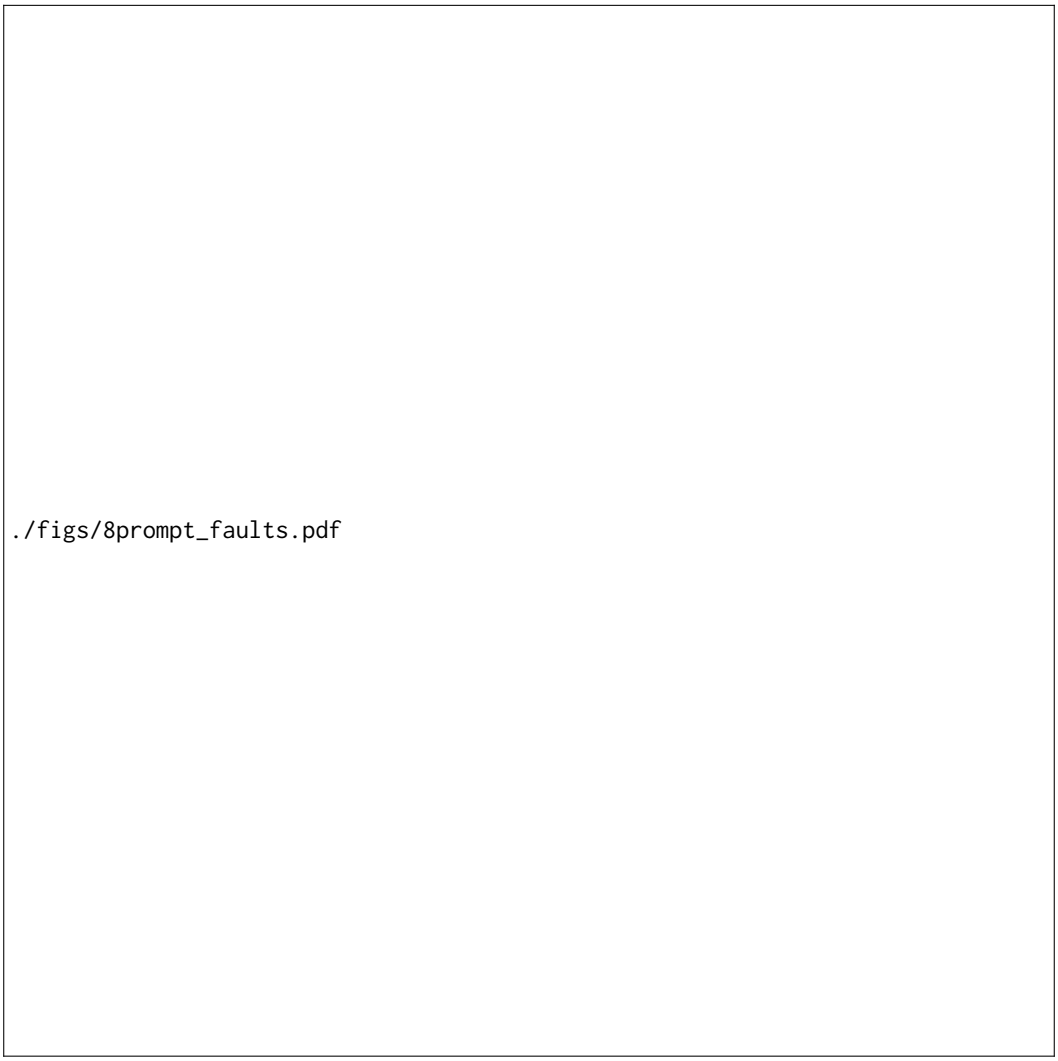


Fig. 6. A prompt fault case.

Motivation 3: Nearly half of LLM application faults (46.9%) are related to response quality, yet conventional observability data miss these faults because they only track general metrics, not response quality.

engineers cannot pinpoint which specific component caused the quality fault, and (2) there are no standardized metrics to quantify response quality objectively. This gap makes it challenging to systematically diagnose response quality faults.

Motivation 4: There are no standardized methods to evaluate quality across different components, making it impossible to pinpoint where quality faults occur.

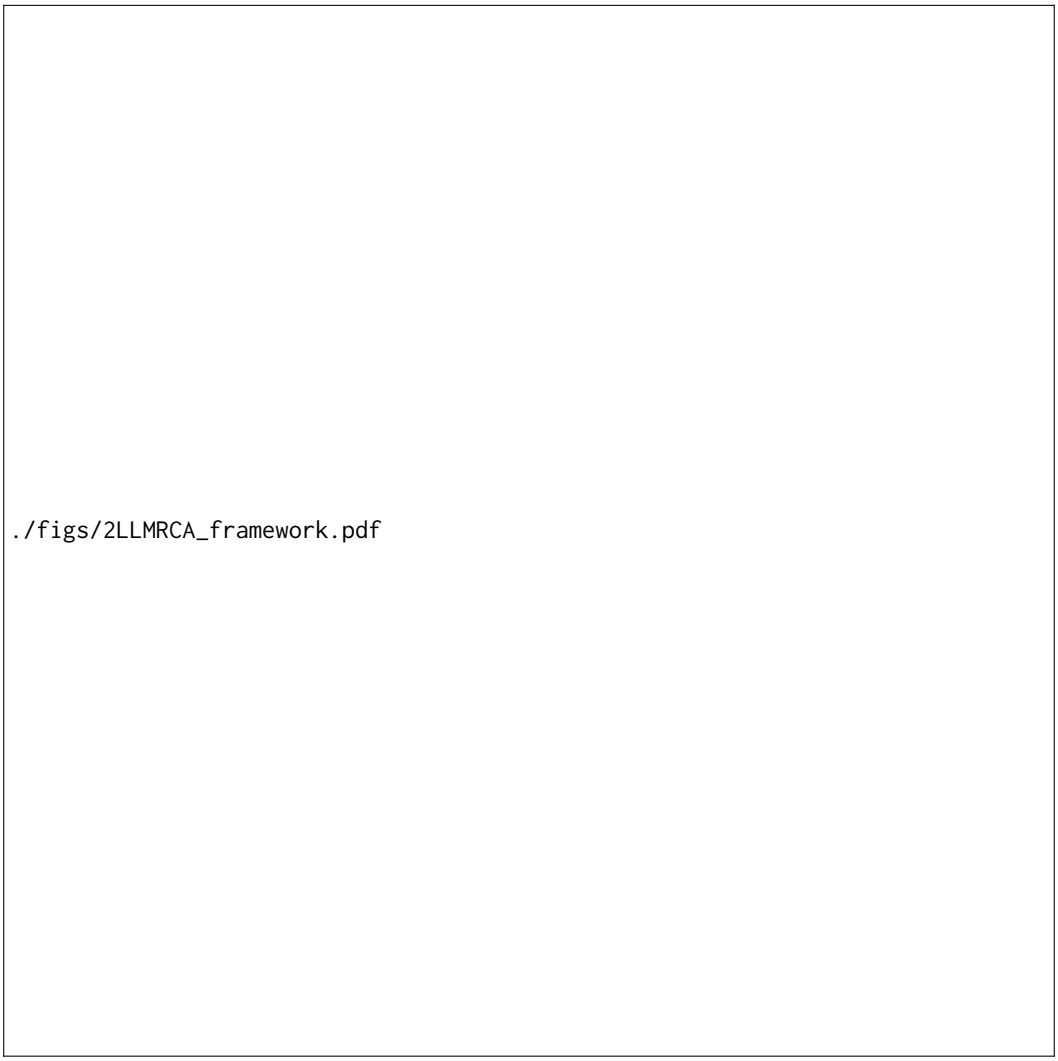


Fig. 7. The framework of LLMRCA.

4 LLMRCA FRAMEWORK

The LLMRCA framework follows a standard multimodal data-based RCA workflow with data collection, causal graph construction, and root cause analysis (§2), using distributed tracing to identify all components involved in LLM processing and their relationships and provide clear boundaries that enable targeted monitoring and analysis of LLM-related operations within the larger system. LLMRCA introduces key innovations, including enriching traces and adding context nodes to diagnose LLM-specific faults, classifying requests to address response time instability, and improving fault diagnosis with advanced graph neural networks and verification principles. The design choices within this framework, such as the specific GNN model and the use of a regression model for classification, are justified through extensive empirical evaluations (§5).

Fig. 8. The difference between traditional RCA and LLMRCA.

As shown in Fig. 7, in phase ① LLMRCA first extracts heterogeneous observability data, including metrics, logs, and traces from the application over a recent period. To overcome the fault-diagnosis blind spots in conventional monitoring systems **(Motivation 2, 3)**, LLMRCA extends observability data collection with LLM-specific information, capturing rich trace attributes like prompt structures, token generation statistics, and context relevance scores. ② The causal graph topology is constructed based on the traces and application information. The observability data are then transformed into a unified time series format and integrated as node attributes in the causal graph to create a comprehensive representation of system behavior. To evaluate the quality across components **(Motivation 4)**, LLMRCA transforms text content into quantifiable time-series metrics by computing token lengths and semantic similarity scores as node attributes in the causal graph. ③ To address the challenge of unstable response times that lead to false positives **(Motivation 1)**, LLMRCA classifies normal requests based on the fitted response time, which is predicted using an XGBoost regression model [6]. The class labels are encoded using a one-hot representation and are integrated as additional node attributes into the causal graph to train the Residual Graph Attention autoencoder model [26]. ④ When a fault occurs, LLMRCA classifies requests, uses the trained ResGAT model to calculate reconstruction errors, and incorporates a hierarchical verifier to identify the root cause. This phase applies Z-score normalization to reconstruction errors to identify root causes at component and inner-component levels, and incorporates a hierarchical verifier to enhance Precision and reduce false positives.

The main differences between traditional RCA and LLMRCA occur in phases ① and ② (§4.1, §4.2). Fig. 8 shows how LLMRCA builds on traditional methods. Traditional

RCA only tracks system metrics like CPU and memory usage. This works for general software but misses many LLM-specific issues. LLMRCA adds two important capabilities: (1) it analyzes the actual text content (prompts, contexts, answers), and (2) it checks if these texts relate to each other correctly. This enables LLMRCA to diagnose novel performance and quality faults.

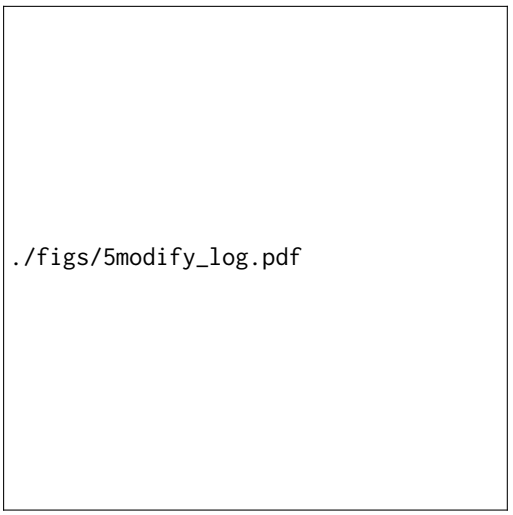
4.1 Data Collection

Data collection aims to aggregate different types of observability data to provide a basis for subsequent analysis. The three types of observability data (metrics, logs, and traces) have different characteristics and are typically used for different tasks.

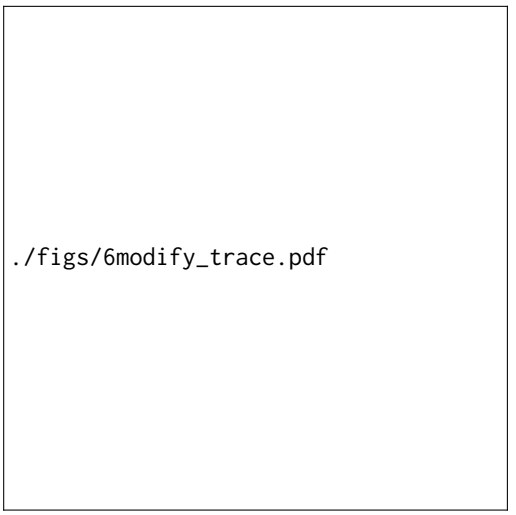
Metrics. Metrics are represented as time-series data, which describe system states over time. This study focuses on system-level metrics (e.g., CPU usage, memory usage). In particular, for embedding models, reranking models, and LLM models, LLMRCA introduces GPU-related metrics (e.g., GPU memory usage, GPU core clock frequency). These metrics are selected to capture resource consumption patterns that are critical indicators of performance bottlenecks in computation-intensive LLM components. These metrics are collected at a 10-second interval. Moreover, if any metric has missing values, the system interpolates these values using the average of adjacent data points.

Logs. Logs are generated by log statements instrumented by developers. LLMRCA adds log statements at key points in the LLM application workflow (e.g., embedding, reranking) and captures log entries from retriever frameworks (e.g., LlamaIndex) and generator frameworks (e.g., vLLM). We combine manual instrumentation for critical logic paths with automated instrumentation of existing frameworks (such as vLLM and LlamaIndex) to collect data and to ensure comprehensive coverage of critical execution paths and LLM-specific operations. Table 2 provides a full analysis of all collection points. A unified log format is adopted to record essential details, including timestamp, component identifier, host identifier (filename and line number), severity level, and log messages, as shown in Fig. 9a. This standardized format facilitates efficient log parsing and extraction of structured information using template-based techniques like DRAIN [12].

Traces. Traces are essential for constructing causal graphs and identifying dependencies across components. Although LLM application development frameworks have enabled tracing based on OpenTelemetry, there are still significant limitations. First, the trace across components within LLM applications is not fully correlated. Therefore, LLMRCA appends trace IDs and parent span IDs to request headers, enabling end-to-end tracing, as shown in Fig. 9b. This distributed context propagation technique ensures complete trace continuity across service boundaries, even when requests traverse multiple frameworks and execution environments. Second, trace points in current frameworks are incomplete. For example, the vLLM framework only records the overall request without detailed spans for the scheduler and decoder procedures. To address this limitation, LLMRCA implements customized instrumentation, particularly focusing on LLM-specific operations like token generation, context processing, and embedding computation. For example, we create spans for key logical steps like `retrieve`, `reranking`, and `llm_generate` (as shown in Fig. 1) by instrumenting the code in LlamaIndex and vLLM. Each span's parent-child relationship is recorded, forming the basis for the causal graph's topology. Third,



(a) Example of log format.



(b) Example of trace ID propagation.

Fig. 9. Example of data collection.

existing trace attributes are insufficient. For example, RAG-enhanced LLM applications developed with the LlamaIndex framework only record the entire query context, without distinguishing retrieved contexts and prompt templates. Therefore, LLMRCA extends the OpenTelemetry specification with domain-specific attributes that capture critical LLM contextual information. This includes breaking down prompts into structured attributes (e.g., `prompt.instruction`, `prompt.context`) and collecting other data such as semantic representations of queries, relevance scores, and token statistics. A detailed breakdown of all collected metrics, logs, and traces is provided in Table 2, including their specific collection points and key attributes. These enriched trace attributes provide the necessary context for diagnosing both performance and quality-related faults in LLM applications.

4.2 Causal Graph Construction

A causal graph $G = (V, E)$ is formally defined as a directed acyclic graph used to describe the causal relationships among components, metrics, and events within a system. Nodes can represent metrics, and their attributes are values [32]. Nodes can also represent system components, and their attributes are metrics (e.g., CPU usage, response time) [16]. In addition, nodes can represent events (e.g., error logs, alarm information) that are related to system faults or abnormal behavior [3, 51]. The edges represent the causal relationship between the nodes, and their weights represent the causal strength. To address the potential time mismatch from heterogeneous data sources, we employ a request-based alignment method. The trace for each request defines a precise time window. Metrics are averaged over this window, and logs with timestamps falling within this window are associated. This approach ensures all data is aligned to a specific request's context.

Table 2. Data collection details.

Data Source	Component / Level	Collection Point	Key Attributes
Metrics (psutil and pynvml)	Application-Level	Request completion event	response_time, completion_tokens, prompt_tokens, request_rate, throughput
	System-Level	Polling at 10s intervals	cpu_usage_percent, mem_rss, mem_vms, disk_read_bytes, disk_write_bytes, gpu_utilization_percent, gpu_memory_used_MB, gpu_power_W, gpu_graph_clock, gpu_sm_activity_percent
Logs (Python logging module)	RAG Pipeline	rag.py: Key steps (e.g., start retrieval)	timestamp, filename:lineno, levelname, message [rag.py:112] - INFO - Starting retrieval... [rag.py:133] - DEBUG - Formatted QA prompt...
	LLM Inference Engine	async_llm_engine.py: Request handling events	timestamp, filename:lineno, level, message [async_llm_engine.py:648] - INFO - Received request... [metrics.py:295] - INFO - Avg prompt throughput...
Traces (OpenTelemetry)	Query (RAG Pipeline)	rag.py: Entry/exit of the main query function	span_id, parent_id, name, start_time, end_time, input.value, output.value
	Retrieve	rag.py: LlamaIndex retriever call	retrieval.documents (content, score, metadata), input.value (query string)
	Embedding	LlamaIndex embedding model call	embedding.model_name, embedding.text, embedding.vector (partial)
	Reranking	rag.py: Reranker model call	reranker.model_name, reranker.input_documents, reranker.output_documents, reranker.query
	LLM Service	rag.py: HTTP client call to vLLM	llm.model_name, llm.token_count (prompt, completion, total), llm.prompts, llm.output_messages
	LLm Request	vLLM internal request processing	gen_ai.request (id, temperature), gen_ai.usage (prompt/completion tokens), gen_ai.latency (e2e, queue time)

LLMRCA extends the traditional causal graph by introducing context nodes that specifically model LLM-specific observability data. Unlike conventional RCA graphs that primarily capture system performance, these context nodes represent semantic and operational attributes unique to LLM processing pipelines. These nodes formalize domain-specific attributes critical for LLM execution behavior modeling, including query contexts, prompt templates, token lengths, and relevance scores, in order to diagnose unique LLM application faults. For example, token length is used to quantify processing load, while relevance scores measure the semantic alignment between user queries and generations.

To achieve multilevel RCA, LLMRCA integrates multimodal observability data to construct a causal graph that accurately represents the runtime states of LLM applications. The graph topology G is constructed based on traces and additional system information.

The node attributes are derived from a unified representation of metrics, logs, and trace attributes, creating a comprehensive view of system behavior across multiple abstraction levels. LLMRCA introduces different types of nodes to comprehensively represent the entire LLM application and identify root causes at multiple levels.

Graph Topology. The nodes V of the causal graph consist of the following types:

- Instance nodes $I = \{i_1, i_2, \ldots, i_k\} \subset V$ represent instances such as applications, components, and vector databases, reflecting both application-level and component-level states.
- Metric nodes $M = \{m_1, m_2, \ldots, m_k\} \subset V$ represent system metrics (e.g., CPU usage, GPU core clock frequency), reflecting host-level states.
- Log nodes $L = \{l_1, l_2, \ldots, l_k\} \subset V$ extract features of log templates, reflecting code-level states (e.g., exception handling, transaction checkpoints).
- Context nodes $C = \{c_1, c_2, \ldots, c_k\} \subset V$ capture enriched trace attributes, such as intermediate request results, reflecting context-level states.

The edges $E \subseteq V \times V$ of the causal graph are constructed as follows.

- Invocation relationships. If an instance node i_k invokes another instance node i_j (based on trace IDs, span IDs, and parent span IDs), an edge $i_k \rightarrow i_j$ is added.
- Monitoring relationships. If a metric m_j is associated with an instance i_k, an edge $i_k \rightarrow m_j$ is added.
- Code implementation relationships. If a log node l_j matches an instance node i_k (based on attributes like component and host identifiers), an edge $i_k \rightarrow l_j$ is added.
- Trace attribute relationships. If a context node c_j belongs to an instance i_k, an edge $i_k \rightarrow c_j$ is added.
- System metric dependencies. We use LLMs to analyze the system architecture and component dependencies to identify metric relationships [11]. By examining hardware, software, and resource patterns, the LLM determines the causal relationships between metrics. When m_k affects m_j (e.g., network packet loss $\rightarrow$ response time, CPU contention $\rightarrow$ processing speed $\rightarrow$ response time), an edge $m_k \rightarrow m_j$ is added.

Node Attributes. Given that the causal graph is constructed on a per-request basis, this study extracts data features relative to each request period rather than a fixed time interval. For each node $v \in V$, we define the attribute function $x : V \rightarrow \mathbb{R}$ as follows.

Instance Node Attribute. LLMRCA uses the span duration from traces as the instance node attribute. This attribute captures the processing time of each request within specific components, reflecting the execution state and performance of each instance node. It helps to identify execution efficiency and performance bottlenecks. The instance node attribute is defined as:

$$x_i^I = T_i, \tag{1}$$

where T_i is the span duration for request i measured in seconds from span start to end time.

Metric Node Attribute. LLMRCA collects metric data at a 10-second interval. For each request, system metrics collected over the 60 seconds preceding the request are averaged to form a sequence to represent the metric node attribute. This approach smooths fluctuations and provides stable performance metrics. For a request occurring at time t_i, the metric

node attribute is defined as:

$$x_i^M = \frac{1}{N} \sum_{k=1}^{N} M(t_i - k \cdot \Delta t), \tag{2}$$

where $M(t)$ is the sampled metric value at time t, $\Delta t = 10$s, and $N = 6$ (60 seconds) to capture the system state.

Log Node Attribute. Log templates are extracted from standardized log entries using Drain [12], and their occurrences within the request period are counted to generate a sequence. The log node attribute is defined as:

$$x_i^L = f(L), \tag{3}$$

where $L = \{l_1, l_2, \ldots, l_n\}$ is the set of log templates during the request processing, and $f(L)$ is the count of each template forming a frequency vector where each dimension represents a specific log pattern.

Context Node Attribute. LLMRCA uses span attributes from traces as the context node attribute, capturing intermediate results (e.g., retrieved documents, reranked contexts) that offer rich contextual information for RCA. The context node attribute is defined as:

$$x_i^C = \text{SpanAttributes}_i, \tag{4}$$

where SpanAttributes_i are metrics extracted from the span attributes for request i, including LLM-specific data such as prompt content, retrieved documents, and generation parameters.

For text attributes (e.g., retrieved documents, reranked contexts), we have two processing methods, as follows:

For $\mathcal{AP}$ RCA, the token length of text is computed to indicate the data processing load.

$$x_i^C = \phi_{len}(D_i), \tag{5}$$

where D_i is the text attribute of the span and ϕ_{len} is the token length computation function that counts tokens according to the LLM's tokenizer.

For $\mathcal{RQ}$ RCA, we analyze three key relationships to pinpoint quality issues:

- *Query-Context relevance.* We evaluate whether the vector database contains up-to-date information by computing the relevance scores between the query and retrieved context.
- *Context-Answer relevance.* We measure how well the LLM utilizes the retrieved context by computing the relevance scores between the context and the generated answer.
- *Prompt evaluation.* We use an LLM to judge whether the prompt design is reasonable.

Both relevance scores use the same cosine similarity formula:

$$x_i = \phi_{sim}(A_i, B_i) = \frac{\mathbf{a}_i \cdot \mathbf{b}_i}{\|\mathbf{a}_i\| \|\mathbf{b}_i\|}, \tag{6}$$

where the similarity is calculated between two key pairs of text inputs: (1) the user's query string and the retrieved context string (Query-Context), and (2) the retrieved context string and the LLM-generated answer string (Context-Answer). For each pair, the text inputs (A_i, B_i) are first converted into embedding vectors $(\mathbf{a}_i, \mathbf{b}_i)$ using an embedding model.

Algorithm 1: Model Training

Input: Normal causal graphs G_n (training set), Requests $\{X_{\text{token\_count}}, y_{\text{response\_time}}\}$, XGBoost parameters, ResGAT parameters

Output: Trained XGBoost model M_{XGBoost}, Trained ResGAT model M_{ResGAT}

 # Train XGBoost for Response Time Prediction

Train XGBoost model: $M_{\text{XGBoost}} \leftarrow \text{XGBoost}(X_{\text{token\_count}}, y_{\text{response\_time}})$

Determine class intervals based on predicted response times: $C \leftarrow \text{DetermineClassIntervals}(y_{\text{response\_time}})$

Record partition points: $P \leftarrow \text{RecordPartitionPoints}(C)$

 # Feature Augmentation using XGBoost Predictions

foreach *new query request* **do**

  Predict response time: $\hat{y} \leftarrow M_{\text{XGBoost}}(X_{\text{token\_count}})$

  Classify request types based on partition points: class $\leftarrow \text{Classify}(\hat{y}, P)$

  One-hot encode the class labels: CausalGraph.IntegrateAttributes(OneHotEncode(class))

 # Train ResGAT for Graph Representation Learning

Initialize ResGAT parameters: $W^{(l)}, W_r^{(l)}, a$ (for each layer)

Set training hyperparameters: η (learning rate), E (number of epochs), L (number of ResGAT layers)

for *epoch* $e \leftarrow 1$ **to** E **do**

  foreach *batch* $\mathcal{B}$ *in* G_n **do**

    foreach *graph* $G_i = (V_i, E_i) \in \mathcal{B}$ **do**

      for *layer* $l \leftarrow 1$ **to** L **do**

$$h_i^{(l+1)} \leftarrow \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{(l)} W^{(l)} h_j^{(l)} + W_r^{(l)} h_i^{(l)}\right)$$

$$\alpha_{ij}^{(l)} \leftarrow \frac{\exp\left(\text{LeakyReLU}\left(a^\top [W^{(l)} h_i^{(l)} \| W^{(l)} h_j^{(l)}]\right)\right)}{\sum_{k \in \mathcal{N}(i)} \exp\left(\text{LeakyReLU}\left(a^\top [W^{(l)} h_i^{(l)} \| W^{(l)} h_k^{(l)}]\right)\right)}$$

      $z_i \leftarrow h_i^{(L)}$

$$h_i^{\text{dec}} \leftarrow \text{LeakyReLU}\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{\text{dec}} W^{\text{dec}} z_j + W_r^{\text{dec}} z_i\right)$$

      $\hat{x}_i \leftarrow \text{Linear}(h_i^{\text{dec}})$

    $\mathcal{L}_{\text{recon}} \leftarrow \frac{1}{n} \sum_{i=1}^{n} \|x_i - \hat{x}_i\|^2$

    Update parameters: $W^{(l)}, W_r^{(l)}, a \leftarrow W^{(l)}, W_r^{(l)}, a - \eta \nabla \mathcal{L}_{\text{recon}}$

return $M_{XGBoost}, M_{ResGAT}$

A low Query-Context relevance indicates retrieval issues, while a low Context-Answer relevance indicates LLM generation issues.

4.3 Model Training

As shown in Fig. 2, different requests passing through the same component have different response times. This study proposes an unsupervised learning approach, including classification for feature augmentation and ResGAT for RCA. Algorithm 1 describes the entire training process of the model.

Feature Augmentation. To address the unstable response time issue, we combine regression and classification. First, we train an XGBoost regression model to predict the fitted response time for each request. This fitted response time is a theoretical normal response time for a request. XGBoost regression is chosen for its efficiency and accuracy because it can handle large-scale data, effectively model nonlinear relationships, and is

well-suited for online tasks. The model is trained on normal requests, using the number of input and output tokens as features (based on the high correlation in Fig. 4) and the actual response time as the ground truth label. Second, we perform classification based on the fitted response time. In the training phase, partition points are determined and fixed using equal interval binning of fitted response times from the training data (normal requests). In the test phase, these fixed partition points are used to classify new requests. The number of classes is configurable. For example, if fitted response times range from 0 to 4.5 seconds and we configure 3 classes, the binning process defines class boundaries as "fast" (0-1.5s), "medium" (1.5-3s), and "slow" (3s+). These dynamically generated thresholds ensure the method is robust. Finally, the resulting class label for each request is one-hot encoded (e.g., [1,0,0] for "fast") and integrated as an additional node attribute into the causal graph.

ResGAT for RCA. ResGAT consists of an encoder-decoder architecture, where the encoder captures the latent probabilistic distribution of nodes, and the decoder reconstructs the features. ResGAT is particularly suited for this task because it uses attention mechanisms to dynamically capture important dependencies between nodes in the causal graph, and its residual connections effectively address the vanishing gradient problem during backpropagation.

The encoder learns latent representations of graph nodes using stacked ResGAT layers. Each ResGAT layer transforms node features by adaptively weighting neighbor information based on their relevance to the central node, while preserving original feature information through residual connections. At layer l, each ResGAT layer updates the feature vector of node i by aggregating features from its neighbors $\mathcal{N}(i)$ using attention coefficients α_{ij}, and incorporating a residual connection that allows for better flow of information across layers:

$$h_i^{(l+1)} = \sigma\left(\sum_{j\in\mathcal{N}(i)} \alpha_{ij}^{(l)} W^{(l)} h_j^{(l)} + W_r^{(l)} h_i^{(l)}\right), \tag{7}$$

$$\alpha_{ij}^{(l)} = \frac{\exp\left(\text{LeakyReLU}\left(a^\top \left[W^{(l)} h_i^{(l)} \| W^{(l)} h_j^{(l)}\right]\right)\right)}{\sum_{k\in\mathcal{N}(i)} \exp\left(\text{LeakyReLU}\left(a^\top \left[W^{(l)} h_i^{(l)} \| W^{(l)} h_k^{(l)}\right]\right)\right)}, \tag{8}$$

where $h_i^{(l)}$ is the feature vector of node i at layer l. $\mathcal{N}(i)$ is the set of neighbors of node i. $\alpha_{ij}^{(l)}$ is the attention coefficient between nodes i and j at layer l. σ is the activation function (e.g., LeakyReLU, ReLU). $W^{(l)}$ and $W_r^{(l)}$ are trainable weight matrices at layer l. a is a trainable attention vector for computing attention scores. $\|$ represents concatenation.

The output $h_i^{(L)}$ of the final layer of ResGAT represents the latent representation of node i after passing through the encoder's multiple ResGAT layers. We can denote this as $z = h_i^{(L)}$.

The decoder reconstructs the node features $\hat{x}_i$ from the latent representations z. The decoder's primary function is to transform the encoded representations back into the original feature space, providing a basis for calculating reconstruction errors that indicate anomalies. It first uses a GAT layer to aggregate information from neighboring latent representations and a residual connection to retain important information, followed by a

linear layer to generate the reconstructed features. This process and reconstruction loss computation are denoted as follows.

$$h_i^{\text{dec}} = \text{LeakyReLU}\left(\sum_{j\in\mathcal{N}(i)} \alpha_{ij}^{\text{dec}} W^{\text{dec}} z_j + W_r^{\text{dec}} z_i\right), \quad \hat{x}_i = \text{Linear}(h_i^{\text{dec}}), \tag{9}$$

$$\mathcal{L}_{\text{recon}} = \frac{1}{n}\sum_{i=1}^{n} \|x_i - \hat{x}_i\|^2, \tag{10}$$

where $\mathcal{L}_{\text{recon}}$ is the reconstruction loss, n is the total number of nodes in the graph, x_i is the original feature vector of node i, and $\hat{x}_i$ is the reconstructed feature vector generated by the decoder. During training, this mean squared error loss is minimized to ensure that the model accurately captures the normal behavior patterns of the causal graph. In the fault detection phase, nodes with high reconstruction errors are identified as potential root causes of anomalies.

4.4 Root Cause Analysis

The comprehensive causal graph constructed from multimodal observability data enables the calculation of the reconstruction errors for node attributes through the ResGAT model. However, these reconstruction errors cannot directly identify root causes due to two main reasons. (1) Heterogeneity of the Causal Graph: Although nodes are uniformly represented using multimodal data, they inherently differ in nature, as the causal graph is heterogeneous. (2) Variations in Reconstruction Error Scales: The ResGAT model fits different node features to varying degrees of accuracy, leading to variations in the scale of reconstruction errors across features.

To address these issues, LLMRCA employs a two-stage approach to realize RCA: (i) first normalizes reconstruction errors to account for heterogeneity, (ii) then applies hierarchical verification principles to identify the most probable root causes at both component and inner-component levels.

As shown in Algorithm 2, we employ Z-score normalization of reconstruction errors. During training, we calculate the reconstruction error for each graph i and node j: $\text{RE}_{ij} = (x_{ij} - \hat{x}_{ij})^2$. The analysis uses the reconstruction errors from the model after the training process is complete. $\mu_j \leftarrow \frac{1}{|G_n|}\sum_{i=1}^{|G_n|} \text{RE}_{ij}$, $\sigma_j \leftarrow \sqrt{\frac{1}{|G_n|}\sum_{i=1}^{|G_n|}(\text{RE}_{ij} - \mu_j)^2}$, where x_{ij} and $\hat{x}_{ij}$ are the original and reconstructed values of node j for graph i. This final stable distribution provides a reference for root cause analysis, following standard practices [16]. μ_j and σ_j are the mean and standard deviation of reconstruction errors for each node type j, computed across all normal graphs in the training set. During prediction, for an abnormal causal graph i, we compute the Z-score for each node j as: $Z_{ij} = \frac{\text{RE}_{ij} - \mu_j}{\sigma_j}$. The Z-score indicates how far the reconstruction error deviates from the normal training distribution. The nodes with the highest Z_{ij} values are potential root causes.

To enhance the precision of RCA and minimize false positives, we implement a hierarchical verifier specifically designed for our causal graph. This verifier, illustrated in Fig. 15, incorporates three structured verification principles.

Algorithm 2: Root Cause Analysis

Input: Normal causal graphs G_n (training set), Abnormal causal graphs G_{abn} (predict set)
Output: Component level root causes, Inner-Component level root causes

 # Compute Reconstruction Errors, Mean and Std for Training Set G_n
foreach *graph* $i \in G_n$ **do**
  foreach *node type* j *in graph* i **do**
    $\mathrm{RE}_{ij} \leftarrow (x_{ij} - \hat{x}_{ij})^2$
foreach *node* j **do**
  $\mu_j \leftarrow \frac{1}{|G_n|}\sum_{i=1}^{|G_n|}\mathrm{RE}_{ij}, \sigma_j \leftarrow \sqrt{\frac{1}{|G_n|}\sum_{i=1}^{|G_n|}(\mathrm{RE}_{ij} - \mu_j)^2}$
 # Compute Z-scores for Abnormal Graph G_{abn}
foreach *graph* $i \in G_{abn}$ **do**
  foreach *node* j *of graph* i **do**
    $Z_{ij} \leftarrow \frac{\mathrm{RE}_{ij} - \mu_j}{\sigma_j}$
 # Apply Verification Principles for RCA
foreach *graph* $i \in G_{abn}$ **do**
  Sort nodes by Z_{ij} in descending order
  # Inner-Component Level Root Causes
  foreach *node* j *in sorted list* **do**
    if *node* j *is a leaf node (metric, log, or context node)* **then**
      Check anomaly propagation principle: **if** *there exists a path to root where all intermediate nodes satisfy* $Z_{ij} > 1$ **then**
        Add node j to Inner-Component level root causes
  # Component Level Root Causes
  Initialize an empty list for Component level root causes
  foreach *node* j *in graph* i **do**
    if *node* j *is an instance node and an ancestor of any Inner-Component level root cause* **then**
      Add node j to the potential Component level root causes
    Sort potential Component level root causes by Z_{ij} in descending order.
return *Component level root causes, Inner-Component level root causes*

- *Inner-component level root cause sorting principle.* Inner-component level root causes must be leaf nodes, indicating that observability data directly links to the root cause. They are sorted by Z-scores.
- *Anomaly propagation principle.* A valid root cause must have at least one complete path from the leaf node (root cause) to the root node, where all intermediate nodes exhibit anomalous behavior.
- *Component level root cause sorting principle.* Component level root causes are sorted in two stages. If component level nodes are ancestors of an inner-component level root cause, they are sorted first. Among these ancestor nodes, further sorting is based on their Z-scores.

5 EXPERIMENTAL EVALUATION

In this section, we evaluate LLMRCA to answer the following research questions (RQs).

- **RQ1:** How effective is LLMRCA in multilevel RCA for LLM applications?
- **RQ2:** How do multimodal data and different modules contribute to effectiveness?

- **RQ3:** How do the different configurations impact the effectiveness?
- **RQ4:** How efficient is LLMRCA, and how does it scale with increasing graph size?

5.1 Experiment Setup

The experiments are conducted on a system running Ubuntu 24.04, equipped with an Intel Xeon Gold 6326 CPU with 64 cores and a clock speed of 2.90GHz. The system has 128GB of memory, 6 NVIDIA A40-40GB GPUs, and the CUDA version is 12.4. All methods are implemented using popular Python libraries. The XGBoost model is configured with 4 classes. The ResGAT model has 4 layers, with the hidden dimension being 32, and the latent dimension being 16. The hidden dimension is the vector size for processing node features. The latent dimension is the final encoder output size. The optimizer is Adam [19] with a learning rate of 0.001, and the number of epochs is 100.

Benchmark LLM Application. We deploy a RAG-enhanced LLM application based on Llamaindex and vLLM and follow the proposed methodology shown in Fig. 7. We evaluate on a single RAG-enhanced LLM application. It is a general-purpose question-answering system. To ensure broad applicability, our benchmark system is representative. First, RAG is widely used and supported by mainstream frameworks like LangChain and LlamaIndex. Second, our system uses standard components (Llama-3-8B, vLLM, Qdrant) common in production. Third, our fault injection covers fault types from real-world LLM application failures. Fourth, our core principles (data collection, causal graph construction, and root cause analysis) can extend to other LLM applications. The framework supports domain adaptation through customizable metrics and extensible nodes. Different domains can add domain-specific nodes: medical applications can add medical terminology accuracy, and financial applications can add compliance checks. The system consists of a retriever, a reranker, and a generator. The process involves embedding, cosine similarity matching, reranking, fusion of the query, context, and prompt, and LLM generation. The vector database Qdrant [36] is built by splitting relevant documents into chunks. The default configuration of the LLM application is as shown in Table 3.

We use RGB [4] as a query dataset, which assesses the noise robustness, negative rejection, information integration, and counterfactual robustness of LLMs in the context of RAG. The application uses RGB benchmark queries to represent real-world workloads. The design of our approach is domain-independent, including the data collection (Table 2), the fault injection (Table 6), and the RCA framework applies universally across application systems.

Due to the lack of observability in the LLM application (e.g., missing system resource metrics, incomplete traces, lack of key logs, and unstandardized log format), we collect system resource metrics using the process IDs (pids) of related application processes (§4.1). We insert logs at key steps and standardize the log format (Fig. 9a). Moreover, we use the OpenTelemetry SDK to propagate trace and span IDs, weaving together different components to obtain complete trace information (Fig. 9b).

System-level metrics are collected using psutil [34] and pynvml [35]. Logs are stored in a local file system with `.log` file format. For trace collection, we use Phoenix [2] to store trace data in `.db` database format. We measured overhead across 1,000 requests: baseline

Table 3. The default configurations.

Parameter	Value
Embedding Model	bge-large-en-v1.5
Embedding Length	1024
Vector Database	Qdrant
Chunk Size	1024
Chunk Overlap	50
Reranking Model	bge-reranker-large
LLM Model	Llama-3-8B-Instruct
Similarity Match Top-k	10
Rerank Top-k	3
LLM Service Framework	vLLM
API Parameter Temperature	0.2
API Parameter Max Tokens	256

Table 4. The injected fault classes.

Class	Faults
$\mathcal{AP}$	CPU contention, GPU contention, Network congestion, Code defect
$\mathcal{AP}$ and $\mathcal{RQ}$	Poor prompt design, Outdated database, LLM fault, Embedding fault, Reranking fault, API parameters fault

Table 5. Summary of LLMRCA datasets.

Dataset	Train	Test for $\mathcal{AP}$	Test for $\mathcal{RQ}$
Single-type Requests	1364/1000	16751/12600	16751/4200
Multi-type Requests	1459/1000	17632/12600	17632/4200

averaged 2.1s, enhanced collection averaged 2.19s (+4.3%). Component breakdown shows metrics (+0.9%), logs (+1.6%), and traces (+1.8%).

Fault Injection. We inject faults into the benchmark to simulate real-world faults and generate anomalous data. Because each fault is injected by us at a known location (e.g., a specific component or resource), the ground truth for each anomalous request is known with certainty. The fault injection design avoids bias. We do not pre-select faults that would better support any single data modality. The fault selection is based on a systematic analysis of real-world LLM application faults in existing research [40, 53], ensuring our evaluation covers representative fault scenarios. Table 6 provides a detailed implementation of our fault injection methodology. Each fault type uses specific injection methods: CPU contention uses cgroups resource control with commands like `"cgcreate -g cpu:/llm_app"`, GPU contention uses `"nvidia-smi"` resource limits, and network congestion adds `"asyncio.sleep"` delays in RAG retrieval functions, and context faults modify prompt templates, knowledge bases, and API parameters. Inspired by faults in LLM applications (Table 1), we inject a total of 42 faults (22 resource-related, 6 code-related, and 14 request context-related). These 42 faults consist of 10 fault types listed in Table 4. They are categorized into two classes based on the faults they cause. Since $\mathcal{RQ}$ related faults often lead to changes in generations and affect $\mathcal{AP}$, there is no $\mathcal{RQ}$ class.

Data Collection. Using the above methods, we collect both normal and anomalous data from experiments involving single-type requests (with stable response time) and multi-type requests (100 types with unstable response time). All data comes from a single RAG-enhanced LLM application to ensure experimental consistency. The type of request refers to the query content. Single-type means all requests use the same query, while multi-type means requests use 100 different queries. All queries are randomly selected from the RGB dataset. For single-type request experiments, we collect 1364 normal entries and 16751 anomalous entries. For multi-type request experiments, we collect 1459 normal entries and 17632 anomalous entries. And then, we performed data cleaning to ensure data quality. For the training set, we remove incomplete records (e.g., missing trace spans or metrics) and filter out outliers whose response times are significantly beyond the normal range (the 5-95 percentiles). For the test set, we review abnormal samples to ensure each is associated with its known ground truth. After data cleaning, each dataset (single-type and multi-type) includes 1000 entries for training, 12600 entries for $\mathcal{AP}$ testing, and 4200 entries for $\mathcal{RQ}$ testing. The test set is larger than the training set because it contains a wide

Table 6. Fault injection implementation details.

Fault	Fault Type	Injection Method	Implementation	Ground Truth Location
CPU Contention	Resource Exhaustion (Host)	cgroups resource control	cgcreate-gcpu:/llm_app;echo50000>/sys/fs/cgroup/cpu/llm_app/cpu.cfs_quota_us. Levels: 5%, 8%, 10%, 12%, 15% CPU limit	CPU metrics
GPU Contention	Resource Exhaustion (Host)	nvidia-smi power limit	nvidia-smi -i 0 -pl 200 (200W, 400W, 600W, 800W). Levels: Power limits 200-800W	GPU metrics
Network Congestion	Component-Request Timeout (Component)	Code modification with asyncio.sleep	Added await asyncio.sleep(1) in rag.py:_aretrieve(). Levels: 1s, 2s, 3s delays	Latency metrics
Code Defect	Bugs in Code (Code)	Source code modification	Added redundant loops and exception handling in key functions. Levels: Repeat printing 2-4 times	Code logs
Poor Prompt Design	Unclear Prompt Context (Context)	Template modification	Changed prompt from "max two sentences, up to 250 characters" to "answer with 300 words". Levels: Template variations	Code logs/Trace attributes
Outdated Database	Knowledge Misalignment (Context)	Knowledge base replacement	Replaced RGB dataset with NQ dataset in vector database. Levels: Complete KB swap	Trace attributes
LLM Fault	Low-Quality Responses (Application)	Model replacement	Changed from Llama-3-8B to Qwen2.5-7B. Levels: Different model architectures	Trace attributes
Embedding Fault	Low-Quality Embeddings (Context)	Model replacement	Changed from bge-large-en-v1.5 to multilingual-e5-large. Levels: Different embedding models	Trace attributes
Reranking Fault	Improper Retrieval Results (Context)	Model replacement	Changed from bge-reranker-large to bge-reranker-v2-m3. Levels: Different reranking models	Trace attributes
API Parameters Fault	Format Mismatch (Code)	Parameter modification	Changed max_tokens from 256 to 1-3, retrieve_num from 10 to 3-6, top_k from 3 to 1-5. Levels: Various parameter ranges	Trace attributes

variety of injected faults to comprehensively evaluate the model's generalization capability. Given that the application's call structure is fixed, this amount of data is sufficient to capture patterns.

Evaluation Metric. We introduce Hit Ratio (HR) and Normalized Discounted Cumulative Gain (NDCG) as the evaluation metrics for RCA. HR@k indicates whether the predicted top-k list contains the root cause. $HR@k = \frac{1}{N}\sum_{i=1}^{N}(s^i \in S^i_{[1:k]})$, where $S^i_{[1:k]}$ represents the predicted top-k root cause for the representation i, s^i denotes the actual root causes, and N is the number of abnormal requests. $NDCG@k$ measures the extent to which the root

causes appear at higher positions in the candidate list. $DCG@k = \frac{1}{N}\sum_{i=1}^{N}(\sum_{i=1}^{k}\frac{2^{rel_i}-1}{\log_2(i+1)})$, $IDCG@k = \frac{1}{N}\sum_{i=1}^{N}(\sum_{i=1}^{k}\frac{2^{rel_i^*}-1}{\log_2(i+1)})$, $NDCG@k = \frac{DCG@k}{IDCG@k}$, where rel_i represents whether the position i is a root cause, while rel_i^* represents whether the position i sorted in descending order of relevance scores is a root cause.

5.2 Baselines

We use the following eight unsupervised RCA methods based on different data types as our baselines. Because no specialized RCA baselines exist for LLM applications, we select state-of-the-art methods from the microservices domain, covering metric-based, trace-based, and multimodal approaches. To ensure fair comparison, all baselines use the same experimental conditions. We obtain source code from public repositories and transform our observability data into each method's input format. All baselines and LLMRCA use the same training dataset and test dataset. Supervised methods are not considered, as they require labeled training data, which is difficult to obtain enough data in production environments. Moreover, the baselines are consistent with our methods due to the unsupervised nature.

- **MonitorRank** [18] is a trace-based RCA approach that employs a customized random walk with self-edges and back edges to identify root causes.
- **MicroScope** [21] is a metric-based RCA approach that identifies root causes by analyzing the correlation of metrics within a dependency network.
- **TraceAnomaly** [24] is a trace-based RCA method that uses deep learning to model normal trace patterns offline and detect anomalous traces online.
- **MicroRCA** [49] is a metric-based RCA approach that uses the PageRank algorithm on an extracted anomaly sub-graph to localize suspicious services.
- **MicroRank** [50] is a trace-based RCA approach that combines the personalized Pagerank method with the Spectrum method to identify suspicious root causes.
- **FaasRCA** [16] integrates multimodal observability data and uses a graph attention network-based autoencoder to identify root causes at different stages of the serverless lifecycle.
- **PDiagnose** [14] integrates multimodal data and converts them into time series, applying a voting mechanism to abnormal time series to determine the root cause.
- **Nezha** [51] is a fine-grained RCA approach that leverages multimodal data to identify root causes at the code region and resource type level.

5.3 RQ1: Effectiveness in Root Cause Analysis

To comprehensively evaluate the effectiveness of LLMRCA in diagnosing $\mathcal{AP}$ related faults, we categorize it at both the component level and inner-component level. The baseline methods are designed for microservice environments rather than LLM applications. No RCA methods exist for LLM applications at the time of our research. Our evaluation shows that existing methods face limitations when applied to LLM applications, particularly in diagnosing response quality faults that need semantic analysis. The evaluation results for different methods are presented for both single-type and multi-type requests. At the component level, the ground truth is defined as the known injected components. At the inner-component level, the ground truth includes the code regions, resource types,

Table 7. Effectiveness comparison for application performance at the component level.

Approach	Single-type Requests					Multi-type Requests				
	HR@1	*HR@3*	*HR@5*	*NDCG@3*	*NDCG@5*	*HR@1*	*HR@3*	*HR@5*	*NDCG@3*	*NDCG@5*
MonitorRank	28.57	57.14	78.57	43.48	52.8	21.43	61.9	88.1	43.54	55.29
MicroScope	33.33	59.52	83.33	47.05	56.98	23.81	57.14	83.33	42.03	53.6
TraceAnomaly	30.95	61.9	73.81	49.07	53.75	28.57	57.14	78.57	45.35	54.15
MicroRCA	26.19	38.1	54.76	32.45	39.54	19.05	35.71	57.14	28.63	37.23
MicroRank	23.81	50	64.29	38.27	43.11	21.43	38.1	59.52	30	38.95
FaasRCA	32.72	58.55	83.33	47.62	58.89	28.16	59.04	83.33	46.19	57.34
PDiagnose	33.33	69.05	76.19	54.12	57.47	23.81	73.81	83.33	52.02	56.52
Nezha	42.86	57.14	88.1	51.56	64.31	33.33	52.38	80.95	44.88	57.5
LLMRCA w/o $\mathcal{M}$	51.34	75	83.33	64.39	67.87	40.89	70.21	83.33	57.33	61.49
LLMRCA w/o $\mathcal{L}$	67.17	73.69	81.24	71.3	74.47	63.16	79.61	82.49	72.82	73.88
LLMRCA w/o $\mathcal{T}_c$	59.29	72.25	88.6	66.85	73.71	50.92	71.62	84.73	62.02	67.41
LLMRCA w/o $\mathcal{V}$	67.5	76.32	87.38	72.2	76.01	51.05	70.88	80.92	62.67	66.93
LLMRCA w/o $\mathcal{R}$	63.69	68.57	84.9	66.3	73.28	57.19	74.1	83.76	67.49	71.64
LLMRCA w/o $\mathcal{C}$	-	-	-	-	-	59.08	78.98	87.11	71.56	74.83
LLMRCA	**73.33**	**79.56**	**91.51**	**76.77**	**81.79**	**65.9**	**79.67**	**89.21**	**72.65**	**76.8**

and contexts extracted from the fault-injected operations. LLMRCA achieves the best performance, as it extracts multimodal data features as heterogeneous graphs, classifies request types, and considers the request context information. In addition, we evaluate the effectiveness of LLMRCA in diagnosing $\mathcal{RQ}$ related faults for both single-type and multi-type requests. The ground truths are derived from context extracted from the fault-injected operations.

5.3.1 Application Performance at Component Level. Table 7 shows the evaluation results of different methods for $\mathcal{AP}$ at the component level for both single-type and multi-type requests. First, the effectiveness of baselines on multi-type requests is lower than on single-type requests due to the interference caused by different response time for different types of requests. Second, multimodal methods (FaasRCA, PDiagnose, and Nezha) generally outperform single-modal methods (MicroScope, MicroRCA, TraceAnomaly, and MicroRank) in terms of effectiveness.

MicroScope and MicroRCA (metric-based methods) achieve HR@1 values of 33.33% and 26.19% for single-type requests, and 23.81% and 19.05% for multi-type requests. MonitorRank, TraceAnomaly, and MicroRank (trace-based methods) achieve HR@1 values of 28.57%, 30.95%, and 23.81% for single-type requests, and 21.43%, 28.57%, and 21.43% for multi-type requests. These methods perform poorly as they neglect critical information from other data modalities. MicroScope and MicroRCA better identify resource-related faults, while MonitorRank, TraceAnomaly, and MicroRank are limited to diagnosing latency-related faults. Single-modal approaches struggle to address complex faults at the component level, which often involve multimodal anomalies.

FaasRCA, PDiagnose, and Nezha (multimodal data-based methods) show better performance by leveraging multiple data sources. For single-type requests, their HR@1 values are 32.72%, 33.33%, and 42.86%. And for multi-type requests, HR@1 values are 28.16%, 23.81%, and 33.33%. However, their accuracy is still insufficient. The primary reasons are: firstly, LLM applications involve fine-grained faults within the request context that these methods do not utilize. Secondly, FaasRCA uses GAT to embed multimodal data

Table 8. Effectiveness comparison for application performance at the inner-component level.

Approach	Single-type Requests					Multi-type Requests				
	HR@1	*HR@3*	*HR@5*	*NDCG@3*	*NDCG@5*	*HR@1*	*HR@3*	*HR@5*	*NDCG@3*	*NDCG@5*
PDiagnose	16.67	42.86	42.86	32.57	32.57	11.9	30.95	33.33	23.61	24.64
Nezha	21.43	33.33	33.33	28.94	28.94	19.05	38.1	38.1	31.07	31.07
LLMRCA w/o $\mathcal{M}$	44.29	47.62	47.62	46.39	46.39	36.03	43.06	43.98	40.32	40.7
LLMRCA w/o $\mathcal{L}$	64.94	78.09	80.43	73.17	74.09	46.61	49.16	52.75	47.91	49.32
LLMRCA w/o $\mathcal{T}_c$	50.16	50.48	58.06	50.36	53.34	46.37	49.04	52.13	47.74	48.94
LLMRCA w/o $\mathcal{V}$	62.31	74.6	82.74	69.13	72.38	45.8	60.63	70.89	54.63	58.62
LLMRCA w/o $\mathcal{R}$	60.62	68.09	73.1	65.32	67.26	52.12	69.2	78.72	61.96	65.96
LLMRCA w/o $\mathcal{C}$	-	-	-	-	-	49.29	60.71	72.29	55.62	60.29
LLMRCA	**65.16**	**81.59**	**87.4**	**75.21**	**77.68**	**61.47**	**78.18**	**81.5**	**70.46**	**71.79**

into high-dimensional vectors. However, it relies solely on neural networks to learn latent patterns, making it sensitive and unstable. PDiagnose uses log statistics and trace latency information but ignores the trace topology, and requires manual configuration of parameters and thresholds, which affects model performance. Nezha extracts multimodal data into an event graph, integrating various data types, but loses raw data details and requires manual alert rules, leading to potential false positives and missed detections.

In conclusion, LLMRCA achieves the best effectiveness in identifying root causes of $\mathcal{AP}$ faults at the component level, with HR@k and NDCG@k values exceeding 73% for single-type requests. And for multi-type requests, HR@1 is 65.9%, and NDCG@3 is 72.65%. Compared to baselines, LLMRCA improves HR@k by 1.26% to 245.93% and NDCG@k by 27.18% to 153.75%.

5.3.2 Application Performance at Inner-component Level. We compare LLMRCA with PDiagnose and Nezha, as only these baselines can identify root causes at the inner-component level. Table 8 shows the evaluation results for $\mathcal{AP}$. For single-type requests, PDiagnose and Nezha achieve HR@1 values of 16.67% and 21.43%. For multi-type requests, the HR@1 values are 11.9% and 19.05%.

First, baselines generally perform worse at the inner-component level than at the component level due to the finer granularity and more potential root causes. Second, similar to the component level, the effectiveness of baselines is generally lower for multi-type requests than for single-type requests, as varying response times interfere with diagnosis. Furthermore, baselines fail to leverage request context information, limiting their ability to identify root causes effectively. Among the baselines, Nezha performs best, since it extracts multimodal data using rule-based approaches and identifies root causes by cross-analyzing normal and fault conditions.

In conclusion, LLMRCA achieves the best effectiveness in identifying root causes of $\mathcal{AP}$ faults at the inner-component level. For single-type requests, LLMRCA achieves HR@1 of 65.16% and NDCG@3 of 75.21%. For multi-type requests, the HR@1 and NDCG@3 values of LLMRCA are 61.47% and 70.46%, respectively. Compared to baselines, LLMRCA improves HR@k by 90.36% to 416.55% and NDCG@k by 126.78% to 198.43%.

5.3.3 Response Quality. Table 9 shows the evaluation results of LLMRCA in $\mathcal{RQ}$ for single-type and multi-type requests. LLMRCA achieves HR@1 and NDCG@3 scores of 92.86% and 97.36% for single-type requests, and 91.19% and 96.75% for multi-type requests. LLMRCA

Table 9. Effectiveness for response quality.

Approach	Single-type Requests					Multi-type Requests				
	HR@1	*HR@3*	*HR@5*	*NDCG@3*	*NDCG@5*	*HR@1*	*HR@3*	*HR@5*	*NDCG@3*	*NDCG@5*
LLMRCA w/o $\mathcal{V}$	64.29	100	100	86.82	86.82	65.65	93.95	99.93	82.42	84.92
LLMRCA w/o $\mathcal{R}$	78.57	92.86	100	87.58	90.35	75.93	99.5	100	90.54	90.75
LLMRCA	**92.86**	**100**	**100**	**97.36**	**97.36**	**91.19**	**100**	**100**	**96.75**	**96.75**

performs consistently well on both single-type and multi-type requests. Its effectiveness in diagnosing $\mathcal{RQ}$ faults is unaffected by unstable response times and maintains high accuracy across different request patterns.

5.3.4 Effectiveness of different fault types. Fig. 10 shows LLMRCA's effectiveness across different fault categories at the inner-component level for multi-type requests. LLMRCA achieves excellent performance for Application and Context faults, with HR@1 reaching 100% and 99.56% respectively. For Host and Component faults, performance remains strong with HR@1 values of 51.52% and 40%. Code faults show moderate performance with 75.08% HR@1. These results demonstrate LLMRCA's capability to address diverse fault types in LLM applications.

5.4 RQ2: Contribution of Multimodal Data and Different Modules

The ablation results for $\mathcal{AP}$ at the component and inner-component levels for both single-type and multi-type requests are shown in Table 7 and Table 8, while the results for $\mathcal{RQ}$ are shown in Table 9. The variants of LLMRCA include: **LLMRCA w/o $\mathcal{M}$ (drops the metrics), LLMRCA w/o $\mathcal{L}$ (drops the logs), LLMRCA w/o $\mathcal{T}_c$ (drops the trace contextual data), LLMRCA w/o $\mathcal{V}$ (drops the hierarchical verifier module), LLMRCA w/o $\mathcal{R}$ (drops the ResGAT residual module), and LLMRCA w/o C (drops the classifier module)**. For $\mathcal{RQ}$ faults, we only use contextual data to identify root causes, so the contributions of multimodal data and the classifier module are not analyzed.

The results show that each data modality and module contributes to the effectiveness of LLMRCA. First, LLMRCA performs best when all multimodal data and modules are included. Second, the metrics contribute the most to effectiveness, followed by trace contexts and logs. This is because fault injection introduced more system resource faults than code defects or request context issues in our dataset. Third, contextual data (e.g., selected models, API parameters) is crucial for fault diagnosis in LLM applications, impacting both $\mathcal{AP}$ and $\mathcal{RQ}$. Fourth, logs have the least impact in our dataset, but this does not indicate that the contribution of logs is low. Since reproducing code region faults is challenging, the number of fault injections is limited. And logs are critical for engineers to troubleshoot faults in practical production. Finally, the hierarchical verifier reduces false positives, and dropping it decreases the HR@1 by up to 30.77%. The residual module improves the robustness of the GAT and enhances effectiveness in both $\mathcal{AP}$ and $\mathcal{RQ}$, and dropping it decreases the HR@1 by up to 16.73%. The classifier module addresses unstable response times in multi-type requests, and dropping it decreases HR@1 by 19.81%.

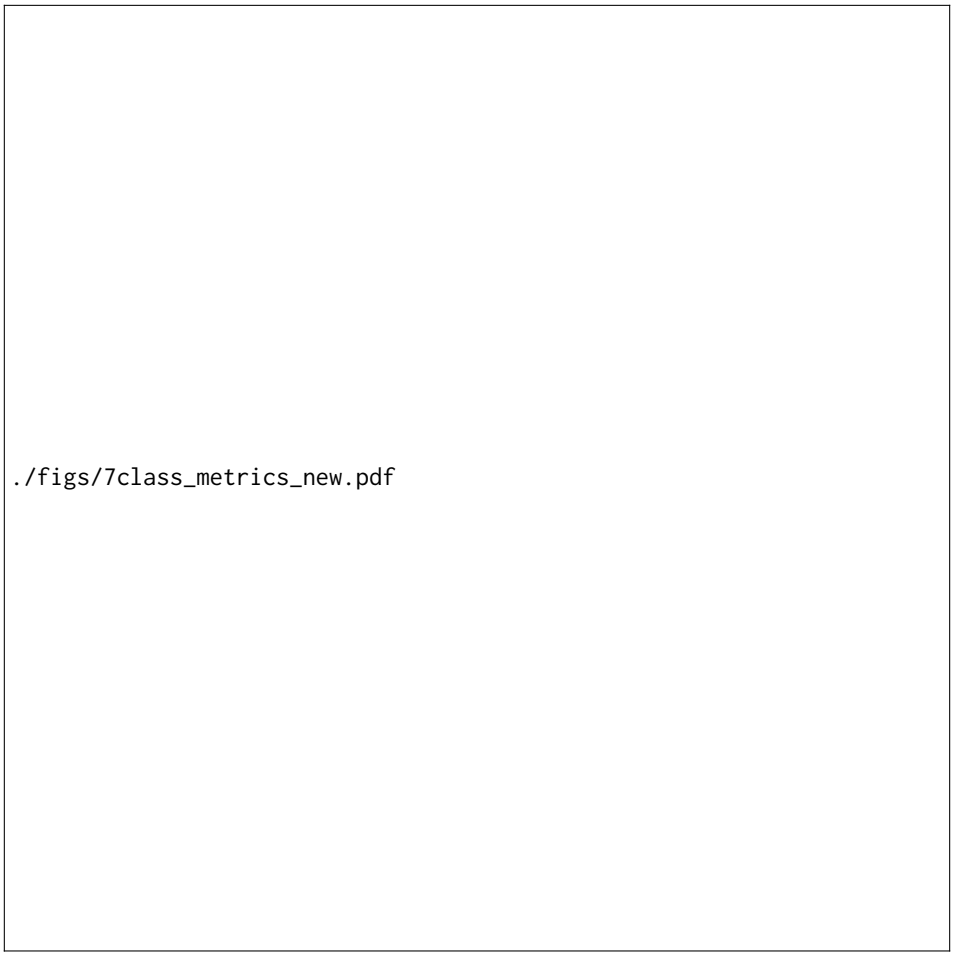


Fig. 10. Effectiveness of LLMRCA on different fault types.

5.5 RQ3: Impact of Configurations

As shown in Fig. 11 and Fig. 12, to evaluate the impact of configurations, we conduct experiments on multi-type requests, focusing on the inner-component level $\mathcal{AP}$ faults and $\mathcal{RQ}$ faults.

Configurations of ResGAT. Fig. 11 (a) shows the impact of the ResGAT Layers Number. LLMRCA performs better when the number of layers is set to 3 or 4. Too few layers limit model fitting, while too many layers reduce efficiency and harm performance. Fig. 11 (b) and (c) show the impact of the ResGAT Hidden Dimension and Latent Dimension. LLMRCA performs well with a Hidden Dimension of 32 or 128 and a Latent Dimension of 16 or 64. Lower dimensions restrict the ability to extract key features, causing information loss and underfitting. Higher dimensions enable learning more complex features, but may lead to overfitting, and increase model complexity.

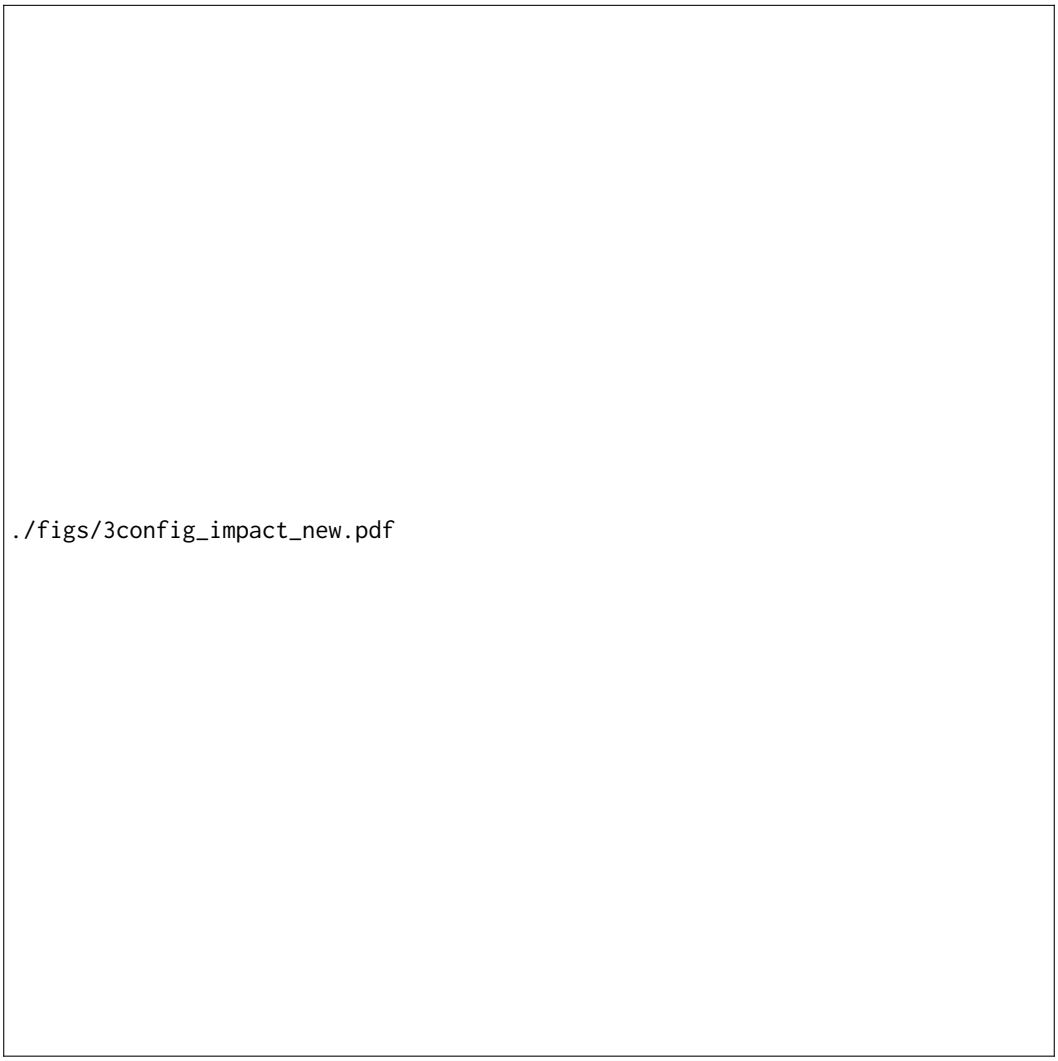


Fig. 11. Impact of hyperparameters: (a) ResGAT layers, (b) ResGAT hidden dimension, (c) ResGAT latent dimension, and (d) number of classifier classes. (P: Performance, Q: Quality, Cin: Inner-Component, M: Multi-type).

Configurations of Classifier. Fig. 11 (d) shows the impact of Xgboost classification. LLMRCA performs best when the class number is set to 3–5. Too few classes cause interference among request types, disrupting model fitting, while too many classes reduce samples per class, leading to underfitting. Additionally, excessive classes with one-hot encoding create a sparse feature matrix, increasing computational complexity and hindering the model's ability to extract meaningful information.

Configurations of Graph Neural Network. Fig. 12 shows the performance of different graph neural networks, and GAT performs the best. Fig. 12 (a) and (b) show the HR@1 and NDCG@3 results for component-level and inner-component-level RCA of $\mathcal{AP}$ faults for multi-type requests. Compared to GAT, the HR@1 and NDCG@3 of GCN decrease by

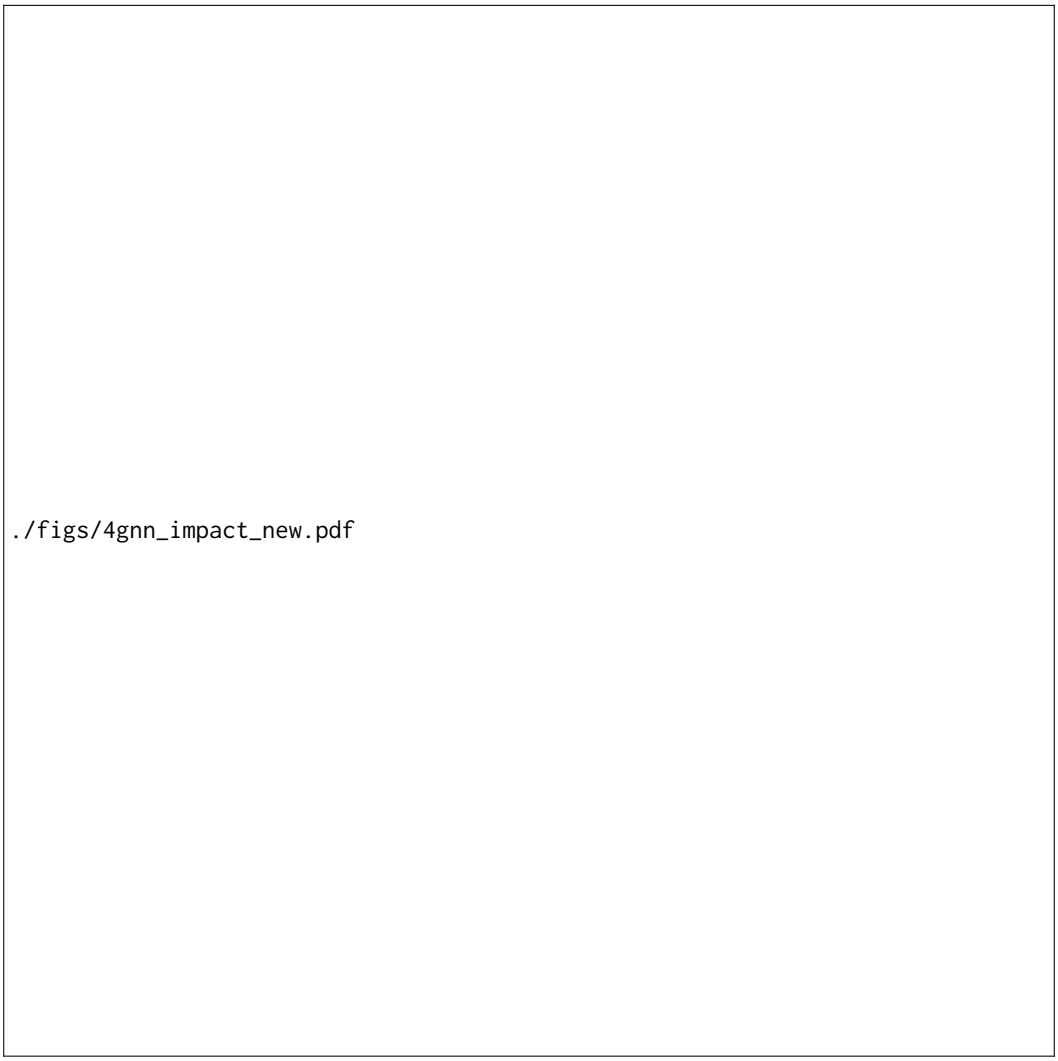


Fig. 12. Comparison of different graph neural network models.

6.93% and 6.96% on average, while those of GraphSAGE decrease by 2.04% and 1.7%. Fig. 12 (c) and (d) show the HR@1 and NDCG@3 results for RCA of $\mathcal{RQ}$ faults for multi-type requests. Compared to GAT, the HR@1 and NDCG@3 of GCN decrease by 29.79% and 16% on average, while those of GraphSAGE decrease by 2.04% and 1.69%. These results show that GAT outperforms GCN and GraphSAGE due to its attention mechanism, which better captures and weights the relationships between nodes in the graph. In contrast, GCN and GraphSAGE use static neighborhood aggregation, which is less effective for complex and dynamic relationships.

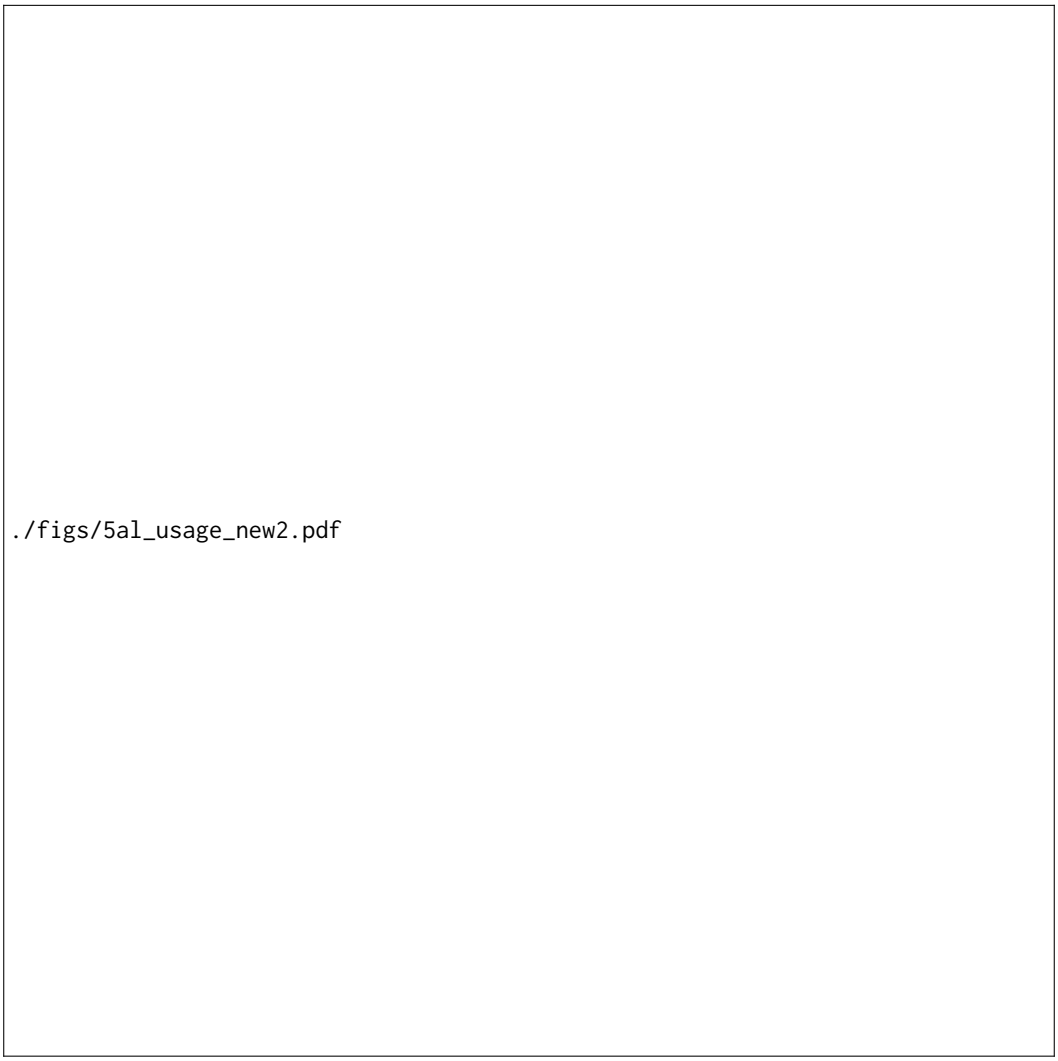


Fig. 13. Resource usage comparison of methods.

5.6 RQ4: Efficiency and Scalability

The efficiency and scalability of LLMRCA are evaluated using a dataset of 100 training requests and 300 test requests for multi-type request scenarios. The evaluation covers resource usage efficiency and system scalability across different graph sizes.

Efficiency. We evaluate LLMRCA's efficiency by comparing resource usage with baseline methods. As shown in Fig. 13, LLMRCA demonstrates balanced performance with a runtime of 14.76s and a memory usage of 411.19MB. Lightweight methods like Nezha achieve faster runtime (8.48s) and lower memory usage (156.07MB), while resource-intensive methods like FaasRCA require similar runtime (15.18s) but higher memory (492.71MB). LLMRCA achieves better accuracy-efficiency trade-offs than most baselines. The disk usage is small

(1.72MB), indicating efficient storage requirements. Importantly, LLMRCA requires training only once. The training overhead is a one-time cost, while the inference time for actual RCA tasks is much lower. As shown in Fig. 14 and the gray area of Fig 13, inference time accounts for only 11.19% of the total runtime at 40 nodes and 23.63% at 240 nodes, making it highly efficient for production deployment.

Scalability. We evaluate LLMRCA's scalability by increasing the graph size from 40 nodes to 240 nodes to measure how computational needs change as the system scale increases. Fig. 14 demonstrates LLMRCA's scalability across different graph sizes. Training time shows sub-linear growth from 14s (40 nodes) to 18.4s (240 nodes), demonstrating excellent scalability with only 31.43% increase for 6× more nodes. Inference time increases linearly from 1.76s to 7.02s, which remains practical for real-time diagnostics. Memory usage grows predictably from 411.19MB to 504.08MB, while disk usage remains stable around 1-4MB across all scales. Besides, we discuss production considerations under varying loads. Experiments with varying sample loads show training time grows approximately linearly (12.8s for 100 samples, 42.19s for 300 samples, 70.74s for 500 samples) while inference time scales linearly with sample load (1.22s for 100 requests, 1.45s for 200 requests, 1.96s for 300 requests). This identifies inference as the primary bottleneck. For deployment, we recommend: (1) deploying parallel LLMRCA instances for high throughput, (2) reduced ResGAT configurations for resource-constrained environments, and (3) retraining only on configuration changes rather than workload variations.

6 Case Study

6.1 Root Cause Analysis Case

We now demonstrate the performance of LLMRCA on a specific query instance to illustrate its application to a concrete root cause analysis. Fig. 15 shows a simplified case of how LLMRCA performs RCA when an application fault (Query) occurs. ResGAT computes Z-scores for the heterogeneous causal graph, with higher Z-scores indicating a higher likelihood of being the root causes. A hierarchical verifier is then applied. First, for inner-component level root causes, they must be leaf nodes and have at least one path to the root node where all intermediate nodes are abnormal ($Z_{ij} > 1$). For example, the anomalous path **Query → LLM Request → LLM Decoder** is identified, while false positive nodes like `gpu1_graph_clock` are filtered out. The top 5 root causes are ranked by Z-scores. Second, at the component level, ancestor nodes of inner-component root causes are ranked first, followed by other nodes. This process ensures accurate identification of root causes while filtering out false-positive nodes.

6.2 End-to-End Case

To show how LLMRCA works in practice, we present a simple example of diagnosing a prompt fault that affects both performance and quality, as shown in Fig. 16.

Fault Scenario. A user asks, "When is the premiere of 'Carole King & James Taylor: Just Call Out My Name'?" The system has a wrong prompt template that tells the LLM to write 300 words instead of a short answer. This makes the response slow (5x longer time) and of poor quality (too long and unfocused). Details of the query content are shown in Fig. 6.

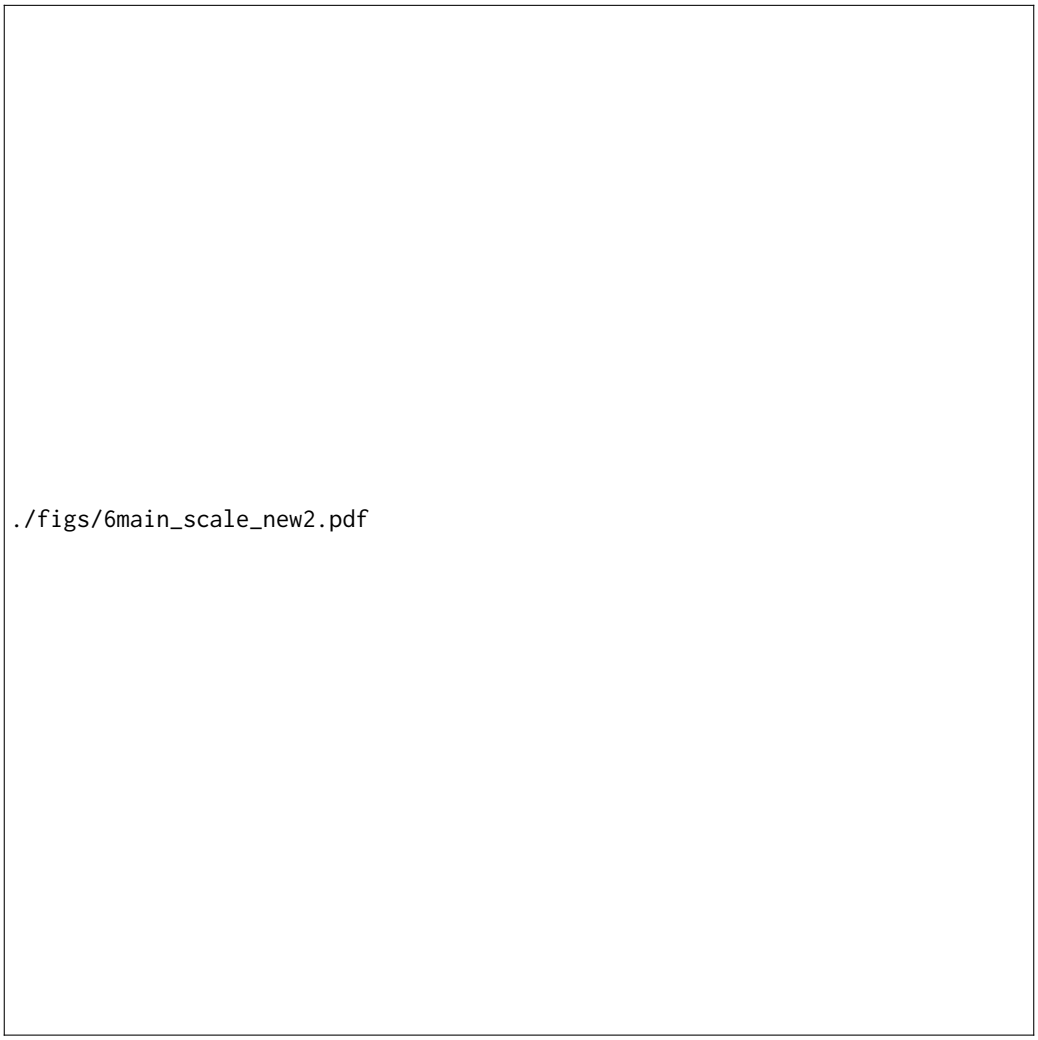


Fig. 14. Scalability with different graph sizes.

Offline Training Phase. First, we train an XGBoost regression model using data from normal requests. The model learns to predict a request's normal response time based on its input and output token counts. Next, we group normal requests into bins based on their predicted response times using equal-interval binning (e.g., 0-1.5s, 1.5-3.0s). For each bin, we build causal graphs from its requests and train a specific graph model. This creates a set of pre-trained models ready for diagnosis.

Online Diagnosis Phase. When an abnormal request arrives, the diagnosis proceeds as follows:

- **Data Collection.** We collect its observability data. Metrics show CPU usage jumps to 85% (normal: 45%), response time increases to 4.2s (normal: 0.8s), and output tokens reach 298 (normal: 50-80). Logs show the wrong prompt: "Please provide

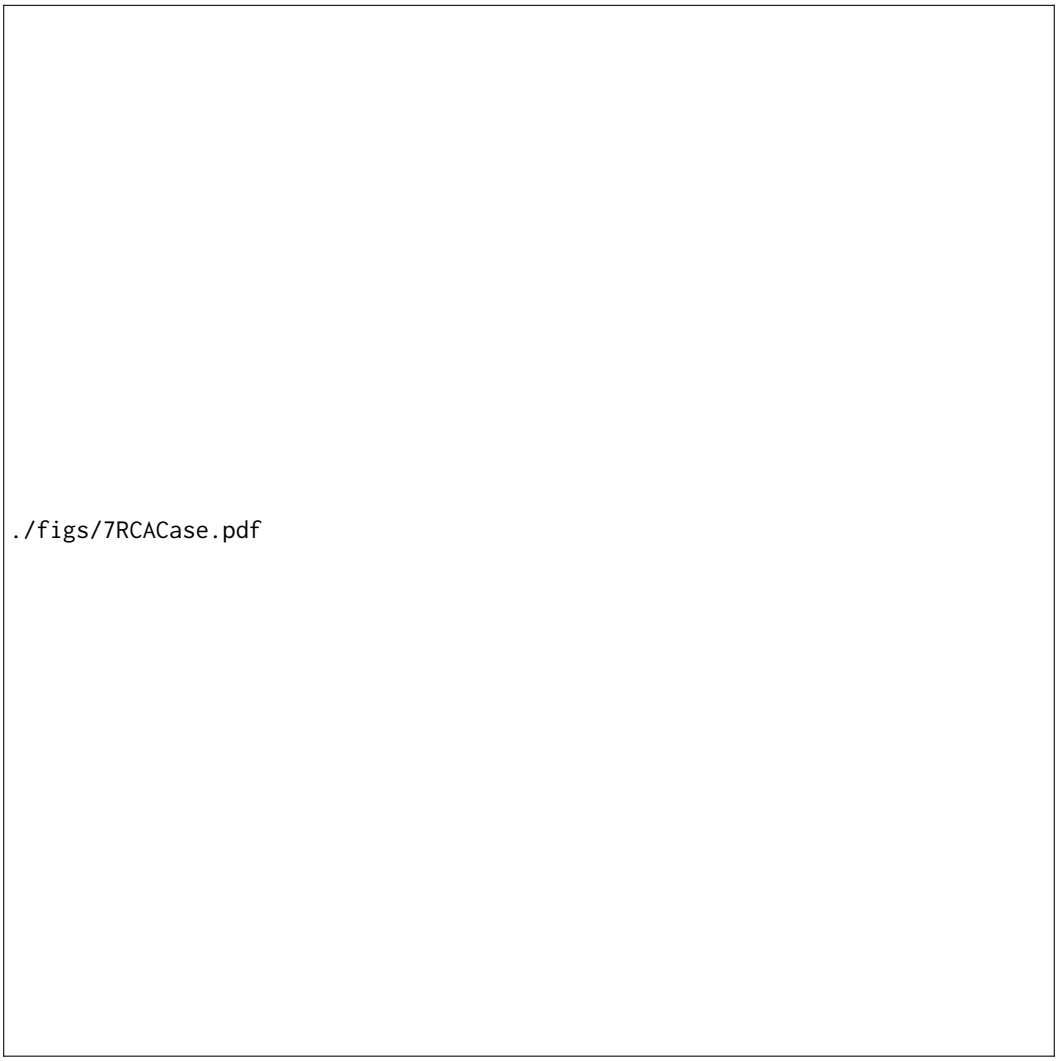


Fig. 15. Root cause analysis case.

a comprehensive analysis in approximately 300 words" (should be: "Answer in max two sentences"). Traces show the prompt instruction mismatch score of 0.77 (calculated as 1 - cosine similarity between expected and actual prompts), completion tokens of 298, and response time of 4.2s.

- **Classification.** The pre-trained XGBoost model predicts that this request, based on its 298 output tokens and input tokens, should have taken only 0.9s. Based on this predicted time, the request is routed to the "0-1.5s" bin.
- **Causal Graph Construction.** We construct a causal graph for the abnormal request. Then, we use the pre-trained graph model for the "0-1.5s" bin to calculate the reconstruction error and Z-score for each node in the graph. The analysis finds that

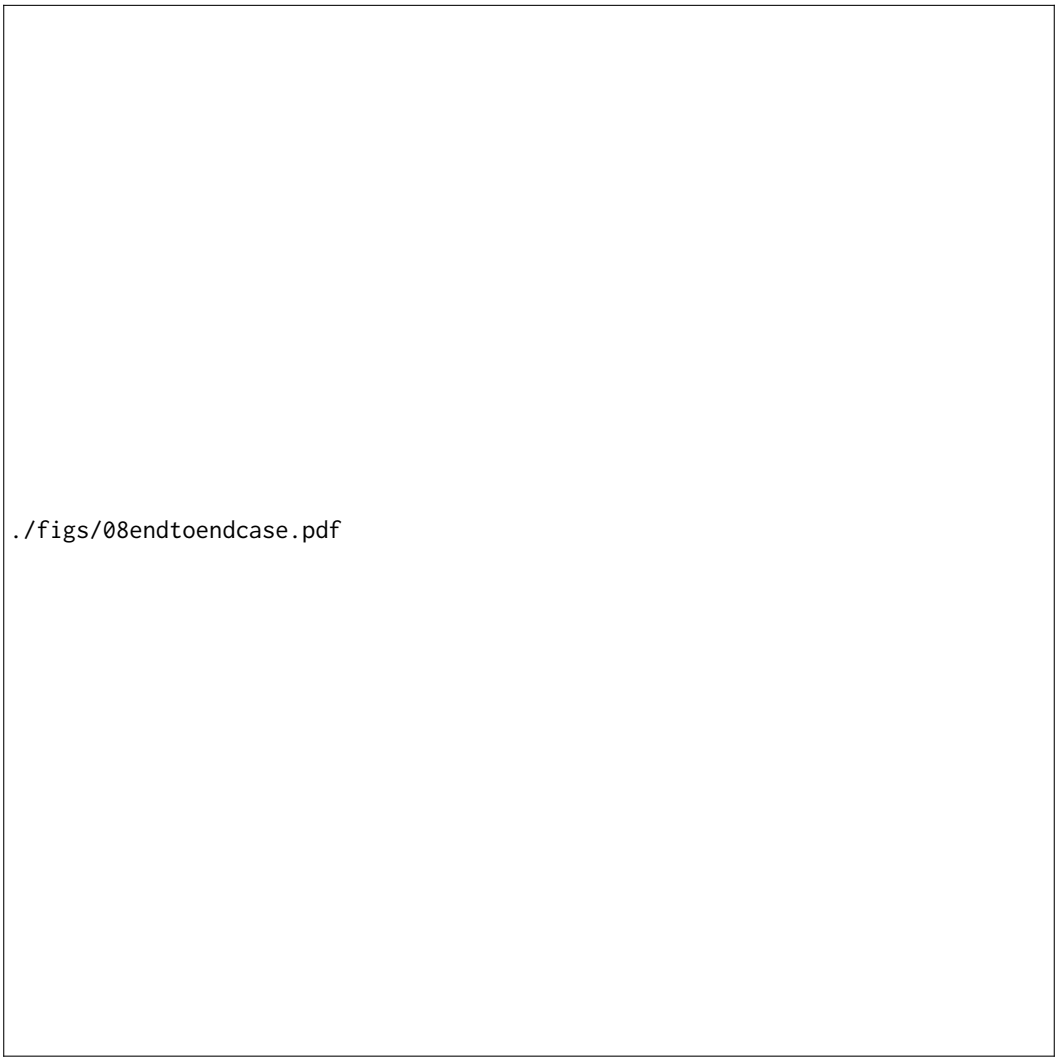


Fig. 16. End-to-end case.

the "prompt mismatch score" node has the highest Z-score (8.7), followed by "token count" (6.2).

- **Root Cause Analysis.** The verifier applies verification principles to analyze the Z-scores, like in Fig. 15: (1) inner-component: identifies the prompt mismatch score as a problem node; (2) anomaly propagation path: Query → LLM Request → prompt mismatch; (3) component: ranks LLM Request as the component-level root cause. As a result, we give a clear diagnosis report that the prompt misconfiguration causes both performance and quality degradation.

7 DISCUSSION

7.1 Limitations and Future Work

LLMRCA shows significant improvements in RCA for RAG-enhanced LLM applications, but there are several limitations that require further investigation. (1) Converting multimodal observability data into a unified time series format may result in information loss, as the unique characteristics of each modality might not be fully preserved. (2) Our current quality evaluation metrics may not capture all aspects of LLM response quality degradation, including the inability to assess factual accuracy, logical reasoning errors, or contextual appropriateness that require deeper semantic understanding. (3) LLMRCA's real-time diagnostic capabilities are limited by its computational requirements, which may affect its applicability in large-scale production environments.

Future work will focus on four directions: (1) developing better multimodal data fusion techniques that preserve modality-specific characteristics while enabling integrated analysis; (2) developing model-based evaluators or LLM-as-a-judge approaches to better diagnose response quality faults; (3) improving the real-time capabilities of the framework to enable automated fault recovery through more efficient graph processing algorithms; and (4) extending LLMRCA to diverse LLM applications through a phased validation roadmap. Short-term: test on RAG variants with different configurations to verify robustness. Mid-term: extend to agent-based applications by modeling components (planning, memory, tools) as causal graph nodes and analyzing new fault types (tool invocation failures, planning errors). Long-term: support multimodal LLM applications by extending data collection to capture visual and audio processing metrics and adapting causal graph construction. For new applications, apply LLMRCA through three steps: collect normal requests following Table 2, construct causal graphs, and retrain models with auto-generated partition points and apply Algorithms 1 and 2. The core code remains unchanged, requiring only configuration of data collection. Our findings also offer implications for practitioners and researchers. For practitioners, our work shows the value of adopting the enhanced observability data collection methods from Section 4.1. By collecting LLM-specific context, such as prompts and retrieved documents, teams can use LLMRCA to better diagnose both performance and quality failures in production. For researchers, our experiments in section 5.5 show that multimodal data fusion improves RCA accuracy. This suggests future work should develop more advanced fusion techniques and investigate how to construct causal graphs for complex LLM applications like multi-agent systems.

7.2 Threats to Validity

The internal validity of this study is affected by several factors: (1) Our fault injection approach is based on a systematic analysis of real-world LLM application faults in studies [40, 53], ensuring the representativeness of production scenarios, and follows established practices in RCA research [16, 51]. Some complex fault scenarios may not be included in our experiments. (2) LLMRCA's performance varies with different parameter settings. Though we conducted parameter sensitivity analyses, optimal configurations may produce different results.

The external validity is affected by several factors: (1) Our evaluation uses only one RAG-enhanced LLM application. However, we consider that this limitation is reduced because the benchmark system is representative of a broad class of LLM applications. As detailed in Section 5.1, its architecture and the set of failures we introduce are based on analysis of existing systems [40, 53]. Therefore, we determine that the results provide a strong foundation for RCA in similar LLM systems. (2) Our experimental environment differs from real production deployments in scale and infrastructure, which can affect the results when they are transferred to real-world settings. (3) Rapid LLM technology evolution poses external validity risks. New architectures may introduce novel fault modes (e.g., MoE model routing failures, reasoning model overthinking failures). Prompt engineering practices require updated quality metrics. Framework effectiveness needs validation on new architectures. We plan to mitigate these through periodic fault type library updates, extensible context node design for new metrics, and support for upgrading to LLM-as-a-judge methods.

8 CONCLUSION

We propose LLMRCA, a multilevel and unsupervised RCA framework designed for LLM applications. By leveraging multimodal observability data and constructing causal graphs, LLMRCA effectively identifies the root causes of application performance and response quality faults. The framework incorporates key modules such as the XGBoost classifier, the ResGAT model, and the hierarchical verifier to enhance accuracy and robustness. Experimental results show that LLMRCA significantly outperforms existing methods, achieving up to 5.1× more accurate results for performance faults and over 90% accuracy for quality faults. LLMRCA provides a comprehensive, interpretable, and scalable solution for improving the reliability of LLM applications in production environments.

Acknowledgments

This work was supported in part by National Key Research and Development Program of China (Grant Number: 2024YFB4505904), the National Natural Science Foundation of China under Grant 62272495 and the Guangdong Basic and Applied Basic Research Foundation under Grant 2023B1515020054.

References

[1] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM. doi:10.1145/3620665.3640366

[2] Arize. 2024. Phoenix. https://github.com/Arize-ai/phoenix. Accessed: 2025-04-12.

[3] Songlin Bao, Tiantian Li, and Bin Cao. 2024. Chain-of-event prompting for multi-document summarization by large language models. *Int. J. Web Inf. Syst.* 20, 3 (2024), 229–247. doi:10.1108/IJWIS-12-2023-0249

[4] Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. 2024. Benchmarking Large Language Models in Retrieval-Augmented Generation. In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan (Eds.). AAAI Press, 17754–17762. doi:10.1609/AAAI.V38I16.29728

[5] 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 *2014 IEEE Conference on Computer Communications,*

INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014. IEEE, 1887–1895. doi:10.1109/INFOCOM.2014.6848128

[6] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi (Eds.). ACM, 785–794. doi:10.1145/2939672.2939785

[7] DCGM. 2025. DCGM-Exporter. https://github.com/NVIDIA/dcgm-exporter. Accessed: 2025-04-12.

[8] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *CoRR* abs/2501.12948 (2025). doi:10.48550/ARXIV.2501.12948 arXiv:2501.12948

[9] Shenghui Gu, Guoping Rong, Tian Ren, He Zhang, Haifeng Shen, Yongda Yu, Xian Li, Jian Ouyang, and Chunan Chen. 2023. TrinityRCL: Multi-Granular and Code-Level Root Cause Localization Using Multiple Types of Telemetry Data in Microservice Systems. *IEEE Trans. Software Eng.* 49, 5 (2023), 3071–3088. doi:10.1109/TSE.2023.3241299

[10] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. 2020. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1387–1397. doi:10.1145/3368089.3417066

[11] 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 *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 931–943. doi:10.1145/3691620.3695475

[12] 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 2017, Honolulu, HI, USA, June 25-30, 2017*, Ilkay Altintas and Shiping Chen (Eds.). IEEE, 33–40. doi:10.1109/ICWS.2017.13

[13] Zilong He, Pengfei Chen, Yu Luo, Qiuyu Yan, Hongyang Chen, Guangba Yu, and Fangyuan Li. 2022. Graph based Incident Extraction and Diagnosis in Large-Scale Online Systems. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 48:1–48:13. doi:10.1145/3551349.3556904

[14] Chuanjia Hou, Tong Jia, Yifan Wu, Ying Li, and Jing Han. 2021. Diagnosing Performance Issues in Microservices with Heterogeneous Data Source. In *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), New York City, NY, USA, September 30 - Oct. 3, 2021*. IEEE, 493–500. doi:10.1109/ISPA-BDCLOUD-SOCIALCOM-SUSTAINCOM52081.2021.00074

[15] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* 33, 8 (2024), 220:1–220:79. doi:10.1145/3695988

[16] Jin Huang, Pengfei Chen, Guangba Yu, Yilun Wang, Haiyu Huang, and Zilong He. 2024. FaaSRCA: Full Lifecycle Root Cause Analysis for Serverless Applications. In *35th IEEE International Symposium on Software Reliability Engineering, ISSRE 2024, Tsukuba, Japan, October 28-31, 2024*. IEEE, 415–426. doi:10.1109/ISSRE62328.2024.00047

[17] Tao Huang, Pengfei Chen, Kyoka Gong, Jocky Hawk, Zachary Bright, Wenxin Xie, Kecheng Huang, and Zhi Ji. 2024. ENOVA: Autoscaling towards Cost-effective and Stable Serverless LLM Serving. *CoRR* abs/2407.09486 (2024). doi:10.48550/ARXIV.2407.09486 arXiv:2407.09486

[18] Myunghwan Kim, Roshan Sumbaly, and Sam Shah. 2013. Root cause detection in a service-oriented architecture. In *ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '13, Pittsburgh, PA, USA, June 17-21, 2013*, Mor Harchol-Balter, John R. Douceur, and Jun Xu (Eds.). ACM, 93–104. doi:10.1145/2465529.2465753

[19] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

[20] LangChain. 2022. LangChain. https://github.com/langchain-ai/langchain. Accessed: 2025-04-12.

[21] JinJin Lin, Pengfei Chen, and Zibin Zheng. 2018. Microscope: Pinpoint Performance Issues with Causal Graphs in Micro-service Environments. In *Service-Oriented Computing - 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11236)*, Claus Pahl, Maja Vukovic, Jianwei Yin, and Qi Yu (Eds.). Springer, 3–20. doi:10.1007/978-3-030-03596-9_1

[22] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log clustering based problem identification for online service systems. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, Laura K. Dillon, Willem Visser, and Laurie A.

Williams (Eds.). ACM, 102–111. doi:10.1145/2889160.2889232
[23] Jerry Liu. 2022. LlamaIndex. https://github.com/jerryjliu/llama_index. Accessed: 2025-04-12.
[24] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, and Dan Pei. 2020. Unsupervised Detection of Microservice Trace Anomalies through Service-Level Deep Bayesian Networks. In *31st IEEE International Symposium on Software Reliability Engineering, ISSRE 2020, Coimbra, Portugal, October 12-15, 2020*, Marco Vieira, Henrique Madeira, Nuno Antunes, and Zheng Zheng (Eds.). IEEE, 48–58. doi:10.1109/ISSRE5003.2020.00014
[25] 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 *28th IEEE/ACM International Symposium on Quality of Service, IWQoS 2020, Hangzhou, China, June 15-17, 2020*. IEEE, 1–10. doi:10.1109/IWQOS49365.2020.9213058
[26] Thanh-Hoang Nguyen-Vo, Trang T. T. Do, and Binh P. Nguyen. 2024. ResGAT: Residual Graph Attention Networks for molecular property prediction. *Memetic Comput.* 16, 3 (2024), 491–503. doi:10.1007/S12293-024-00423-5
[27] Kaiwen Ning, Jiachi Chen, Jingwen Zhang, Wei Li, Zexu Wang, Yuming Feng, Weizhe Zhang, and Zibin Zheng. 2024. Defining and Detecting the Defects of the Large Language Model-based Autonomous Agents. *CoRR* abs/2412.18371 (2024). doi:10.48550/ARXIV.2412.18371 arXiv:2412.18371
[28] NVIDIA. 2025. CUDA Toolkit. https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html. Accessed: 2025-04-12.
[29] OpenAI. 2024. ChatGPT hit by outage. https://status.openai.com/. Accessed: 2025-04-12.
[30] OpenAI. 2025. ChatGPT. https://chatgpt.com/. Accessed: 2025-04-12.
[31] OpenTelemetry. 2025. OpenTelemetry Collector. https://opentelemetry.io/docs/collector. Accessed: 2025-04-12.
[32] Luan Pham, Huong Ha, and Hongyu Zhang. 2024. Root Cause Analysis for Microservice System based on Causal Inference: How Far Are We?. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 706–715. doi:10.1145/3691620.3695065
[33] Prometheus. 2025. Prometheus Metric Exporter. https://prometheus.io/docs/instrumenting/exporters. Accessed: 2025-04-12.
[34] psutil. 2025. About Cross-platform lib for process and system monitoring in Python. https://github.com/giampaolo/psutil. Accessed: 2025-04-12.
[35] pynvml. 2025. Provide Python access to the NVML library for GPU diagnostics. https://github.com/gpuopenanalytics/pynvml. Accessed: 2025-04-12.
[36] Qdrant. 2025. Qdrant: High-performance, massive-scale Vector Database and Vector Search Engine for the next generation of AI. https://github.com/qdrant/qdrant. Accessed: 2025-04-12.
[37] Dongyu Ru, Lin Qiu, Xiangkun Hu, Tianhang Zhang, Peng Shi, Shuaichen Chang, Cheng Jiayang, Cunxiang Wang, Shichao Sun, Huanyu Li, Zizhao Zhang, Binjie Wang, Jiarong Jiang, Tong He, Zhiguo Wang, Pengfei Liu, Yue Zhang, and Zheng Zhang. 2024. RAGChecker: A Fine-grained Framework for Diagnosing Retrieval-Augmented Generation. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/27245589131d17368cccdfa990cbf16e-Abstract-Datasets_and_Benchmarks_Track.html
[38] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. 2024. A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications. *CoRR* abs/2402.07927 (2024). doi:10.48550/ARXIV.2402.07927 arXiv:2402.07927
[39] Shreya Shankar and Aditya G. Parameswaran. 2022. Towards Observability for Production Machine Learning Pipelines [Vision]. *Proc. VLDB Endow.* 15, 13 (2022), 4015–4022. doi:10.14778/3565838.3565853
[40] Yuchen Shao, Yuheng Huang, Jiawei Shen, Lei Ma, Ting Su, and Chengcheng Wan. 2024. Vortex under Ripplet: An Empirical Study of RAG-enabled Applications. *CoRR* abs/2407.05138 (2024). doi:10.48550/ARXIV.2407.05138 arXiv:2407.05138
[41] Shivanshu Shekhar, Tanishq Dubey, Koyel Mukherjee, Apoorv Saxena, Atharv Tyagi, and Nishanth Kotla. 2024. Towards Optimizing the Costs of LLM Usage. *CoRR* abs/2402.01742 (2024). doi:10.48550/ARXIV.2402.01742 arXiv:2402.01742
[42] Gaurang Sriramanan, Siddhant Bharti, Vinu Sankar Sadasivan, Shoumik Saha, Priyatham Kattakinda, and Soheil Feizi. 2024. LLM-Check: Investigating Detection of Hallucinations in Large Language Models. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/3c1e1fdf305195cd620c118aaa9717ad-Abstract-Conference.html
[43] Traceloop. 2024. OpenLLMetry. https://github.com/traceloop/openllmetry. Accessed: 2025-04-12.

[44] Hanzhang Wang, Zhengkai Wu, Huai Jiang, Yichao Huang, Jiamu Wang, Selçuk Köprü, and Tao Xie. 2021. Groot: An Event-graph-based Approach for Root Cause Analysis in Industrial Settings. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 419–429. doi:10.1109/ASE51524.2021.9678708

[45] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024. A survey on large language model based autonomous agents. *Frontiers Comput. Sci.* 18, 6 (2024), 186345. doi:10.1007/S11704-024-40231-1

[46] Thorsten Wittkopp, Philipp Wiesner, and Odej Kao. 2024. LogRCA: Log-Based Root Cause Analysis for Distributed Services. In *Euro-Par 2024: Parallel Processing - 30th European Conference on Parallel and Distributed Processing, Madrid, Spain, August 26-30, 2024, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 14802)*, Jesús Carretero, Sameer Shende, Javier García-Blas, Ivona Brandic, Katzalin Olcoz, and Martin Schreiber (Eds.). Springer, 362–376. doi:10.1007/978-3-031-69766-1_25

[47] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Perric Cistac, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. Association for Computational Linguistics, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[48] Li Wu, Johan Tordsson, Jasmin Bogatinovski, Erik Elmroth, and Odej Kao. 2021. Microdiag: Fine-grained performance diagnosis for microservice systems. In *2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence)*. IEEE, 31–36.

[49] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. MicroRCA: Root Cause Localization of Performance Issues in Microservices. In *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020*. IEEE, 1–9. doi:10.1109/NOMS47738.2020.9110353

[50] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. 2021. MicroRank: End-to-End Latency Issue Localization with Extended Spectrum Analysis in Microservice Environments. In *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia (Eds.). ACM / IW3C2, 3087–3098. doi:10.1145/3442381.3449905

[51] Guangba Yu, Pengfei Chen, Yufeng Li, Hongyang Chen, Xiaoyun Li, and Zibin Zheng. 2023. Nezha: Interpretable Fine-Grained Root Causes Analysis for Microservices on Multi-modal Observability Data. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 553–565. doi:10.1145/3611643.3616249

[52] Guangba Yu, Zicheng Huang, and Pengfei Chen. 2023. TraceRank: Abnormal service localization with dis-aggregated end-to-end tracing data in cloud native systems. *J. Softw. Evol. Process.* 35, 10 (2023). doi:10.1002/SMR.2413

[53] Guangba Yu, Gou Tan, Haojia Huang, Zhenyu Zhang, Pengfei Chen, Roberto Natella, and Zibin Zheng. 2024. A Survey on Failure Analysis and Fault Injection in AI Systems. *CoRR* abs/2407.00125 (2024). doi:10.48550/ARXIV.2407.00125 arXiv:2407.00125

[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 Trans. Serv. Comput.* 16, 6 (2023), 3851–3864. doi:10.1109/TSC.2023.3290018

[55] Zangwei Zheng, Xiaozhe Ren, Fuzhao Xue, Yang Luo, Xin Jiang, and Yang You. 2023. Response Length Perception and Sequence Scheduling: An LLM-Empowered LLM Inference Pipeline. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/ce7ff3405c782f761fac7f849b41ae9a-Abstract-Conference.html