

Flow-of-Action: SOP Enhanced LLM-Based Multi-Agent System for Root Cause Analysis

Changhua Pei*

chpei@cnic.cn

Computer Network Information
Center, Chinese Academy of Sciences
Beijing, China

Zexin Wang†‡

wangzexin@cnic.cn

Computer Network Information
Center, Chinese Academy of Sciences
Beijing, China

Fengrui Liu

Zeyan Li

liufengrui.work@bytedance.com
lizeyan.42@bytedance.com
ByteDance
Beijing, China

Yang Liu†

liuyang@cnic.cn

Computer Network Information
Center, Chinese Academy of Sciences
Beijing, China

Xiao He

xiao.hx@bytedance.com

ByteDance
Hangzhou, China

Rong Kang

kangrong.cn@bytedance.com

ByteDance
Beijing, China

Tieying Zhang§

Jianjun Chen

tieying.zhang@bytedance.com
jianjun.chen@bytedance.com
ByteDance
San Jose, United States

Jianhui Li§

Gaogang Xie
lijh@cnic.cn
xie@cnic.cn

Computer Network Information
Center, Chinese Academy of Sciences
Beijing, China

Dan Pei

peidan@tsinghua.edu.cn
Tsinghua University
Beijing, China

Abstract

In the realm of microservices architecture, the occurrence of frequent incidents necessitates the employment of Root Cause Analysis (RCA) for swift issue resolution. It is common that a serious incident can take several domain experts hours to identify the root cause. Consequently, a contemporary trend involves harnessing Large Language Models (LLMs) as automated agents for RCA. Though the recent ReAct framework aligns well with the Site Reliability Engineers (SREs) for its thought-action-observation paradigm, its hallucinations often lead to irrelevant actions and directly affect subsequent results. Additionally, the complex and variable clues of the incident can overwhelm the model one step further. To confront these challenges, we propose **Flow-of-Action**, a pioneering Standard Operation Procedure (SOP) enhanced LLM-based multi-agent system. By explicitly summarizing the diagnosis steps of SREs, SOP imposes constraints on LLMs at crucial junctures, guiding the RCA process towards the correct trajectory. To facilitate the rational and effective utilization of SOPs, we design an SOP-centric framework

called **SOP flow**. SOP flow contains a series of tools, including one for finding relevant SOPs for incidents, another for automatically generating SOPs for incidents without relevant ones, and a tool for converting SOPs into code. This significantly alleviates the hallucination issues of ReAct in RCA tasks. We also design multiple auxiliary agents to assist the main agent by removing useless noise, narrowing the search space, and informing the main agent whether the RCA procedure can stop. Compared to the ReAct method's 35.50% accuracy, our Flow-of-Action method achieves 64.01%, meeting the accuracy requirements for RCA in real-world systems.

CCS Concepts

- Software and its engineering → Software maintenance tools.

Keywords

Root Cause Analysis, Multi-Agent System, Large Language Model

ACM Reference Format:

Changhua Pei, Zexin Wang, Fengrui Liu, Zeyan Li, Yang Liu, Xiao He, Rong Kang, Tieying Zhang, Jianjun Chen, Jianhui Li, Gaogang Xie, and Dan Pei. 2025. Flow-of-Action: SOP Enhanced LLM-Based Multi-Agent System for Root Cause Analysis. In *Companion Proceedings of the ACM Web Conference 2025 (WWW Companion '25), April 28-May 2, 2025, Sydney, NSW, Australia*. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3701716.3715225>

*Also with Hangzhou Institute for Advanced Study, University of Chinese Academy of Sciences.

†Also with University of Chinese Academy of Sciences.

‡Also with ByteDance. Work done during the internship at ByteDance.

§Corresponding Authors.



This work is licensed under a Creative Commons Attribution 4.0 International License.
WWW Companion '25, April 28-May 2, 2025, Sydney, NSW, Australia

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1331-6/2025/04

<https://doi.org/10.1145/3701716.3715225>

1 Introduction

In today's large-scale web systems and services, traditional monolithic applications encounter notable challenges including intricate

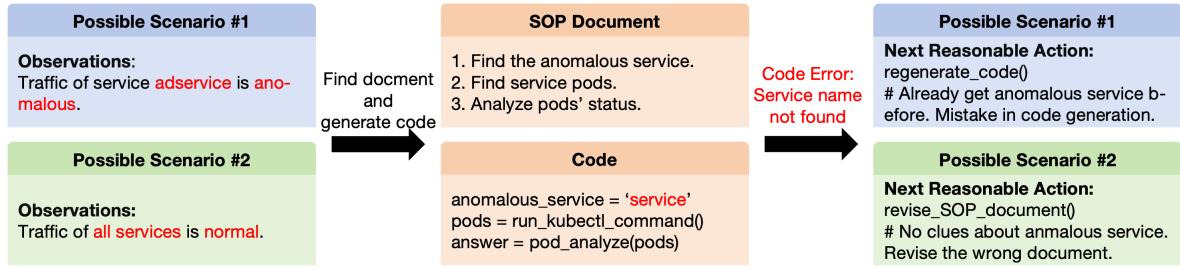


Figure 1: Illustration example of challenge 2.

deployment processes and limited scalability [1, 8, 24, 26, 37], attributed to the proliferation of services and frequent service iterations. In response to this context, Microservices Architecture (MSA) has surfaced and continually evolved [2]. By disassembling monolithic applications into small, self-sufficient service units, each dedicated to specific business functionalities, MSA presents benefits such as loose coupling, independent deployment, and effortless scalability. Nevertheless, with the escalation of user numbers and their corresponding demands, the diversity and quantity of MSA instances also increase. Despite the implementation of numerous monitor tools, recurrent incidents arise from hardware malfunctions or misconfigurations, posing challenges to reliability assurance. These incidents lead to substantial financial losses [27]. For instance, on November 12, 2023, Alibaba experienced a large-scale outage, resulting in the interruption of multiple services for nearly three hours¹.

To promptly tackle these incidents, Root Cause Analysis (RCA) has emerged as a prominent research area within Artificial Intelligence for IT Operations (AIOps) in recent years [4, 6, 13, 30]. Traditional RCA techniques, in order to address the difficulties of manual fault diagnosis, have employed deep learning methods to learn from historical faults [12]. However, these methods have two main drawbacks. First, they have poor adaptability to new scenarios, requiring model retraining when faced with a new situation. Second, they only output the root cause of the fault without providing the entire diagnostic process, resulting in poor explainability. This situation often results in Site Reliability Engineers (SREs) harboring a sense of distrust towards the results, as they fear that misidentifying the root cause could potentially result in further wasted repair time or exacerbate faults by addressing the wrong issue. Over the recent years, Large Language Model (LLM) agents like ReAct [32] and ToolFormer [21] have been deployed across diverse domains. LLM agents harness their robust natural language understanding capabilities to adeptly coordinate various tools, allowing SREs to see the entire troubleshooting process and providing rich explanations for the root causes. Nonetheless, despite the considerable prowess of LLM agents, the efficient and accurate utilization of LLM agents in RCA encounters ongoing challenges.

Challenge 1: Randomness and hallucinations leading to irrational action selection

Current LLMs primarily function as probabilistic models [17, 18], thereby exhibiting pronounced randomness and tendencies towards

generating hallucinations. Employing an LLM agent for RCA activities necessitates the retrieval and comprehension of diverse data modalities (metric [14], log [20], trace [34]) and the extensive utilization of API tools. As the scope of the context expands, issues often emerge such as inaccurate parameter extraction leading to failures in tool invocation and discrepancies between tool invocations and the context at hand. Instances of randomness or hallucinations at any stage can significantly impact the subsequent trajectory of the RCA procedure, hindering the accurate identification of the true root cause.

Challenge 2: Complex and variable observations leading to multiple reasonable actions

Existing LLM agents are typically bundled with a diverse array of tools [16], especially within complex domains like RCA, where the number of APIs can escalate to hundreds. Each API invocation results in varied observations, thereby introducing intricacies in action selection. Furthermore, even when confronted with identical observations, multiple plausible actions may be viable. For example, as shown in Figure 1, within the context of a code error “Service name not found”, the root cause could originate from errors in the code generation phase or inaccuracies in associated SOP document, prompting multiple feasible actions like code regeneration or document revision.

To confront the challenges outlined above, we propose **Flow-of-Action**, a Standard Operating Procedure (SOP) enhanced Multi-Agent System (MAS). Initially, to mitigate the impact of randomness and hallucinations in the orchestration process, we integrate SOPs into the knowledge base and propose the **SOP flow**. Specifically, SOPs outline a standardized set of steps for RCA, while SOP flow represents an efficient and accurate process built upon SOPs for their effective utilization. Through prompt engineering, we ensure that the orchestration of the main agent loosely follows the SOP flow in the absence of unexpected circumstances. Subsequently, to tackle the second challenge, compared with the thought-action-observation paradigm, we propose the thought-**actionset**-action-observation paradigm. Flow-of-Action avoids immediate action selection and instead generates a reasoned action set before making the final decision on the course of action. Besides, we devise a novel MAS. Specifically, we introduce multiple agents such as MainAgent, CodeAgent, JudgeAgent, ObAgent, and ActionAgent, each entrusted with distinct responsibilities, collaborating harmoniously to enhance root cause identification.

Our key contributions are summarized as follows:

¹<https://www.datacenterdynamics.com/en/news/alibaba-cloud-hit-by-outage-second-in-a-month/>

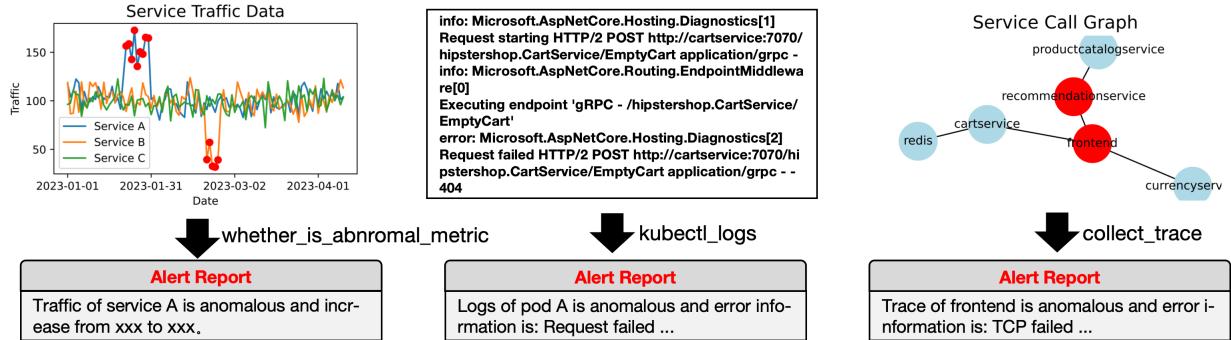


Figure 2: Multimodal data collection and analysis.

- We propose the Flow-of-Action framework, the first agent-based fault localization process centered around SOPs. With this framework, we significantly reduce the inefficiency in action selection of the native ReAct framework and reducing the cost of trial and error.
- We introduce the concept of SOPs to integrate the expert experience into the LLM to greatly reduce hallucinations during RCA. For any given fault, we can automatically match the most relevant set of SOPs and can also generate new SOPs automatically, extending the limited set of human-generated SOPs.
- We innovatively propose a multi-agent collaborative system, including JudgeAgent and ObAgent. JudgeAgent assists the MainAgent in determining whether the root cause of the fault has been identified in the current iteration, while ObAgent helps MainAgent extract fault types and key information from massive amounts of data, addressing the information overload issue in the RCA process.
- Through a fault-injection simulation platform of a real-world e-commerce system, Flow-of-Action has increased the localization accuracy from 35% to 64% compared to ReAct, proving the effectiveness of the Flow-of-Action framework.

2 Flow-of-Action

In this section, we will present the design of Flow-of-Action. As illustrated in Figure 3, the Flow-of-Action is a MAS built upon the ReAct. It encompasses three key design components: the SOP flow, the action set, and the MAS. We will delve into each of these components in the subsequent sections. Prior to their detailed exploration, we will introduce the foundational knowledge required, including the knowledge base and tools utilized by the Flow-of-Action.

2.1 Knowledge Base of Agents

Given the restricted context length of LLMs, Retrieval-Augmented Generation (RAG) has experienced notable progress [7]. However, the quality of text retrieved by RAG significantly influences the ultimate outcomes. Many existing RAG methodologies segment documents within the knowledge base and employ semantic block embeddings to calculate similarity for retrieval. This approach, however, does not consistently yield optimal results in RCA. Therefore,

we have devised an innovative knowledge base model integrating SOP knowledge and historical incident knowledge.

2.1.1 SOP Knowledge. With the successful integration of SOPs in the realm of code generation [5], there is a growing recognition that relying solely on LLMs to execute intricate tasks like RCA is impractical. SOPs, to a certain extent, impose constraints on LLMs at crucial junctures, guiding the entire process towards the correct trajectory. Consequently, we have embedded SOPs into the knowledge base, which are either authored by engineers based on domain expertise or extracted through automation tools. As shown in Figure 3, each SOP constitutes a self-contained unit comprising two attributes: name and steps. The name encapsulates essential information about the SOP, which is translated into a vector for subsequent retrieval purposes.

2.1.2 Historical Incidents. As highlighted by Chen et al. [3], in systems where similar incidents occur frequently, historical incident data proves invaluable in identifying the root cause of ongoing incidents. Consequently, we incorporate the performance details of historical incidents into the knowledge base. Each historical incident is characterized by two key attributes: manifestation and type. When retrieving similar incidents, we evaluate similarity by comparing the embedding of the current observation with the embedding of the manifestation of historical incidents. However, relying solely on embeddings for assessment can introduce significant errors. To tackle this issue, we have intentionally devised the ObAgent (elaborated upon subsequently) to address this challenge.

2.2 Tools of Agents

Within LLM agents, tools typically refer to pre-defined functions. During the action phase, LLM invokes relevant tools to obtain the necessary information. In Flow-of-Action, the tools utilized primarily fall into three categories: tools for multimodal data collection and analysis, tools related to SOP flow, and other tools. Each category will be discussed in detail below.

2.2.1 Multimodal Data Collection and Analysis. Within the realm of MSA, which encompasses diverse modalities of data such as metrics, traces, and logs, the importance of multimodal data for RCA has been underscored by existing methodologies [33, 35]. Consequently, we have implemented a comprehensive monitoring system

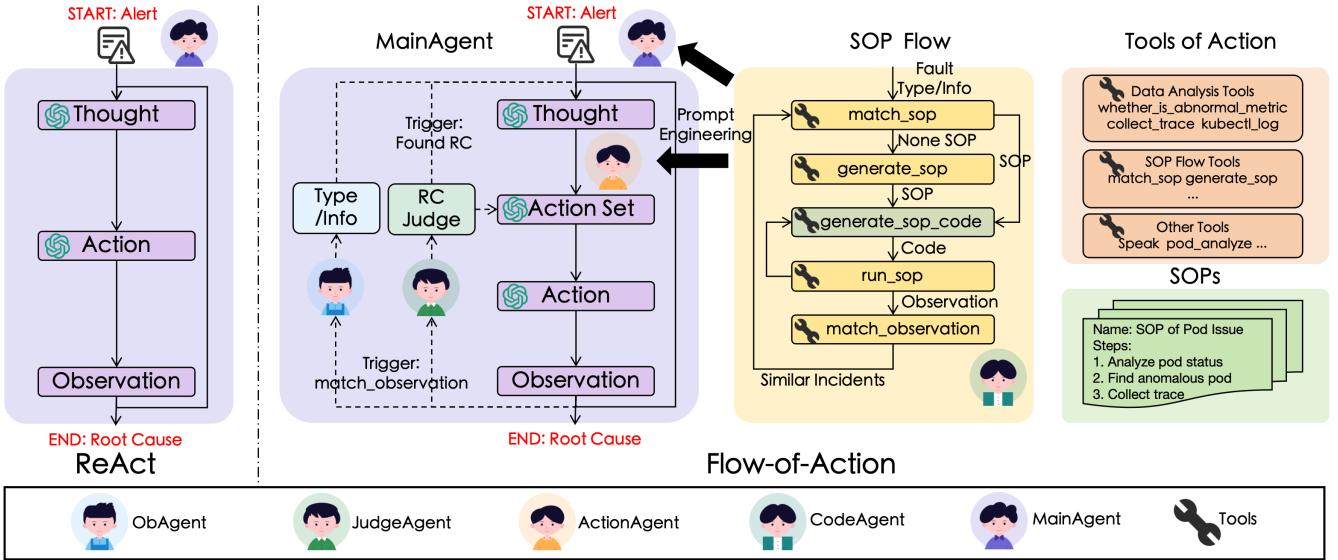


Figure 3: Comparison of ReAct and Flow-of-Action. RC means root cause. Dashed lines represent paths triggered under specific conditions. When the previous action is *match_observation*, JudgeAgent and ObAgent are triggered. When JudgeAgent finds the root cause, it triggers the input of the analysis result to thought and adds *Speak* to action set.

Table 1: Description of SOP flow tools.

	Input	Output	LLM Usage
match_sop	Fault Type/Information	SOP	No
generate_sop	Fault Type/Information	SOP	Yes
generate_sop_code	SOP	SOP Code	Yes
run_sop	SOP Code	Result after running the code	No
match_observation	Observation	Similar incidents	No

to aggregate multimodal data. While LLMs excel in processing textual data, their effectiveness in interpreting structured data types like metrics is constrained, especially in the presence of data noise. Therefore, it is imperative to preprocess the data by denoising and transforming it into textual format for enhanced comprehension by LLMs. As depicted in Figure 2, we have devised the following components: *whether_is_abnormal_metric* to leverage time series anomaly detection algorithms [25, 28, 31] for identifying metric anomalies and converting them into fault-related text; *collect_trace* for capturing abnormal span details across the entire call chain and converting them into text format; and *kubectl_logs* for extracting abnormal log information from each pod within the Kubernetes system.

2.2.2 SOP Flow Tools. As previously mentioned, we have introduced a flow centered around SOPs. This comprehensive flow is meticulously crafted based on common workflows employed by SREs in practical settings, integrating innovative concepts such as code. Details regarding the tools utilized within the flow are delineated in Table 1. Moreover, to preempt unexpected incidents during the flow's operation, we have developed a variety of targeted auxiliary tools. For example, within the context of *generate_sop*,

we have introduced *get_relevant_metric* to streamline the retrieval of pertinent metric names.

2.2.3 Other Tools. The flow aims to establish a standardized and generalized process for intricate RCA tasks, devoid of service- or business-specific components within the tools themselves. However, a broader array of tools is necessitated when generating SOPs or SOP code, or when executing operations beyond the flow, to query the authentic operational state of the system. In addition to the previously mentioned tools for querying and analyzing multimodal data, a suite of tailored analysis tools has been devised for MSA, including *pod_analyze* and *service_analyze*. These tools employ queries on specific attribute data within the Kubernetes system to ascertain the system's status. Upon identification, *Speak* is employed to communicate the discovered root cause to all pertinent stakeholders. For a comprehensive elucidation of these tools, kindly consult the Appendix B.

2.3 SOP Flow

The SOP flow represents a comprehensive logic chain of actions tailored to the SOP mentioned earlier. It serves to instruct LLMs on how to effectively utilize SOP knowledge. For instance, in the initial stages of RCA, it is essential to identify which SOPs are most

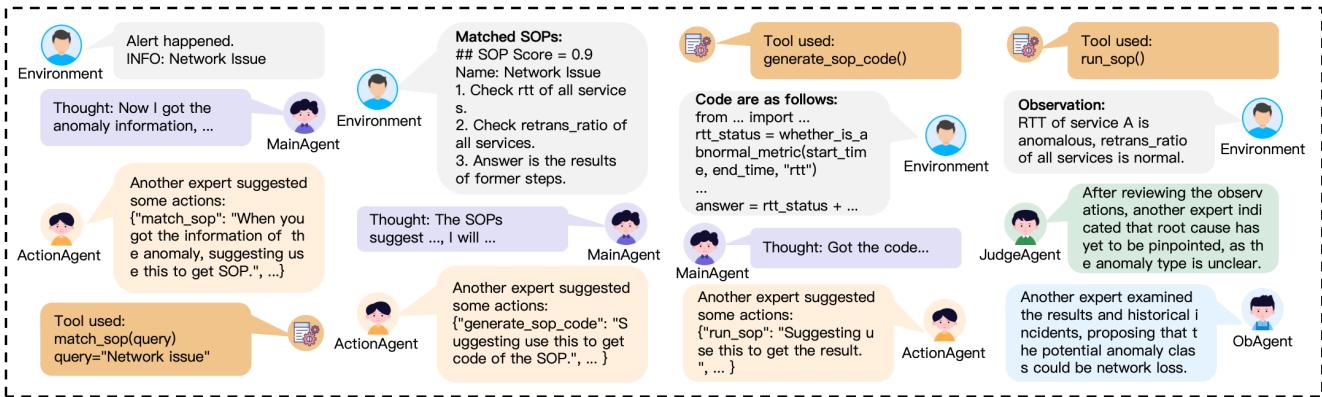


Figure 4: Example of Flow-of-Action.

relevant to the incident (corresponding to *match_sop*). Additionally, if a particular incident does not align with any existing SOP, the automation of SOP generation should be considered (corresponding to *generate_sop*). While the comprehensive SOP flow can be visually represented, as illustrated in Figure 3, in practical application, the full SOP flow is presented in the form of prompts to the MainAgent to aid in thought processes and to the ActionAgent to generate a more rational action set. The SOP flow prompt information provided to the LLMs is displayed in Figure 5. By implementing such soft constraints, we aim to tackle the issue of chaotic tool orchestration while still maintaining the flexibility of LLMs. Unlike methods like FastGPT [10], we do not enforce strict workflow constraints on LLM orchestration. Figure 4 provides an example of the Flow-of-Action. Subsequently, we will systematically elucidate critical transitional subflows within the SOP flow.

```
Rules and Format Instructions for Tool Using

if at the beginning and last action doesn't exist:
    next action should be match_sop
elif last action == match_sop:
    last observations are all matched SOPs
    next action should be generate_sop_code # Parameters: cause_name of the SOP document should be the unexecuted SOP with higher score, you shouldn't execute one SOP twice. If one SOP has been executed already, choose another one.
    if no SOPs matched or the SOPs are not relevant:
        next action should be generate_sop
elif last action == generate_sop_code:
    last observations are code
    next action should be run_sop
elif last action == run_sop:
    last observations are result after running code
    if some error happened:
        next action should be generate_sop_code # regenerate the right code
    else:
        next action should be match_observation # Parameters: the query should be the whole original observation without any delete
elif last action == match_observation:
    last observations are possible anomaly class
    next action should be match_sop # match SOP of the possible anomaly class
elif last action == generate_sop:
    last observation is the new SOPs you got.
    next action should be generate_sop_code to generate the code
```

Figure 5: SOP flow prompt information of LLMs.

2.3.1 Fault Type/Information→SOP. In our flow, we initially utilize *match_sop* to associate the fault information with the relevant SOP. This matching process involves computing the similarity between the current query and all SOP name embeddings, ranking them, and selecting the top k matches. To avoid matching with highly irrelevant SOPs, a filtering threshold is established. Nevertheless, in real-world contexts where new fault types frequently emerge,

instances may arise where pertinent SOPs cannot be matched. To tackle this challenge, we introduce *generate_sop* to devise new SOPs for queries that do not align with existing SOPs. Specifically, we utilize LLMs to generate new SOPs and leverage existing SOPs as few-shot prompts to guide the development of more standardized and coherent SOPs. Figure 6 shows an example of an SOP for handling IO errors generated by an LLM. Although not highly precise, the general direction of analysis is correct. The logic is rigorous, and the diagnostic results provide useful assistance to SREs.

Generated SOP for IO Error

- get_relevant_metric:** Get the relevant metrics related to the anomaly.
- whether_is_abnormal_metric:** Check if the IO metrics are abnormal.
- collect_trace:** Collect trace data for the anomalous service to investigate further. (`start_time`, `end_time`, `servicename`)
- The answer is the observations obtained from former steps.

Figure 6: Generated SOP for IO error.

Within the entirety of the flow, the generation of SOPs stands as a pivotal phase as it directly influences the subsequent RCA process. To enhance the precision of RCA, we have devised hierarchical SOPs. Our objective is for the RCA process to progress from a macro to micro level, from a general to specific perspective, mirroring real-world scenarios more closely. For instance, we first address network issues before delving into network partition problems.

2.3.2 SOP→SOP Code. Once a suitable SOP is obtained, due to the interdependence of steps within the SOP, it is generally necessary to execute the SOP step by step to achieve the desired outcome. However, in real-world scenarios, SOPs are typically concise texts, making it relatively difficult for engineers lacking domain knowledge to execute the entire SOP. Utilizing an agent based on LLM to execute the SOP is a more rational and efficient approach. However, directly instructing the agent to execute all steps of the SOP one by one often leads to errors. This is because LLM tends to focus more on proximal text, and the outcome of a particular step can significantly influence the selection of subsequent actions.

Therefore, we have designed *generate_sop_code* to convert the entire SOP into code for simultaneous execution. This approach offers three main advantages. Firstly, numerous works, including Chain-of-Code [11], have demonstrated that executing code in LLM environments is far more accurate than executing text [15], aligning well with the precise requirements of RCA. Secondly, in many scenarios, including RCA, there exist numerous atomic operations where we wish for several actions to be executed together or none at all, as executing a single action in isolation may not yield useful results. SOPs exemplify this situation, where executing only a portion may not yield the desired fault information. Converting SOPs to code effectively addresses this issue, as once the code is executed, it must run from start to finish. Lastly, SOP code represents a collection of multiple actions, enabling the execution of multiple actions with a single tool invocation, thereby significantly reducing LLM token and resource consumption.

2.3.3 SOP Code→Observation. After obtaining the SOP code, the flow invokes *run_sop* to execute the entire SOP code. However, the generation of code is not always accurate and may lead to various issues, such as syntax errors or incorrect variables within the code. In such instances, our flow expects to re-match suitable parameters and use *generate_sop_code* to generate new, correct code. Once the code is error-free, we can smoothly execute it to obtain the desired results.

2.3.4 SOP Code→Fault Type/Information. As mentioned earlier, the definition of SOP is hierarchical, and our RCA process follows a layered and progressive approach. Upon executing *run_sop* and obtaining a new observation, we seek guidance to determine the next steps in the localization process. The ideal approach is to identify potential fault types based on the observation. Relying solely on the domain knowledge of the LLM agent is evidently insufficient for accurate judgment in a specific domain, necessitating fine-tuning of the LLM model or the introduction of more domain-specific knowledge. Inspiration from various methods [3] suggests that most fault types have occurred historically. Therefore, we use *match_observation* to recall similar historical incidents based on observation. The ObAgent is then utilized to determine potential fault types or provide descriptions of faults for subsequent RCA processes.

2.4 Action Set

In section 1, we mentioned that in RCA, it is relatively challenging for the LLM agent to perform reasonable planning. This difficulty primarily arises from two reasons: the variability of observations and the existence of multiple possible actions for a given observation. Instantaneously identifying and executing the most reasonable action from numerous viable choices is an exceedingly challenging task for the LLM.

To address this challenge, we have devised a mechanism known as the action set. Specifically, drawing inspiration from the CoT [29], we first generate a series of reasonable actions comprising a set, with each action accompanied by a textual explanation of the rationale behind its selection. This set primarily consists of two components: actions generated by the ActionAgent and actions identified by the JudgeAgent. The ActionAgent incorporates flow

information and numerous examples in the prompt to enhance the rationality of the generated actions. However, this may still overlook reasonable flow actions. Therefore, we have established a rule based on the flow to ensure that the action set is comprehensive and logical. For instance, if the preceding action was *generate_sop*, the subsequent action of *generate_sop_code* is added to the set. Secondly, the JudgeAgent evaluates whether the root cause has been identified during the current RCA process. If the root cause is pinpointed, the action *Speak* is included in the action set.

Through action set, we have effectively mitigated the challenges posed by diverse observations and a plethora of feasible actions that could potentially hinder agent planning. Furthermore, the strategic design of the action set has enabled the LLM Agent to attain a nuanced equilibrium between stochasticity and determinism. Within RCA, excessive randomness may induce divergence in the localization process, impeding the formation of effective diagnostics. Conversely, an overly deterministic approach may incline the model towards scripted operations, limiting its capacity to handle unforeseeable and rapidly changing circumstances.

2.5 Multi-Agent System

We have designed a MAS consisting of a single main agent along with multiple auxiliary agents. The MainAgent serves as the principal entity with authority, while the other agents are responsible for providing suggestions to it. The MainAgent orchestrates the entire localization process. The ActionAgent provides a feasible set of actions for the MainAgent to choose from. The ObAgent offers potential anomaly types or information after the MainAgent completes *match_observation*. The JudgeAgent determines whether the root cause has been identified. However, even if the JudgeAgent believes the root cause has been found, the MainAgent may not necessarily use *Speak* to conclude the entire localization process. Taking additional steps and gathering more information may lead to a more accurate root cause determination. The CodeAgent plays a crucial role in the SOP flow, possessing information on all tools and generating appropriate code for subsequent use. Through the MAS, the burden on the MainAgent is significantly reduced. It only needs to consider the opinions of other agents and make relatively accurate judgments based on the entire localization process. Such division of labor also aligns more closely with real-world operational scenarios.

3 Evaluation

3.1 Experiment Setup

3.1.1 Dataset. We have deployed the widely used microservices system GoogleOnlineBoutique², an e-commerce system consisting of over 10 services, on the Kubernetes platform. Building upon this, we have implemented Prometheus, Elastic, DeepFlow [22], and Jaeger to collect metric, log, and trace data (Detailed in Appendix A). Anomalies are injected into microservices' pods using ChaosMesh³. There are a total of 9 types of anomalies injected, including CPU stress and memory stress (detailed in Table 2). Leveraging this setup, we have generated a dataset comprising 90 incidents. For further

²<https://github.com/GoogleCloudPlatform/microservices-demo>

³<https://github.com/chaos-mesh/chaos-mesh>

details on microservices architecture and multimodal data, kindly consult the resources available at <https://benchmark.aiops.cn/>.

Table 2: Fault Types

Type	Description
CPU Stress	Generate threads to occupy CPU resources.
Memory Stress	Generate threads to occupy memory.
Pod Failure	Make pods inaccessible for a period of time.
Network Delay	Cause network delay for a pod.
Network Loss	Cause packet loss in a pod's network.
Network Partition	Network disconnection, partition.
Network Duplicate	Cause network packet to be retransmitted.
Network Corrupt	Cause packets on network to be out of order.
Network Bandwidth	Limit the bandwidth between nodes.

3.1.2 Evaluation Metric And Baseline Methods. In the field of RCA, the specific location of the root cause is a critical focus for SREs. Additionally, categorizing the type of root cause is equally important, as SREs often specialize in different department like networking group or hardware group. Therefore, we have designed evaluation metrics focusing on both root cause location and fault type. Following the principle from mABC [36], we consider redundant causes to be less detrimental than missing causes. Hence, we utilize two metrics: Root Cause Location Accuracy (LA) and Root Cause Type Accuracy (TA).

$$LA = \frac{L_c - \sigma \times L_i}{L_t}, TA = \frac{T_c - \sigma * T_i}{T_t} \quad (1)$$

L_c and T_c represent all correctly identified root cause locations and types, while L_i and T_i denote the incorrectly identified locations and types. L_t and T_t represent total number of locations and types. σ serves as a hyperparameter with a default value of 0.1. To prevent an excessive number of root causes, we limit the maximum number of root causes to three in LLM-based methods. In addition, we employed the Average Path Length (APL) to evaluate the efficiency of the LLM Agents. APL is defined as $\frac{\sum_{k=1}^N L_k}{N}$, where L_k represents the diagnosis path length of the k-th sample, and N denotes the number of samples for which diagnosis was completed within the specified maximum path length.

Regarding baseline methods, we have chosen several open-source Kubernetes RCA tools, such as K8SGPT [9] and HolmesGPT [19]. Since the implementation of RCA agents is highly specific to the scenarios, they are not open-source and are challenging to migrate. Therefore, we have developed some general-purpose open-source frameworks, such as CoT [29], ReAct [32], and Reflexion [23], to serve as our baselines.

3.2 RQ1: Overall Performance

Based on Table 3, our Flow-of-Action surpasses the SOTA by 23% in the LA metric and 28% in the TA metric. Despite the support of LLMs, K8SGPT and HolmesGPT continue to exhibit poor performance. This can be attributed to the significant limitations in the information they access. For instance, K8SGPT primarily queries Kubernetes metadata for attribute information, which is often insufficient for RCA, as faults may not necessarily manifest in metadata.

CoT performs reasonably well in some common simple tasks due to the robust reasoning capabilities of LLMs. However, in RCA, where tasks are complex and diverse scenarios arise, even seasoned SREs struggle to promptly determine a series of pinpointing steps. Consequently, CoT fares poorly in the RCA domain. While ReAct integrates reasoning for each observation, the array of tools and diverse observations present challenges in rational orchestration. This is why we introduce the action set and SOP flow. Reflexion builds upon ReAct by introducing a path reflection mechanism. However, given that previous paths are predominantly incorrect, reflecting on a wealth of erroneous knowledge makes it arduous to arrive at accurate insights.

In terms of the APL metric, ReAct often erroneously identifies root causes due to a lack of proper judgment criteria, resulting in a relatively low APL. In contrast, Reflexion necessitates continuous path reflection, leading to numerous iterations and a higher APL. Flow-of-Action maintains an APL within an acceptable range, crucial for optimal performance in RCA tasks. In RCA tasks, the APL's magnitude is not fixed. Excessive values can escalate resource consumption and induce knowledge clutter, while inadequate values may lead to incomplete knowledge.

3.3 RQ2: Impact of Action Set Size

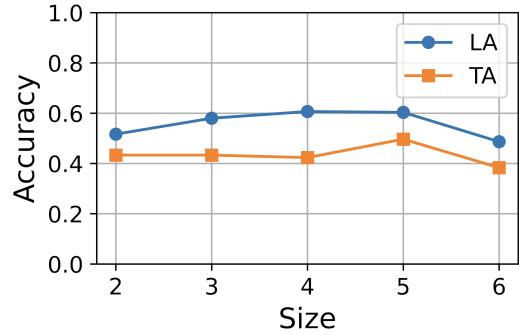


Figure 7: Accuracy of different action set sizes.

As shown in Figure 3, we have introduced the action set mechanism, where the size of the action set impacts the subsequent selection of actions. We conducted validation on a subset of the dataset and the results are shown in Figure 7. We observed that the LA and TA remain relatively stable with changes in the action set size. This stability is attributed to the fact that, despite variations in the action set size, relevant flow tools are encompassed within the action set due to the constraints of the rules in SOP flow. Furthermore, the entire RCA process typically follows the flow, thereby minimizing significant fluctuations in accuracy. However, as the size increases, accuracy initially rises and then declines. This phenomenon occurs because smaller action sets restrict randomness, rendering the model incapable of handling complex scenarios. Conversely, larger sizes introduce more randomness, leading to a loss of control by the model. Hence, we opt for a moderately sized default value of 5 as it strikes a balance between these extremes.

Table 3: Performance of different models. The best scores for each evaluation metric are bolded, and the second-best scores are underlined. Exclusive utilization of the APL metric is restricted to methodologies leveraging LLM agents. The fixed and specific accuracy of K8SGPT and HolmesGPT, i.e. 11.11, is due to their ability to handle only one type of fault.

Model	Base	LA	TA	Average	APL
K8SGPT	GPT-3.5-Turbo	11.11	11.11	11.11	-
HolmesGPT	GPT-3.5-Turbo	11.11	11.11	11.11	-
CoT	GPT-3.5-Turbo	20.89	15.56	18.26	-
CoT	GPT-4-Turbo	36.00	29.22	32.61	-
ReAct	GPT-3.5-Turbo	13.11	25.22	19.17	9.41
ReAct	GPT-4-Turbo	47.67	23.33	35.50	<u>10.76</u>
Reflexion	GPT-3.5-Turbo	21.56	22.22	21.89	22.38
Reflexion	GPT-4-Turbo	33.67	24.44	29.06	28.09
Flow-of-Action	GPT-3.5-Turbo	<u>54.22</u>	<u>53.89</u>	<u>54.06</u>	18.83
Flow-of-Action	GPT-4-Turbo	70.89	57.12	64.01	15.10

Table 4: Ablation study. The LLM backbone we use is GPT-3.5-Turbo.

Method	LA	TA	Average	APL
Flow-of-Action	54.22	53.89	54.06	18.83
w/o SOP Knowledge	8.56	22.11	15.39	20.00
w/o SOP Flow	15.11	39.89	27.50	19.78
w/o Action Set	44.67	40.00	42.34	11.48
w/o ActionAgent	32.78	34.56	33.67	18.42
w/o ObAgent	40.11	28.67	34.39	19.31
w/o JudgeAgent	36.11	33.89	35.00	20.00

3.4 RQ3: Ablation Study

We conducted a detailed ablation study by removing each module and each agent of Flow-of-Action, with the results summarized in Table 4. When the SOP was removed, lacking domain-specific guidance, the model relied solely on its own orchestration, essentially reverting to ReAct. The significantly low accuracy underscores the crucial role of SOP. It is worth mentioning that when SOP knowledge is removed, the SOP flow becomes ineffective as well, thus removing SOP knowledge is equivalent to removing both SOP knowledge and SOP flow.

Upon removing the prompts related to the SOP flow, we noticed a significant decrease in LA, while TA remained relatively effective. This is because SOP knowledge and relevant tools were still present and could provide type information through tools like *match_observation* or *match_sop*. However, the absence of the flow hindered the complete execution of the SOP, leading to the incapacity to discern location information.

The absence of the action set rendered the model unable to make correct judgments in complex and rare scenarios. However, in most cases, the model still performed adequately, resulting in a moderate decrease in effectiveness. Without the action set, the model tended to rely more on tools determined by the flow, reducing the likelihood of excessive tool invocations and thus significantly lowering APL.

At the multi-agent level, the removal of any single agent led to a certain degree of decrease in accuracy. This is attributed to the complexity of the RCA task, where having a single agent handle all processes may lead to oversight and hallucinations. In contrast,

a MAS with one main agent and multiple auxiliary agents effectively addresses this issue. The main agent can make decisions by considering the opinions of others, reducing the cognitive load and consequently achieving higher accuracy.

Regarding APL, apart from the significant impact of removing the action set, the effects of other ablations were relatively similar. This is due to the imposed limit of 20 steps to prevent unbounded loops that could render the RCA process unending.

4 Conclusion

The occurrence of frequent incidents necessitates RCA for swift issue resolution. Applying LLM agents in RCA presents numerous challenges. To address the challenges, we propose Flow-of-Action, a novel SOP-enhanced MAS. Flow-of-Action effectively leverages SOP by designing the SOP flow to alleviate hallucinations in the orchestration process. The action set mechanism efficiently tackles the challenge of selecting actions in the face of diverse observations. By employing a main agent supported by multiple auxiliary agents, Flow-of-Action further refines the delineation of responsibilities among agents, thereby enhancing the overall accuracy. Experimental results demonstrate the efficacy of Flow-of-Action.

5 Acknowledgment

This work was supported by the Chinese Academy of Sciences (241711KYSB20200023), the National Natural Science Foundation of China (62202445), and the National Natural Science Foundation of China-Research Grants Council (RGC) Joint Research Scheme (62321166652).

References

- [1] Sarthak Chakraborty, Shaddy Garg, Shubham Agarwal, Ayush Chauhan, and Shiv Kumar Saini. 2023. Causal: Causal graph for instance level microservice data. In *Proceedings of the ACM Web Conference 2023*. 2905–2915.
- [2] Hongyang Chen, Pengfei Chen, Guangba Yu, Xiaoyun Li, Zilong He, and Huxing Zhang. 2024. MicroFl: Non-Intrusive and Prioritized Request-Level Fault Injection for Microservice Applications. *IEEE Transactions on Dependable and Secure Computing* (2024).
- [3] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, et al. 2024. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 674–688.
- [4] Ruomeng Ding, Chaoyun Zhang, Lu Wang, Yong Xu, Minghua Ma, Xiaomin Wu, Meng Zhang, Qingjun Chen, Xin Gao, Xuedong Gao, et al. 2023. Trace-Diag: Adaptive, Interpretable, and Efficient Root Cause Analysis on Large-Scale Microservice Systems. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1762–1773.
- [5] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metapt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352* (2023).
- [6] Azam Ikram, Sarthak Chakraborty, Subrata Mitra, Shiv Saini, Saurabh Bagchi, and Murat Kocaoglu. 2022. Root cause analysis of failures in microservices through causal discovery. *Advances in Neural Information Processing Systems* 35 (2022), 31158–31170.
- [7] Soyeong Jeong, Jinheon Baek, Sukmin Cho, Sung Ju Hwang, and Jong C Park. 2024. Adaptive-rag: Learning to adapt retrieval-augmented large language models through question complexity. *arXiv preprint arXiv:2403.14403* (2024).
- [8] Xinrui Jiang, Yicheng Pan, Meng Ma, and Ping Wang. 2023. Look Deep into the Microservice System Anomaly through Very Sparse Logs. In *Proceedings of the ACM Web Conference 2023*. 2970–2978.
- [9] k8sgpt ai. 2023. k8sgpt. <https://github.com/k8sgpt-ai/k8sgpt>.
- [10] Labring. 2023. FastGPT. <https://github.com/labring/FastGPT>.
- [11] Chengshi Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. 2023. Chain of code: Reasoning with a language model-augmented code emulator. *arXiv preprint arXiv:2312.04474* (2023).
- [12] Zeyan Li, Nengwen Zhao, Mingjie Li, Xianglin Lu, Lixin Wang, Dongdong Chang, Xiaohui Nie, Li Cao, Wenchang Zhang, Kaixin Sui, et al. 2022. Actionable and interpretable fault localization for recurring failures in online service systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 996–1008.
- [13] Cheng-Ming Lin, Ching Chang, Wei-Yao Wang, Kuang-Da Wang, and Wen-Chih Peng. 2024. Root Cause Analysis in Microservice Using Neural Granger Causal Discovery. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 206–213.
- [14] Panagiotis Misiakos, Chris Wendler, and Markus Püschel. 2024. Learning DAGs from data with few root causes. *Advances in Neural Information Processing Systems* 36 (2024).
- [15] Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. 2023. LogicLM: Empowering large language models with symbolic solvers for faithful logical reasoning. *arXiv preprint arXiv:2305.12295* (2023).
- [16] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789* (2023).
- [17] Alec Radford. 2018. Improving language understanding by generative pre-training. (2018).
- [18] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
- [19] robusta dev. 2024. holmesgpt. <https://github.com/robusta-dev/holmesgpt>.
- [20] Carl Martin Rosenberg and Leon Moonen. 2020. Spectrum-based log diagnosis. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–12.
- [21] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2024. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems* 36 (2024).
- [22] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, et al. 2023. Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 420–437.
- [23] Noah Shim, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems* 36 (2024).
- [24] Gagan Somashekhar, Anurag Dutt, Mainak Adak, Tania Lorido Botran, and Anshul Gandhi. 2024. GAMMA: Graph Neural Network-Based Multi-Bottleneck Localization for Microservices Applications. In *Proceedings of the ACM on Web Conference 2024*. 2905–2909.
- [25] Shreshth Tuli, Giuliano Casale, and Nicholas R Jennings. 2022. Tranad: Deep transformer networks for anomaly detection in multivariate time series data. *arXiv preprint arXiv:2201.07284* (2022).
- [26] Lu Wang, Chaoyun Zhang, Ruomeng Ding, Yong Xu, Qihang Chen, Wentao Zou, Qingjun Chen, Meng Zhang, Xuedong Gao, Hao Fan, et al. 2023. Root cause analysis for microservice systems via hierarchical reinforcement learning from human feedback. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 5116–5125.
- [27] Zexin Wang, Jianhui Li, Minghua Ma, Ze Li, Yu Kang, Chaoyun Zhang, Chetan Bansal, Murali Chintalapati, Saravan Rajmohan, Qingwei Lin, et al. 2024. Large Language Models Can Provide Accurate and Interpretable Incident Triage. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 523–534.
- [28] Zexin Wang, Changhua Pei, Minghua Ma, Xin Wang, Zhihan Li, Dan Pei, Saravan Rajmohan, Dongmei Zhang, Qingwei Lin, Haiming Zhang, et al. 2024. Revisiting VAE for Unsupervised Time Series Anomaly Detection: A Frequency Perspective. In *Proceedings of the ACM on Web Conference 2024*. 3096–3105.
- [29] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
- [30] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. Microrca: Root cause localization of performance issues in microservices. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 1–9.
- [31] Haowen Xu, Wenxiao Chen, Nengwen Zhao, Zeyan Li, Jiahao Bu, Zhihan Li, Ying Liu, Youjian Zhao, Dan Pei, Yang Feng, et al. 2018. Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications. In *Proceedings of the 2018 world wide web conference*. 187–196.
- [32] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* (2022).
- [33] Zhenhe Yao, Changhua Pei, Wenxiao Chen, Hanzhang Wang, Liangfei Su, Huai Jiang, Zhe Xie, Xiaohui Nie, and Dan Pei. 2024. Chain-of-Event: Interpretable Root Cause Analysis for Microservices through Automatically Learning Weighted Event Causal Graph. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 50–61.
- [34] Zhenhe Yao, Haowei Ye, Changhua Pei, Guang Cheng, Guangpei Wang, Zhiwei Liu, Hongwei Chen, Hang Cui, Zeyan Li, Jianhui Li, et al. 2024. SparseRCA: Unsupervised Root Cause Analysis in Sparse Microservice Testing Traces. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*.
- [35] 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*. 553–565.
- [36] Wei Zhang, Hongcheng Guo, Jian Yang, Yi Zhang, Chaoran Yan, Zhoujin Tian, Hangyuan Ji, Zhoujun Li, Tongliang Li, Tieqiao Zheng, et al. 2024. mABC: multi-Agent Blockchain-Inspired Collaboration for root cause analysis in micro-services architecture. *arXiv preprint arXiv:2404.12135* (2024).
- [37] Lecheng Zheng, Zhengzhang Chen, Jingrui He, and Haifeng Chen. 2024. MULAN: Multi-modal Causal Structure Learning and Root Cause Analysis for Microservice Systems. In *Proceedings of the ACM on Web Conference 2024*. 4107–4116.

A Multimodal Data Collection

We first deploy various data collection systems (Figure 8, Figure 9, Figure 10, Figure 11). For metrics, we start by deploying Prometheus, which collects architecture-level metrics, such as pod-level and node-level indicators that are generally standardized and unrelated to business logic. Additionally, we deploy DeepFlow to gather business-level metrics, such as business traffic data. For anomaly detection, we use traditional rule-based methods because they are fast and convenient.

For trace data, we deploy Jaeger to collect all trace data, where each trace represents a call chain containing multiple spans, with each span corresponding to a single call. Anomalies can occur within any span. In the current environment, detecting trace anomalies is relatively straightforward, as a span failure typically includes

an associated error message. Therefore, we directly extract error messages to generate alert reports. For log data, we use Elastic for collection. Since abnormal logs usually contain specific keywords, extracting anomalies based on keywords has become widely accepted. We also adopt this keyword-based approach for log anomaly detection.



Figure 8: Prometheus Dashboard.



Figure 9: Deepflow Dashboard.

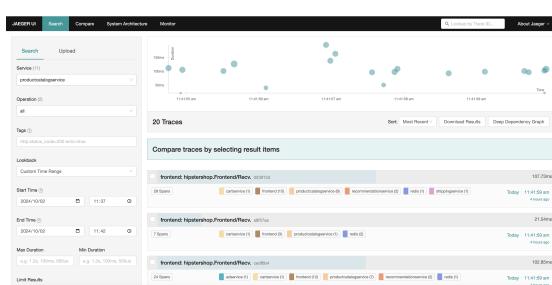


Figure 10: Jaeger Dashboard.

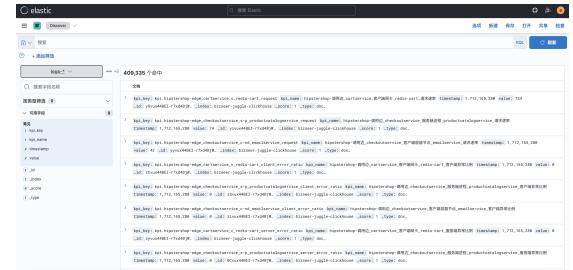


Figure 11: Elastic Dashboard.

B Tools

Table 5: Description of Tools

Tool	Description
pod_analyze	Analyzing all pods' status.
node_analyze	Analyzing all nodes' status.
service_analyze	Analyzing all services' status.
deployment_analyze	Analyzing all deployments' status.
statefulset_analyze	Analyzing all statefulsets' status.
run_kubectl_command	Executing kubectl commands generated by LLMs.
get_all_namespace	Obtaining a list of all namespaces.
get_relevant_metric	Obtaining relevant metric names according to query.