LogSage: An LLM-Based Framework for CI/CD Failure Detection and Remediation with Industrial Validation

Weiyuan Xu[1,2,†], Juntao Luo[2,†,‡], Tao Huang[2], Kaixin Sui[2], Jie Geng[2],
Qijun Ma[2], Isami Akasaka[2], Xiaoxue Shi[2], Jing Tang[2], Peng Cai[1,*]
[1]East China Normal University, Shanghai, China
[2]ByteDance, Shanghai, China
72275900021@stu.ecnu.edu.cn, pcai@dase.ecnu.edu.cn
{luojuntao,huangtao.806955,sunyikai.s,gengjie.02,maqijun,
isami.akasaka,shixiaoxue.111,tangjing.fisher}@bytedance.com

***Abstract*—Continuous Integration and Deployment (CI/CD) pipelines are critical to modern software engineering, yet diagnosing and resolving their failures remains complex and labor-intensive. We present LogSage, the first end-to-end LLM-powered framework for root cause analysis (RCA) and automated remediation of CI/CD failures. LogSage employs a token-efficient log preprocessing pipeline to filter noise and extract critical errors, then performs structured diagnostic prompting for accurate RCA. For solution generation, it leverages retrieval-augmented generation (RAG) to reuse historical fixes and invokes automation fixes via LLM tool-calling.**

On a newly curated benchmark of 367 GitHub CI/CD failures, LogSage achieves over 98% precision, near-perfect recall, and an F1 improvement of more than 38% points in the RCA stage, compared with recent LLM-based baselines. In a year-long industrial deployment at ByteDance, it processed over 1.07M executions, with end-to-end precision exceeding 80%. These results demonstrate that LogSage provides a scalable and practical solution for automating CI/CD failure management in real-world DevOps workflows.

***Index Terms*—Continuous Integration, Continuous Deployment, Large Language Models, Log Analysis, Root Cause Diagnosis, Failure Remediation**

I. Introduction

As the backbone of modern software engineering, Continuous Integration and Continuous Deployment (CI/CD) has become critical infrastructure for reliable, rapid software delivery [1]. It enables higher release frequency, faster iteration, and reduced operational risk, and is widely adopted across leading technology companies [2]–[4]. Large-scale empirical studies also confirm its growing adoption in open-source communities and its pivotal role in modern development practices [5].

In parallel, large language models (LLMs) have achieved impressive results across a range of software engineering tasks [6], including requirements engineering [7], code retrieval [8], automated code review [9], and unit test enhancement [10]. Encouraged by these advances, recent work has begun exploring LLM-based approaches for failure detection and remediation—opening up new possibilities for intelligent automation in software delivery.

Yet despite progress in automated software engineering, real-world CI/CD pipelines still suffer frequent failures. Diagnosing and fixing them typically requires on-call engineers to inspect lengthy, semi-structured logs filled with irrelevant or misleading information, a process that is time-consuming and detrimental to both development velocity and product stability. While many failures follow recurring patterns and organizations accumulate substantial resolution knowledge, this knowledge is often stored in unstructured formats that are difficult to retrieve or reuse with traditional non-LLM methods, leading to redundant work and wasted resources.

A major barrier is the limited academic focus on CI/CD log analysis. Although log-based anomaly detection is well studied, most work targets streaming system logs [11] and assumes structured formats or consistent templates [12]–[16], which generalize poorly to noisy, context-rich, file-level CI/CD logs. Even recent CI/CD log analysis efforts [17] rely on template-based deep learning techniques with poor generalizability. These approaches often require large labeled datasets for training, lack interpretability, and cannot produce actionable remediation strategies. While LLMs offer promising reasoning capacity, their application to CI/CD root cause analysis remains underexplored [18], and existing baselines leave considerable room for improvement in diagnostic accuracy, repair guidance, and end-to-end integration.

To address these gaps, we present **LogSage**, the first end-to-end LLM-powered framework for CI/CD failure detection and remediation. LogSage conducts fine-grained root cause analysis (RCA) and automated resolution directly from raw logs, producing human-readable diagnostic reports and applying executable fixes through tool-calling. Its two-stage pipeline combines token-efficient log preprocessing for RCA with multi-route retrieval-augmented generation (RAG) for solution generation, retrieving domain knowledge from diverse internal

* Corresponding author.† Equal contribution.‡ Work conducted during the internship at ByteDance.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00310

sources and integrating it with RCA report to synthesize executable remediation strategies. This enables accurate RCA, interpretable reasoning, and automated recovery with minimal developer effort. Our key contributions are as follows:

- **First end-to-end LLM framework:** LogSage is the first LLM-based framework for fully automated CI/CD failure detection and remediation. It integrates a token-efficient log preprocessing strategy that filters noise, expands context, and adheres to token constraints without relying on templates, together with a multi-route RAG-based solution generator that reuses historical fixes and leverages LLM tool-calling for executable remediation.
- **Curated dataset:** We release a dataset of **367** real-world CI/CD failures from GitHub repositories, each annotated with key log evidence, enabling reproducible RCA evaluation and fostering future research (see Appendix A).
- **Extensive validation:** On the curated dataset, LogSage improves RCA F1 score by over **38%** compared with prompt-based LLM baselines, achieving **98% precision** and near-perfect recall. In large-scale industrial deployment at ByteDance, it processed over **1.07M executions** with sustained adoption and **80%+ end-to-end precision**, demonstrating practical usability in production.

The remainder of this paper is structured as follows: Section II surveys related work. Section III introduces the LogSage framework, including both its high-level architecture and implementation details. Section IV presents experiments and evaluates the RCA stage. Section V reports on the real-world deployment of LogSage at ByteDance. Finally, Section VI concludes the paper and outlines directions for future research.

II. Related Work

With the rise of AI techniques in software engineering, CI/CD pipelines have increasingly adopted machine learning (ML), deep learning (DL), and LLMs to improve efficiency, reliability, and fault tolerance [19]. In this section, we review prior work from three perspectives: traditional AI methods in CI/CD pipelines, log anomaly detection, and the emerging application of LLMs for failure detection and remediation.

A. AI in CI/CD Pipelines

Traditional ML approaches in CI/CD focus primarily on predicting test outcomes, build success, or optimizing test execution to reduce cost. These methods typically rely on structured features such as code metrics and commit history, with limited support for runtime failure analysis or recovery. For instance, CNNs have been used to identify false positives in static code analysis [20], while other studies aim to skip unnecessary tests or prioritize test selection to save time and cost [21]–[25]. Additionally, defect prediction has been explored as an indirect way to reduce failures, though these methods do not directly address fault localization or remediation [26], [27].

DL methods enhance predictive accuracy and generalization in failure prediction tasks [28], [29]. However, most models lack semantic understanding of failure causes. Mahindru et al. [30] proposed the LA2R system, which combines log anomaly detection with metadata prediction to retrieve relevant resolutions. While effective, this approach depends heavily on templates and rule-based clustering, limiting its adaptability to evolving log formats and unseen failures.

B. Log Anomaly Detection

Among CI/CD tasks, log anomaly detection is particularly critical and challenging due to the unstructured and context-dependent nature of log data. Early deep learning work by Du et al. [13] used LSTMs to learn normal log sequences and detect anomalies. Meng et al. [14] extended this idea with Template2Vec to capture sequential and quantitative anomalies. Yang et al. [15] employed a GRU-based attention model and semi-supervised learning to reduce labeling cost, while Zhang et al. [16] addressed unstable logs using attention-based Bi-LSTMs. Recent work by Le et al. [11], [12] introduced parser-independent Transformer-based methods that improve generalization by avoiding reliance on log parsing.

Given their strong semantic understanding and reasoning abilities, LLMs are well suited for log anomaly detection. Studies have shown that even prompt-based LLMs can detect anomalies in streaming logs [31]–[34]. Qi et al. [35] proposed a GPT-4-based detection framework, while Shan et al. [36] leveraged configuration knowledge for anomaly detection. Almodovar et al. [37] fine-tuned LLMs for better performance, and Ju et al. [38] adopted RAG to enhance log interpretation. Hybrid methods have also been proposed: Guan et al. [39] combined BERT and LLaMA for semantic vector extraction, and Hadadi et al. [40] used LLMs to compensate for missing context in unstable environments. In-context learning techniques were further used to refine log representation and guide fine-tuning [41], [42].

Despite these advances, most studies focus on streaming logs, with limited attention to file-based CI/CD logs. Moreover, few works provide large-scale industrial validation or full pipeline integration, leaving a gap for further exploration.

C. LLMs for Failure Detection and Remediation

Beyond anomaly detection, LLMs offer unique potential in end-to-end failure diagnosis and automated remediation. A recent survey identified root cause analysis and remediation as key use cases for LLMs in AIOps [43]. For root cause analysis, Roy et al. [44] and Wang et al. [45] proposed tool-augmented LLM agents, while Zhang et al. [46] and Goel et al. [47] used in-context learning and RAG for incident diagnosis with real-world data. On the remediation side, Sarda et al. [48], [49] and Ahmed et al. [50] automated fault recovery in cloud environments. Wang et al. [51] incorporated Stack Overflow into LLM-based solution generation, and Khlaisamniang et al. [52] applied generative AI for self-healing systems.

In the CI/CD context, however, LLM-based solutions remain underexplored. Chaudhary et al. [53] proposed an LLM-based assistant for CI/CD operations, and Sharma et al. [54] outlined key challenges in adopting LLMs for fault handling in build pipelines. While promising, these efforts have yet to

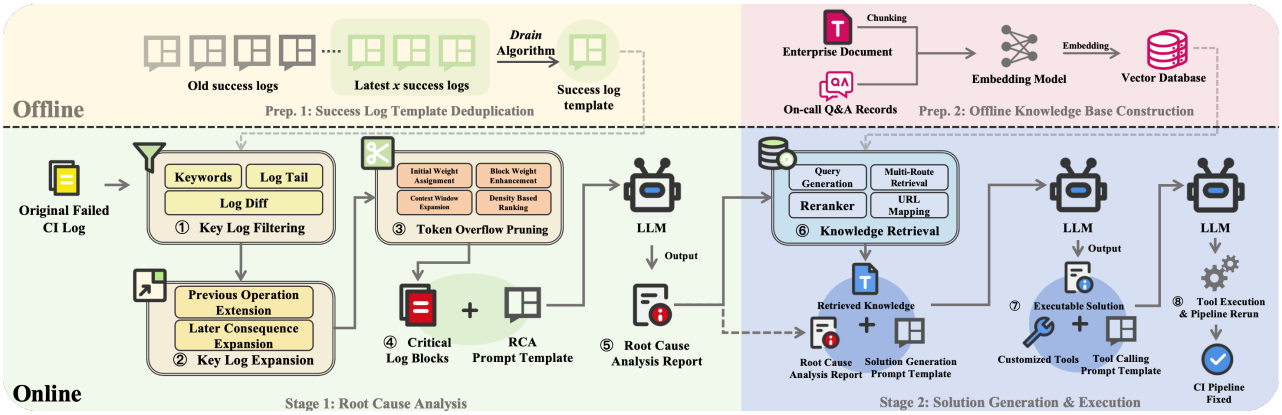

Fig. 1: Overview of the **LogSage framework**, consisting of an offline preparation phase for log template deduplication and knowledge base construction, and an online operational phase for RCA and solution generation with execution.

address the complexity of real-world CI/CD logs or provide comprehensive remediation workflows.

Overall, while ML and DL methods have laid the foundation for predictive CI/CD analytics, LLMs bring new opportunities for building robust, context-aware, and automated failure detection and remediation systems. However, systematic integration of LLMs into CI/CD pipelines, especially for file-based logs and knowledge-driven automated remediation, remains an open research challenge.

III. Methodology

In this section, we first provide an overview of the LogSage framework architecture in Section III-A, then we elaborate on the detailed workflows and key components of the two stages of LogSage in Section III-B and Section III-C, respectively.

A. Overview of LogSage

The two-stage architecture of LogSage is illustrated in Figure 1. The system consists of both offline and online phases. In the offline phase, success log templates are deduplicated via the Drain algorithm (Prep. 1). Enterprise documents and on-call Q&A records are embedded into vector databases to construct a knowledge base (Prep. 2). In the online phase, Stage 1 processes failed logs through (1) key log filtering, (2) expansion, and (3) token pruning, yielding (4) critical log blocks that form RCA prompt to generate (5) RCA report. Stage 2 leverages (6) knowledge retrieval and LLM-based tool selection to generate (7) executable solutions, followed by (8) automated tool execution and pipeline rerun, ultimately restoring the CI/CD pipeline.

B. Root Cause Analysis Stage

In this section, we present the design and implementation of the RCA stage. The goal of this stage is to extract the most relevant portions of raw CI/CD failure logs and enable accurate reasoning by the LLM, which then returns a structured diagnostic report containing key error lines and an interpretable root cause summary. This stage must address several practical challenges inherent in real-world CI/CD log analysis:

First, **log heterogeneity**: CI/CD pipelines differ widely in their execution steps, commands, output formats, and logging styles, making it infeasible to rely on a unified, structured template to accommodate all scenarios.

Second, **informational noise and misleading lines**: Raw logs often contain numerous irrelevant `WARNING` or `ERROR` messages that do not reflect the actual cause of failure. Naively feeding the full log into the LLM can lead to degraded reasoning quality and even hallucinations. Conversely, filtering logs too aggressively (e.g., by extracting only lines containing keywords such as `ERROR` or `FAILED`) may omit critical contextual information, causing the model to speculate and increasing the risk of misdiagnosis.

Third, **input length constraints**: LLMs cannot accept arbitrarily long inputs. Excessively long log sequences not only increase inference cost and latency but also reduce reasoning accuracy due to context dilution. In industrial CI/CD environments, these limitations become particularly critical, as practical deployments require LLM applications to maintain a stable context length and predictable response latency in order to ensure system robustness and operational usability.

To address these challenges, we design a dedicated log preprocessing pipeline prior to model invocation. This pipeline consists of the following modules:

- **Key Log Filtering**: Extracts candidate log lines from the *failed log* based on keyword matching, log tail prioritization and log diff against recent *success log template*.
- **Key Log Expansion**: Adds contextual lines surrounding the extracted errors to preserve semantic coherence and prevent information loss.
- **Token Overflow Pruning**: Ranks expanded log blocks by weight calculation and prunes low-priority blocks to ensure input remains within a predefined token limit.
- **Dynamic Prompt Assembly**: Constructs RCA prompt by integrating role-playing, chain-of-thought reasoning, few-shot learning, and output-format constraints. Then

dynamically combines them with the processed critical log blocks to guide the LLM toward accurate and reproducible RCA reasoning.

The processed critical log blocks are delivered to the LLM through a structured prompt, allowing the model to perform accurate reasoning and generate a root cause analysis report.

*1) **Key Log Filtering Module**:* This module is designed for accurately extracting relevant error log lines from the *failed log* by combining *log diff, keyword matching* and *log tail prioritization*. These strategies collectively ensure broad and precise coverage of failure-relevant log lines. The filtering process is formalized in Algorithm 1.

Algorithm 1 Key Log Filtering Process

Input: $failed\_log$, $success\_log\_templates$ {from offline preparation}
Output: $filtered\_log$
Initialize $candidate\_pool \leftarrow \{\}$
for each line l in $failed\_log$ **do**
  $template \leftarrow \text{extract\_template}(l)$
  $position \leftarrow \text{get\_position}(l, failed\_log)$
  if $template \notin success\_log\_templates$ **then**
    Add l to $candidate\_pool$ {log diff}
  end if
  if $\text{contains\_keyword}(l)$ **then**
    Add l to $candidate\_pool$ {keyword matching}
  end if
  if $\text{is\_in\_log\_tail}(position)$ **then**
    Add l to $candidate\_pool$ {log tail prioritization}
  end if
end for
$filtered\_log \leftarrow \text{deduplicate}(candidate\_pool)$
return $filtered\_log$

Log diff is the most critical strategy in the key log filtering module. It leverages the repetitive nature of CI/CD pipelines: executions typically share almost identical configurations across consecutive runs, yielding highly consistent log outputs during successful executions, and even when changes occur, they are usually incremental and persist across multiple subsequent runs. Exploiting this stability, the system performs an offline process that applies the *Drain* algorithm [55] to recent *success logs* of the same pipeline, extracting structural templates that characterize stable and recurring log lines. These templates are then stored and used online to filter the *failed log*: lines matching the success-run templates are treated as background outputs that are highly unlikely to contain failure-related information, and are therefore excluded from the error candidate set. This log-diff approach effectively eliminates misleading `WARNING` or `ERROR` lines while avoiding rigid handcrafted rules, thereby significantly reducing noise in downstream analysis.

To maintain the freshness and relevance of filtering templates derived from *success logs*, LogSage adopts an offline ***log template deduplication*** strategy: for each pipeline task, only the most recent x successful logs are retained for template extraction, where x is configurable. Empirical analysis across diverse CI/CD projects in ByteDance shows that setting $x = 3$ achieves the best trade-off between template diversity and noise reduction, and is used as the default in our system. The value of x can be tuned based on pipeline stability and log variance, thereby improving the generalizability of LogSage.

Keyword matching identifies log lines containing high-risk terms based on a curated set of failure-related keywords mined from historical CI/CD failure cases. Any log line matching one or more of these keywords is added to a candidate pool for further downstream processing. The keyword set includes:

```
fatal, fail, panic, error,
exit, kill, no such file, err:,
err!, failures:, err , missing,
exception, cannot
```

The ***log tail prioritization*** strategy is motivated by the empirical observation: most critical error logs tend to appear near the end of the file, as CI/CD pipeline failures often lead to abrupt termination. Accordingly, log lines appearing at the end of the *failed log* are prioritized during candidate selection.

Through these strategies, the *Key Log Filtering* module is able to deconstruct the *failed log* with high precision and identify all potentially problematic log lines. After removing redundant lines, the filtered logs are structured as pairs of line numbers and corresponding log lines, serving as input for subsequent processing.

*2) **Key Log Expansion Module**:* Based on manual experience in analyzing CI/CD failure logs, it is often insufficient to rely solely on the `ERROR` log lines that directly report failures. The true root cause of a pipeline error is typically identified by examining several lines before and after the failure line. This observation motivated the design of the *Key Log Expansion* module, which provides additional contextual information to support LLM-based root cause analysis, helping to mitigate hallucinations and logical errors caused by missing context.

Starting from the key error lines identified by the previous module, the expansion module includes m lines before and n lines after each key line to form a log block. We adopt an asymmetric expansion strategy where $n > m$: the preceding m lines ensure that the operations leading to the error are captured, while the succeeding n lines provide richer post-error information such as stack traces to cover information that is often more critical than the pre-error context. Overlapping blocks resulting from multiple key lines are merged into a single cohesive block to maintain contextual continuity. This expansion ensures that the LLM can access sufficient surrounding information to interpret the key log lines accurately.

In practice, we set $m = 4$ and $n = 6$ for both online deployments and offline experiments. Empirical results show that this configuration is sufficient to retain most of the contextual information necessary for accurate root cause analysis.

*3) **Token Overflow Pruning Module**:* In practical production environments, the token length limitation of LLMs

imposes a critical constraint, as overly long input tokens can reduce compliance with instructions, introduce hallucinations, and increase computational cost and latency. To address this, LogSage incorporates a *Token Overflow Pruning* module that enforces a predefined token limit during RCA. This module leverages a structured log weighting and enhancement algorithm to assess the relative importance of expanded log blocks produced in the previous module. This module includes four components: *initial weight assignment* to identify candidate lines, *pattern-based weight enhancement* to emphasize critical error lines, *contextual window expansion* to preserve semantic continuity, and *density-based block ranking* to prioritize the most informative blocks while satisfying the token constraint.

Initial Weight Assignment: Given the *failed log* $L = \{l_1, l_2, ..., l_n\}$ and log lines from the candidate pool $I = \{i_1, i_2, ..., i_m\}$, we define a weight list $W = \{w_1, w_2, ..., w_n\}$, where each w_j represents the weight assigned to line l_j:

$$w_j = \begin{cases} 3 & \text{if } \frac{|I|}{|L|} \leq \alpha \text{ and } |I| \leq \beta \text{ and } j \in \text{candidate pool} \\ 1 & \text{if } j \in \text{candidate pool} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

This adaptive rule (Equation 1) adds an initial weight to each log in the candidate pool and ensures that higher weights are assigned to sparse yet informative candidate log lines. The parameters α and β are tunable thresholds that control the criteria for assigning high weights. Based on empirical observations, we set $\alpha = 0.7$ and $\beta = 500$ in practice.

Pattern-Based Weight Enhancement. To emphasize critical log lines, we apply a rule-based scheme: (i) lines containing typical failure markers (e.g., `--- FAIL:`, `Failures:`) are assigned the maximum weight $w_i = 10$; (ii) lines with curated keywords or section headers (e.g., starting with #) receive a moderate boost ($w_i = w_i + 2$); and (iii) remaining candidates in the recall pool are lightly reinforced with $w_i = w_i + 1$. This design ensures both explicit and implicit failure signals are prioritized in subsequent pruning and diagnosis.

Contextual Window Expansion: To ensure the contextual integrity of high-weight critical log lines, log lines with weights above the threshold θ are expanded into log blocks. For any $w_i \geq \theta$, the log blocks in its neighborhood $[i-m, i+n]$ are added to the candidate pool, where m and n represent the number of previous and next lines as defined previously.

The threshold θ is adaptively defined by Equation 2 according to two different situations:

$$\theta = \begin{cases} 1 & \text{if } \max(W) = 1 \text{ or } |\{w_i \geq 1\}| \leq \gamma \\ 3 & \text{otherwise} \end{cases} \tag{2}$$

In the first scenario, all key log lines receive a uniform weight of 1, or the number of high-weight lines falls below a threshold γ, suggesting that the preceding filtering step has limited effectiveness in isolating truly critical lines. In this case, broader contextual expansion is required to ensure the LLM has sufficient information. In contrast, when high-weight lines are adequately identified, context expansion is applied selectively around those lines to maintain focus and efficiency. In practice, we empirically set $\gamma = 500$.

Density-Based Block Ranking: In order to compare the weights between different log blocks, we define the log block weight density. For contiguous candidate log lines that are grouped into blocks $B_j = [s_j, e_j]$, their weight density is computed as Equation 3:

$$\text{density}(B_j) = \frac{\sum_{i=s_j}^{e_j} w_i}{e_j - s_j + 1} \tag{3}$$

After computing the weight density for all candidate log blocks, they are ranked in descending order of density and indexed as $B_1, B_2, \ldots, B_K$, where k denotes the rank of the k-th block in this sorted list. A greedy selection algorithm is then applied to include as many blocks as possible while ensuring that the total token count does not exceed the predefined limit:

$$\text{Token Limit} \geq \sum_{i=1}^{k-1} \text{Token}(B_i) + \text{Token}(B_k)$$

Based on empirical observations, we set the token limit at 22,000, which provides sufficient contextual coverage while maintaining a reasonable computational cost. Under this constraint, high-density log blocks are retained with priority, while low-density blocks exceeding the token limit are discarded to achieve effective pruning.

Overall, this weighted pruning strategy strikes a practical balance between capturing critical error information and controlling input length. It is well-defined and tunable, allowing parameter adjustments across diverse real-world CI/CD systems, and thereby demonstrating strong generalizability.

*4) **Root Cause Analysis Prompt Template Design**:* To facilitate RCA, LogSage employs a task-specific prompt template (Fig. 2) that incorporates prompt engineering techniques such as role-playing, few-shot learning, and output format constraints. This template provides the LLM with explicit task instructions and takes filtered critical error log blocks as input, guiding the model to focus on diagnostic reasoning.

C. Solution Generation and Execution Stage

In this section, we present the design and implementation of the solution generation and execution stage. LogSage combines critical log blocks and RCA report from the previous stage with domain-specific knowledge retrieved via a multi-route RAG mechanism. The system constructs a prompt enriched with RCA report, domain knowledge and well-designed tools, enabling the LLM to produce executable remediation suggestions and automatically select and invoke appropriate repair tools to resolve CI/CD pipeline failures.

*1) **Offline Knowledge Base Construction**:* At ByteDance, a large volume of domain knowledge has been accumulated through CI/CD platform operations. This includes:

- 1,206 Feishu documents detailing production CI/CD issues and resolutions, contributed by various teams.
- 23,344 historical on-call Q&A pairs recorded by rotating engineering teams responsible for incident response.

Feishu documents are segmented using a chunking strategy with a 3,000 max-token cutoff and embedded via LLM-based encoders. Q&A pairs, due to their brevity, are stored directly as `<question, answer>` entries. To ensure high availability, the entire knowledge base is replicated across both *VikingDB* and *Elasticsearch*, enabling failover retrieval in production environments.

*2) **Online Multi-Route Retrieval Mechanism**:* Online process consists of four steps: *query construction*, *coarse retrieval*, *reranking*, and URL mapping.

RAG Query Construction: To mitigate the inherent variability in natural-language root cause descriptions, we reformulate the RAG query by combining the LLM-generated root cause with the corresponding critical log snippet. This hybrid query construction enhances retrieval efficiency and accuracy by better aligning contextual semantics. For example:

LLM Output: `Unit test TestFilterPushCdnOnCreate failed.`
CI Error Message:

```
Warn 2024-04-26 18:08:39,457 v1(7)
stream_create.go:1524 ... unmarshal err ...
ReadMapCB: expect { or n, but found \x00 ...
```

Multi-Route Coarse Retrieval: LogSage's coarse-grained retrieval combines three orthogonal dimensions: (1) *database infrastructure*, including VikingDB and Elasticsearch; (2) *matching granularity*, using both `query2doc` (query-to-content) and `query2query` (query-to-title) strategies; and (3) *similarity metric*, supporting `BM25` (sparse lexical) and `KNN` (dense vector) retrievals. These combinations yield six base retrieval paths. To further enhance recall coverage, we integrate two auxiliary routes: `lark_wiki` (for enterprise documents via Lark search API) and `rds_match` (for internal historical query log mining). Thus resulting in 8 coarse retrieval routes:

```
query2doc_viking_knn     query2query_viking_knn
query2doc_es_knn         query2doc_es_keyword
query2query_es_keyword   query2query_es_knn
lark_wiki                rds_match
```

Reranking: After coarse retrieval, reranking is performed using BGE (BAAI General Embedding) model and cosine similarity:

- For Feishu documents, the entire chunk is used to compute similarity, as titles are often unaligned with specific CI/CD issues. The top 10 results are retained.
- For Q&A pairs, since the query already aligns with the "question" field, only the top 100 coarse candidates are reranked, and the top 10 are selected.

URL Mapping: To prevent hallucinations caused by long URLs during generation, all retrieved links are replaced with numbered placeholders (e.g., `[CI Guide](files_0)`) before being passed to the LLM. Once a solution is generated, the placeholders are mapped back to the original URLs, allowing users to access the full content for further reference.

Root Cause Analysis Prompt Template

 # Role: You are a CI/CD failure diagnosis assistant. Your task is to identify the root cause of pipeline failures based on execution logs and configuration info.

 # Skills:

- **Task Type Identification:** Read config files to determine the task type (e.g., unit test, code scan). Output under `Diagnosis Process → Task Type`.
- **Error Log Analysis:** Read logs to identify up to 10 key error lines. Focus on terminal and causal errors. Output as `line range + conclusion`. Do NOT analyze normal/warning logs.
- **Root Cause Inference:** Use log and config analysis (don't mix unrelated errors). List up to 3 likely causes with concrete names and detailed, objective explanation. No fix suggestions.

 # Output Format:

- Two parts: `Diagnosis Process` and `Root Cause`.
- For `Diagnosis Process`, include:
  - Task type: e.g., `Run npm dependency installation`
  - Error analysis: e.g., `Lines 6{12: Unit test 'abc' failed due to result mismatch`
  - Summarize causally related/similar errors in one line
  - When referencing too many lines, use only first 5 + `etc.`
- For `Root Cause`, format each cause as:
  `[High Likelihood] Unit test `abc` failed due to ...`
- Prefer one cause, max three. Use concrete info (test name, file, dep).

 # Notes:

- Be concise and factual. Use "lines a, b, c–d" format when needed.
- Use `inline code` for log lines, `code blocks` for log content.
- No fix suggestions allowed.
- All results will be used for solution generation. Follow rules strictly.

 # Constraints:

- DO NOT include normal/process/non-critical logs.
- DO NOT analyze similar/adjacent logs separately.
- DO NOT output more than 5 log line references without using `etc.`

Fig. 2: Prompt template for LogSage's root cause analysis stage.

*3) **Solution Generation and Automated Execution**:* Upon completing knowledge retrieval, LogSage proceeds to the remediation stage, where it synthesizes solutions based on RCA report and retrieved domain knowledge. This stage explicitly decouples *solution generation* from *tool execution*, improving output focus by separating reasoning-oriented and action-oriented prompts.

LogSage prioritizes generating high-quality, actionable suggestions grounded in enterprise-specific knowledge. While internal tools can automate a subset of typical failures, many CI/CD issues remain beyond full automation. For such cases, LogSage still provides structured and executable guidance for developers, balancing intelligent assistance with interpretability rather than pursuing full end-to-end autonomy.

Prompt Construction for Solution Generation: To generate high-quality remediation suggestions, the system populates a predefined prompt template (Fig. 3) with the RCA report, critical log blocks, and retrieved domain knowledge. This prompt is then passed to the LLM to generate an executable solution tailored to the specific CI/CD failure context.

Automated Tool Selection and Execution: Once a solution is generated, LogSage leverages LLM tool-calling guided by the generated solution, the available tool set, and a specially designed prompt template. The LLM then selects and invokes the most appropriate tools from the internal automation toolkit. Currently, these tools include:

- `lint_fix`: automatically resolves lint-related issues without human intervention by leveraging the LLM's contextual understanding and code generation capabilities;
- `image_fix`: addresses container image version mismatches and automatically retrieves the correct images;
- `clean_cache`: clears build or dependency caches;
- `apply_resource`: requests access to internal resources.

After selecting the appropriate tool, the model fills in the required parameters and returns an interactive tool card with an execution button to the user interface. Upon user confirmation, the system invokes the selected tool and automatically re-triggers the CI/CD pipeline. If the remediation succeeds, the pipeline proceeds normally.

Solution Generation Prompt Template

 # Role: You are a CI/CD pipeline failures fix assistant. Your task is to generate repair suggestions based on error logs, RCA report, and retrieved troubleshooting documents.

 # Skills:

- **Targeted Suggestions:** Use RCA report and relevant documents to propose solutions. Each solution must align with the diagnosed error.
- **Multiple Options:** For each issue, list all applicable solutions. Choose document content most relevant to the CI/CD context.
- **Cite Source:** Every solution must mention the source document and include its official link (if provided). Do not fabricate URLs.
- **Command-level Guidance:** Provide actionable suggestions (e.g., install commands or CI config changes), not vague descriptions.

 # Output Format:

- Use section: `### Solutions`.
- For a single root cause:

```
### Solutions
#### Problem: <summary>
##### 1. <Suggestion>
... Refer to [Doc Name](file_0)
```

- For multiple root causes:

```
### Solutions
#### Problem 1: ...
##### 1. ...
##### 2. ...
#### Problem 2: ...
```

- Always include test name, file path, or dependency name when available.

 # Notes:

- Be concise and avoid redundant descriptions.
- Do not suggest solutions not backed by retrieved documents.
- Do not add file links; only use document links if provided.

 # Constraints:

- DO NOT generate new URLs — only use those in the documents.
- DO NOT omit document attribution in solutions.
- DO NOT provide vague advice like "check the config".

Fig. 3: Prompt template for LogSage's solution generation stage.

IV. Experimental Evaluation for RCA Stage

In our survey of related work, we found that existing research provides suitable baselines for comparison with LogSage's first stage (LLM-based RCA), but no appropriate counterparts exist for the second stage. Due to this limitation, we designed our experiments in two parts: for the first stage, we collected CI/CD task data from public GitHub repositories to compare LogSage with existing LLM-based RCA approaches, highlighting its theoretical performance advantages. The evaluation of the second stage, as well as the end-to-end effectiveness of LogSage, is presented in the following section V using data from ByteDance's real-world industrial deployments.

The research objectives for RCA in this section are:

- **RQ1**: How does LogSage perform in root cause analysis precision compared to existing LLM-based baselines?
- **RQ2**: How does LogSage's cost efficiency in the root cause analysis stage compare to LLM-based baselines?

We conducted the following experiments around the above questions.

A. Experimental Setup

To comprehensively validate the performance of LogSage across various CI/CD scenarios, we built a public dataset for comparative evaluation against other LLM-based root cause analysis baselines. The models selected for this experiment are GPT-4o, Claude-3.7-Sonnet and Deepseek V3, with hyperparameters set to temperature = 0.1, and all other settings using the default values of the respective models.

*1) **Dataset Description**:* To obtain representative and diverse CI/CD failure log cases, referring to [56], we crawled the top 1,000 GitHub repositories by star count and filtered non-engineering-related repositories to ensure the cases have practical software engineering relevance. We then used the GitHub Action public API to retrieve the latest 300 CI/CD run logs for each qualifying repository and identified success and failure run pairs with the same workflow ID. After filtering out cases where no suitable run pairs exist, we obtained 367 cases from 76 repositories, each manually analyzed to identify the key failing log lines and the root causes. The first 117 cases were used to train the *Drain* algorithm and build the few-shot log lines pool, with the remaining 250 cases used as the test set. Data collection was completed on April 25, 2025, and the dataset will be open-sourced on GitHub (see Appendix A).

*2) **Baseline Setup**:* According to our survey in Section II, LogSage is the first method specifically designed for CI/CD root cause analysis, and thus there are no existing baselines that can be directly compared. As the best available alternatives, we adopt LogPrompt [33] and LogGPT [35] as baselines, since they represent recent LLM-based approaches for general log analysis and anomaly detection. We do not compare against non-LLM methods, as such approaches are typically restricted to specific log formats or require training models from scratch, which contrasts with LogSage's training-free, plug-and-play integration into arbitrary CI/CD systems.

- **LogPrompt**: LogPrompt is an interpretable log analysis framework that utilizes prompt engineering techniques to guide LLMs in detecting anomalies without requiring any fine-tuning. It is designed for real-world online log analysis scenarios and emphasizes interpretability by generating natural language explanations alongside detection results. Evaluations across nine industrial log datasets show that LogPrompt outperforms traditional deep learning methods in zero-shot settings.
- **LogGPT**: LogGPT explores ChatGPT for log-based anomaly detection using a structured prompt-response architecture. It integrates chain-of-thought reasoning and domain knowledge injection to improve detection accuracy. The system generates structured diagnostic outputs with explanations and remediation suggestions. Experiments on benchmark datasets such as BGL demonstrate its competitiveness against state-of-the-art deep learning baselines under few-shot and zero-shot conditions.

We evaluate LogPrompt and LogGPT using their original prompt templates, with optimal settings: LogPrompt (few-shot = 20, window = 100) and LogGPT (few-shot = 5, window = 30). As shown in Table I, LogSage is the only method that supports the full pipeline from interpretable root cause analysis to automated solution execution. LogPrompt can produce interpretable diagnostic outputs but lacks the ability to generate solutions. Although LogGPT claims to generate remediation suggestions, it operates without any external knowledge integration and relies solely on the LLM's pre-trained knowledge, often resulting in hallucinated or impractical outputs.

TABLE I: Comparison of LogSage and Baseline Methods

Method	Anomaly Detection	Interpretable Root Cause	Solution Generation	File-level Processing	Automated Execution
LogSage	✓	✓	✓	✓	✓
LogGPT	✓	✓	○	×	×
LogPrompt	✓	✓	×	×	×

*3) **Metrics Description**:* We evaluate RCA task using standard metrics: Precision, Recall and F1-Score, with TP, FP, FN and TN defined specifically for this scenario as follows:

- **TP**: A correct detection. For LogSage, critical error log lines must overlap $\geq$ 90% with ground truth; for LogPrompt and LogGPT, they must fall within a predefined context window.
- **FP**: Detected lines that fail to meet the above criteria.
- **FN**: LogSage produces no output; LogPrompt/LogGPT fail to detect anomalies despite overlap.
- **TN**: Only applicable to LogPrompt and LogGPT when no detection intersects the context window.

B. *RQ1: Root Cause Analysis Precision Comparison*

To compare the RCA performance of LogSage with the two baseline methods and four prompt settings, we conducted experiments on three mainstream LLMs. The results are shown in Table II. We conducted Wilcoxon signed-rank tests comparing LogSage with each baseline variant, and all F1 score improvements are statistically significant ($p < 0.001$).

It is evident that LogSage performs consistently well across all models, with near-perfect scores and minimal fluctuation, indicating high precision and robustness in handling diverse CI/CD log formats, lengths, and error types. In contrast, LogPrompt's performance suffers in precision, while LogGPT shows significant variability, with both methods exhibiting low recall, which suggests that their window-based sampling might miss key contextual information, leading to errors in anomaly detection.

RQ1 Result: LogSage significantly outperforms the baseline methods in root cause analysis, demonstrating high precision, recall, and F1-score across various models, ensuring stability and reliability.

C. *RQ2: Root Cause Analysis Cost Comparison*

Considering the cost and time consumption associated with LLMs, we analyzed the token usage and query rounds for each method from the baseline experiments. The distribution of the data is shown in the violin plots Figure 4 and 5.

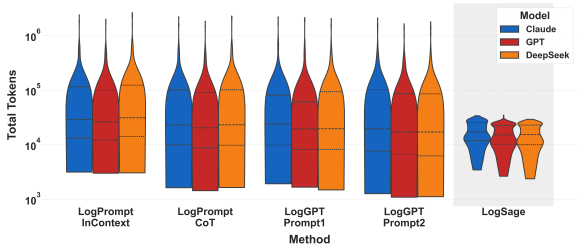

Fig. 4: Token usage for RCA across methods and LLMs.

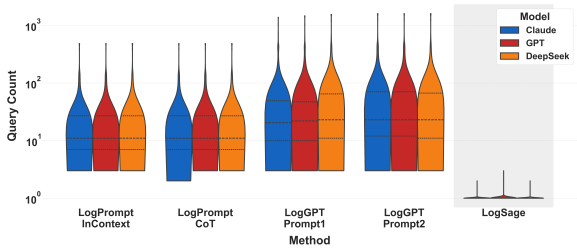

Fig. 5: Query rounds for RCA across methods and LLMs.

As shown in the token graph, the majority of CI/CD logs in the dataset have a length ranging from 10^4 to 10^5 tokens, with a long-tail distribution for some projects with unusually large logs. LogPrompt and LogGPT, which use stream-based processing through a sliding window without token length limits, show a rapid increase in token consumption as log length increases, leading to unnecessary cost in real-world scenarios. In contrast, LogSage not only achieves the best overall performance, but also benefits from a predefined token

TABLE II: Comparison of methods performance (Precision, Recall, F1-score) across different LLMs. * indicates F1 scores that are statistically significant at $p < 0.001$ compared with all baselines (Wilcoxon signed-rank test).

Group	Prompt Type	GPT-4o			Claude-3.7-Sonnet			Deepseek V3		
		P	R	F1	P	R	F1	P	R	F1
LogSage	N/A	0.9798	**1.0000**	**0.9898***	0.9837	0.9918	**0.9878***	**0.9838**	0.9959	**0.9898***
LogPrompt	CoT	0.8580	0.4654	0.6035	0.8467	0.3361	0.4812	0.8620	0.4477	0.5893
	InContext	0.6120	0.4530	0.5206	0.6864	0.4046	0.5091	0.6923	0.5018	0.5819
LogGPT	Prompt-1	0.7280	0.3081	0.4330	0.8140	0.3075	0.4464	0.8006	0.3394	0.4767
	Prompt-2	0.7185	0.3484	0.4693	0.7715	0.3073	0.4396	0.7380	0.4438	0.5543

limit, maintaining stable token consumption across different models and ensuring more predictable costs.

As shown in the query round graph, LogPrompt and LogGPT exhibit a linear relationship between query rounds and token length. For most logs, they require nearly 10 query rounds to complete root cause analysis, with some extreme cases needing up to 1,000 rounds. Such a high number of query rounds is not only highly inefficient but also prone to failure in real-world application scenarios. In comparison, LogSage efficiently limits the query rounds to around 1 for most cases, with minimal retries in edge cases, offering significant time advantages.

TABLE III: Efficiency Comparison Across Methods

Method	**Avg. Tokens**	**Avg. Queries**	**Token Variability**
LogSage	**17,853**	**1.0**	**14.46%**
LogPrompt	152,615	31.7	26.30%
LogGPT	122,353	89.4	19.00%

We also analyzed the average token consumption and query rounds across the methods in Table III. The average token consumption of LogPrompt and LogGPT are 152k and 122k tokens respectively, whereas LogSage maintains an average consumption of under 18k tokens—amounting to only 11.84% of LogPrompt's and 14.75% of LogGPT's. Moreover, LogSage demonstrates stable cross-model efficiency, with a normalized token variability of just 14.46%, significantly lower than LogPrompt's 26.30% and LogGPT's 19.00%.

In terms of query rounds, LogSage exhibits strong practical applicability by requiring only 1.0 query round on average to perform accurate and actionable RCA for CI/CD failures across the entire test set. In contrast, LogPrompt, benefiting from a large window size, requires an average of 31.7 queries, while LogGPT, due to its smaller window size, incurs an average of 89.4 queries, rendering it nearly infeasible in real-world scenarios.

RQ2 Result: LogSage efficiently completes root cause analysis in an average of 1 query round with only 11. 84% to 14. 75% token consumption compared to baseline methods, making it a highly cost-effective solution in real-world production environments.

V. Industrial Deployment Validation

We conducted a year-long online deployment and manual evaluation of LogSage to assess its accuracy and effectiveness in real-world CI/CD environments.

*1) **Integration Method**:* LogSage was directly embedded into the company's internal CI/CD platform (Fig. 8). When a CI/CD run failed, users could invoke LogSage from the failure page, review the diagnosed root cause and suggested fix, optionally trigger the repair tool, and provide feedback. Successful fixes automatically retriggered the pipeline, ensuring seamless integration and minimal workflow disruption.

*2) **Deployment Scope**:* Since May 2024, LogSage has processed **1,070,613** CI/CD failures across the company, serving **36,845** developers, and has been made available to all R&D teams using CI/CD services company-wide. Weekly active users exceeded **5,000** and coverage has stayed above **80%** since Oct 2024 (Fig. 6), showing broad and sustained adoption.

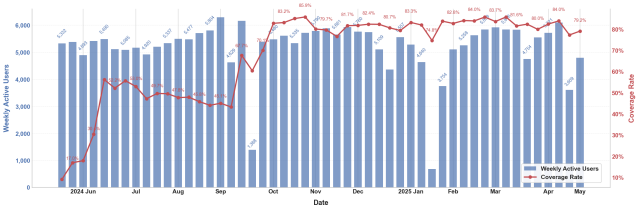

Fig. 6: Weekly active user count and coverage rate.

*3) **Online Effectiveness**:* During the online deployment phase of LogSage, we evaluated its two-stage accuracy through manual assessment and measured the availability of its four automated remediation tools via automatic logging.

Two-Stage Accuracy. The two-stage accuracy was evaluated based on weekly random sample of 150 online cases, which were manually reviewed and scored. For the RCA stage, experienced engineers examined the corresponding CI/CD task records and failure logs, manually debugging each case to determine the true root cause, which was then compared against LogSage's output. The evaluation criteria for the solution generation stage are summarized in appendix A, focusing primarily on the relevance of the generated solution to the actual root cause, the relevance of the retrieved knowledge, and the executability of the final recommendation.

The evaluation team independently assessed the outputs from Stage I and Stage II to measure the online accuracy of each stage. Due to the high cost of manual evaluation, assessments were conducted at fixed intervals only during the early rapid iteration phase of the project. The evaluation results from the 10 weeks' iteration period between August 2024 and October 2024 are shown in 7. As shown in the figure, LogSage achieves over **85%** RCA accuracy and over **80%** end-to-end accuracy in real-world deployment scenarios, clearly demonstrating its usability in production environments.

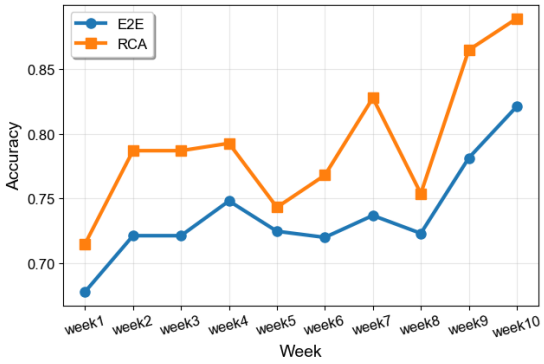

Fig. 7: Human-Annotated Online Accuracy

Auto-Repair Effectiveness. To assess the impact of auto-remediation, we collected engineering metrics on:

- Tool Coverage Rate: Ratio of cases where LogSage successfully recommended a repair tool over all cases that reached the solution generation stage.
- Re-run Success Rate: Proportion of re-runs that passed automatically after executing the suggested fix.

These metrics were gathered via system logging. The average tool coverage rate was 22.4%, with a median of 16.7%, reflecting that overall tool support across CI/CD failures remains partial and that there is significant room to expand coverage. Nevertheless, when tools were applicable, LogSage achieved a re-run success rate of 47.4% on average, with a maximum of 60.0%, median of 47.1%, and minimum of 36.8%. These results indicate that nearly half of the supported failures could be successfully resolved without human intervention—significantly reducing developer effort and turnaround time.

Taken together, these findings underscore both the practical feasibility and long-term potential of integrating automated repair into CI/CD workflows. As the first large-scale industrial deployment of its kind, LogSage demonstrates not only immediate operational value but also a promising direction for further improvement.

*4) **User Survey**:* Based on a recent survey of 191 front-line developers, LogSage received an average satisfaction score of 7.97 out of 10, reflecting strong acceptance and perceived utility in production workflows. Developers highlighted its ease of use, actionable suggestions, and seamless integration, while also providing feedback that has informed subsequent improvements.

VI. Conclusion & Future Work

In this paper, we present LogSage, the first end-to-end LLM-based framework for CI/CD failure diagnosis and automated remediation, validated through large-scale industrial deployment. LogSage operates in two complementary phases. In the offline preparation phase, it leverages log template extraction and enterprise knowledge integration to construct reusable references for diagnosis. In the online operational phase, it performs root cause analysis by filtering, expanding, and pruning failed logs, followed by dynamically assembled diagnostic prompts and multi-route knowledge retrieval. Finally, LogSage generates executable solutions through LLM-guided tool selection and automated reruns, enabling accurate, interpretable root cause analysis and effective remediation of complex CI/CD failures.

Empirical evaluations across multiple LLM backends and baselines show that LogSage achieves significant improvements in both performance and efficiency. It outperforms state-of-the-art LLM-based methods in RCA precision while reducing token consumption by over 85% compared to prior approaches. In industrial settings, LogSage demonstrates sustained adoption, processing over 1 million CI/CD failures and maintaining high user coverage with end-to-end precision exceeding 80%.

While encouraging, several aspects warrant discussion. The effectiveness of the *log diff* strategy relies on the repetitive nature of CI/CD pipelines: despite configuration heterogeneity, executions of the same pipeline remain largely stable across runs, allowing recurring outputs to be filtered regardless of system differences. Although our deployment used internal infrastructures such as Feishu documents and vector databases, the design is infrastructure-agnostic and can be instantiated with alternative knowledge bases in other organizations. Our baseline selection focused on LLM-based methods, as traditional CI/CD or AIOps approaches typically require training from scratch or depend on rigid log formats, making them less comparable to LogSage's training-free design. Finally, relying solely on LLMs for diagnosis introduces risks, as domain-specific semantics may not always be fully captured, leaving room for hallucinations or semantic gaps. These considerations highlight both the strengths and boundaries of LogSage, while pointing toward opportunities for broader integration.

In future work, we plan to upgrade LogSage into a more autonomous and adaptive LLM-Agent capable of orchestrating complex remediation workflows through iterative reasoning and proactive decision-making. We also aim to extend its scope beyond reactive failure handling to broader DevOps scenarios, including fault prediction, anomaly prevention, and automated incident response. These directions involve integrating with observability tools, modeling failure trends, and aligning with real-world operational workflows—pushing LogSage toward becoming an intelligent and proactive DevOps collaborator.

REFERENCES

[1] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda," *Journal of Systems and Software*, vol. 123, pp. 176–189, 2017. doi: 10.1016/j.jss.2015.06.063

[2] Microsoft Inside Track, "DevOps is sending engineering practices up in smoke", Microsoft, Apr. 15, 2024. [Online]. Available: https://www.microsoft.com/insidetrack/blog/devops-is-sending-engineering-practices-up-in-smoke/ [Accessed: May 10, 2025].

[3] Engineering at Meta, "Rapid release at massive scale", Meta Engineering Blog, Aug. 31, 2017. [Online]. Available: https://engineering.fb.com/2017/08/31/web/rapid-release-at-massive-scale/ [Accessed: May 10, 2025].

[4] GitHub Engineering, "Building organization-wide governance and re-use for CI/CD and automation with GitHub Actions", GitHub Blog, Apr. 5, 2023. [Online]. Available: https://github.blog/enterprise-software/devops/building-organization-wide-governance-and-re-use-for-ci-cd-and-automation-with-github-actions/ [Accessed: May 10, 2025].

[5] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects", in *Proc. 31st IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, Singapore, 2016, pp. 426–437. doi: 10.1145/2970276.2970358.

[6] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review", *arXiv preprint arXiv:2308.10620*, 2024. [Online]. Available: https://arxiv.org/abs/2308.10620

[7] A. Hemmat, M. Sharbaf, S. Kolahdouz-Rahimi, K. Lano, and S. Y. Tehrani, "Research directions for using LLM in software requirement engineering: A systematic review", *Frontiers in Computer Science*, vol. 7, 2025. doi: 10.3389/fcomp.2025.1519437.

[8] H. Li, X. Zhou, and Z. Shen, "Rewriting the code: A simple method for large language model augmented code search", in *Proc. 62nd Annu. Meeting Assoc. Comput. Linguistics (ACL)*, Bangkok, Thailand, Aug. 2024, pp. 1371–1389. doi: 10.18653/v1/2024.acl-long.75.

[9] T. Sun, J. Xu, Y. Li, Z. Yan, G. Zhang, L. Xie, L. Geng, Z. Wang, Y. Chen, Q. Lin, W. Duan, and K. Sui, "BitsAI-CR: Automated code review via LLM in practice", in *Proc. 33rd ACM Int. Conf. Found. Softw. Eng. (Ind. Track)*, 2025. *arXiv preprint arXiv:2501.15134*, 2024. [Online]. Available:https://arxiv.org/abs/2501.15134.

[10] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at Meta," in *Companion Proc. of the 32nd ACM Int. Conf. on the Foundations of Software Engineering (FSE 2024)*, Porto de Galinhas, Brazil, 2024, pp. 185–196. doi: 10.1145/3663529.3663839

[11] V.-H. Le and H. Zhang, "Log-based anomaly detection with deep learning: how far are we?," in *Proc. 44th Int. Conf. Software Engineering (ICSE)*, Pittsburgh, Pennsylvania, pp. 1356–1367, 2022. doi: 10.1145/3510003.3510155

[12] V.-H. Le and H. Zhang, "Log-based anomaly detection without log parsing," in *Proc. 36th IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, Melbourne, Australia, pp. 492–504, 2022. doi: 10.1109/ASE51524.2021.9678773

[13] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: anomaly detection and diagnosis from system logs through deep learning," in *Proc. 2017 ACM SIGSAC Conf. Computer and Communications Security (CCS)*, Dallas, Texas, USA, 2017, pp. 1285–1298. doi: 10.1145/3133956.3134015

[14] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou, "LogAnomaly: unsupervised detection of sequential and quantitative anomalies in unstructured logs," in *Proc. 28th Int. Joint Conf. Artificial Intelligence (IJCAI)*, 2019, pp. 4739–4745. Available: https://www.ijcai.org/proceedings/2019/0658.pdf

[15] L. Yang, J. Chen, Z. Wang, W. Wang, J. Jiang, X. Dong, and W. Zhang, "Semi-supervised log-based anomaly detection via probabilistic label estimation," in *Proc. 2021 IEEE/ACM 43rd Int. Conf. Software Engineering (ICSE)*, 2021, pp. 1448–1460. doi: 10.1109/ICSE43902.2021.00130

[16] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J.-G. Lou, M. Chintalapati, F. Shen, and D. Zhang, "Robust log-based anomaly detection on unstable log data," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Tallinn, Estonia, 2019, pp. 807–817. doi: 10.1145/3338906.3338931

[17] F. Hassan, N. Meng, and X. Wang, "UniLoc: Unified fault localization of continuous integration failures," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–31, Article 136, Sep. 2023. doi: 10.1145/3593799

[18] A. K. Arani, T. H. M. Le, M. Zahedi, and M. A. Babar, "Systematic literature review on application of learning-based approaches in continuous integration," *IEEE Access*, vol. 12, pp. 135419–135450, 2024. doi: ACCESS.2024.3424276

[19] A. S. Mohammed, V. R. Saddi, S. K. Gopal, S. Dhanasekaran, and M. S. Naruka, "AI-driven continuous integration and continuous deployment in software engineering," in *Proc. 2024 2nd Int. Conf. on Disruptive Technologies (ICDT)*, 2024, pp. 531–536. doi: 10.1109/ICDT61202.2024.10489475

[20] S. Lee, S. Hong, J. Yi, T. Kim, C.-J. Kim, and S. Yoo, "Classifying false positive static checker alarms in continuous integration using convolutional neural networks," in *Proc. 2019 12th IEEE Conf. on Software Testing, Validation and Verification (ICST)*, 2019, pp. 391–401. doi: 10.1109/ICST.2019.00048

[21] G. Grano, T. V. Titov, S. Panichella, and H. C. Gall, "How high will it be? Using machine learning models to predict branch coverage in automated testing," in *Proc. 2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, 2018, pp. 19–24. doi: 10.1109/MALTESQUE.2018.8368454

[22] G. Grano, T. V. Titov, S. Panichella, and H. C. Gall, "Branch coverage prediction in automated testing," *J. Softw. Evol. Process*, vol. 31, no. 9, pp. 1–18, Oct. 2019. doi: 10.1002/smr.2158

[23] K. Al-Sabbagh, M. Staron, M. Ochodek, and W. Meding, "Early prediction of test case verdict with bag-of-words vs. word embeddings," in *Proc. CEUR Workshop Proceedings*, Dec. 2019. Available: https://ceur-ws.org/Vol-2568/paper6.pdf

[24] K. W. Al-Sabbagh, M. Staron, R. Hebig, and W. Meding, "Predicting test case verdicts using textual analysis of committed code churns," in *Proc. Int. Workshop on Software Measurement and Int. Conf. on Software Process and Product Measurement (IWSM Mensura)*, Haarlem, The Netherlands, 2019, pp. 138–153. Available: https://ceur-ws.org/Vol-2476/paper7.pdf

[25] R. Abdalkareem, S. Mujahid, and E. Shihab, "A machine learning approach to improve the detection of CI skip commits," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2740–2754, 2021. doi: 10.1109/TSE.2020.2967380

[26] Y. Koroglu, A. Sen, D. Kutluay, A. Bayraktar, Y. Tosun, M. Cinar, and H. Kaya, "Defect prediction on a legacy industrial software: a case study on software with few defects," in *Proc. 4th Int. Workshop on Conducting Empirical Studies in Industry (CESI)*, Austin, TX, USA, 2016, pp. 14–20. doi: 10.1145/2896839.2896843

[27] R. Mamata, K. Smith, A. Azim, Y.-K. Chang, Q. Taiseef, R. Liscano, and G. Seferi, "Failure prediction using transfer learning in large-scale continuous integration environments," in *Proc. 32nd Annu. Int. Conf. on Computer Science and Software Engineering (CASCON)*, Toronto, Canada, 2022, pp. 193–198. Available: https://dl.acm.org/doi/10.5555/3566055.3566079

[28] A. Mishra and A. Sharma, "Deep learning based continuous integration and continuous delivery software defect prediction with effective optimization strategy," in *Knowledge-Based Systems*, vol. 296, p. 111835, 2024. [Online]. doi: 10.1016/j.knosys.2024.111835

[29] I. Saidani, A. Ouni, and M. W. Mkaouer, "Improving the prediction of continuous integration build failures using deep learning," in *Automated Software Engineering*, vol. 29, no. 1, pp. 1–61, May 2022. [Online]. doi: 10.1007/s10515-021-00319-5

[30] R. Mahindru, H. Kumar, and S. Bansal, "Log anomaly to resolution: AI based proactive incident remediation," in *Proc. 36th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, 2021, pp. 1353–1357. doi: 10.1109/ASE51524.2021.9678815

[31] Y. Xiao, V.-H. Le, and H. Zhang, "Demonstration-free: Towards more practical log parsing with large language models," in *Proc. 39th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, Sacramento, CA, USA, 2024, pp. 153–165. doi: 10.1145/3691620.3694994

[32] C. Egersdoerfer, D. Zhang, and D. Dai, "Early exploration of using ChatGPT for log-based anomaly detection on parallel file systems logs," in *Proc. 32nd Int. Symp. on High-Performance Parallel and Distributed Computing (HPDC '23)*, Orlando, FL, USA, 2023, pp. 315–316. [Online]. doi: 10.1145/3588195.3595943

[33] Y. Liu, S. Tao, W. Meng, J. Wang, W. Ma, Y. Chen, Y. Zhao, H. Yang, and Y. Jiang, "Interpretable online log analysis using large language

models with prompt strategies," in *Proc. 32nd IEEE/ACM Int. Conf. Program Comprehension (ICPC)*, Lisbon, Portugal, pp. 35–46, 2024. doi: 10.1145/3643916.3644408

[34] Y. Liu, S. Tao, W. Meng, F. Yao, X. Zhao, and H. Yang, "LogPrompt: Prompt engineering towards zero-shot and interpretable log analysis," in *Proc. 2024 IEEE/ACM 46th Int. Conf. on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2024, pp. 364–365. doi: 10.1145/3639478.3643108

[35] J. Qi, S. Huang, Z. Luan, S. Yang, C. Fung, H. Yang, D. Qian, J. Shang, Z. Xiao, and Z. Wu, "LogGPT: exploring ChatGPT for log-based anomaly detection," in *Proc. 2023 IEEE Int. Conf. High Performance Computing & Communications, Data Science & Systems, Smart City & Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, pp. 273–280, 2023. doi: 10.1109/HPCC-DSS-SmartCity-DependSys60770.2023.00045

[36] S. Shan, Y. Huo, Y. Su, Y. Li, D. Li, and Z. Zheng, "Face it yourselves: an LLM-based two-stage strategy to localize configuration errors via logs," in *Proc. 33rd ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA)*, Vienna, Austria, pp. 13–25, 2024. doi: 10.1145/3650212.3652106

[37] C. Almodovar, F. Sabrina, S. Karimi, and S. Azad, "LogFiT: Log anomaly detection using fine-tuned language models," in *IEEE Transactions on Network and Service Management*, vol. 21, no. 2, pp. 1715–1723, 2024. doi: 10.1109/TNSM.2024.3358730

[38] H. Ju, "Reliable online log parsing using large language models with retrieval-augmented generation," in *Proc. 2024 IEEE 35th Int. Symp. on Software Reliability Engineering Workshops (ISSREW)*, 2024, pp. 99–102. doi: 10.1109/ISSREW63542.2024.00055

[39] W. Guan, J. Cao, S. Qian, J. Gao, and C. Ouyang, "LogLLM: log-based anomaly detection using large language models," *arXiv preprint arXiv:2411.08561*, 2025. Available: https://arxiv.org/abs/2411.08561

[40] F. Hadadi, Q. Xu, D. Bianculli, and L. Briand, "LLM meets ML: Data-efficient anomaly detection on unseen unstable logs," *arXiv preprint arXiv:2406.07467*, 2025. Available: https://arxiv.org/abs/2406.07467

[41] A. Fariha, V. Gharavian, M. Makrehchi, S. Rahnamayan, S. Alwidian, and A. Azim, "Log anomaly detection by leveraging LLM-based parsing and embedding with attention mechanism," in *Proc. 2024 IEEE Canadian Conf. on Electrical and Computer Engineering (CCECE)*, 2024, pp. 859–863. doi: 10.1109/CCECE59415.2024.10667308

[42] M. He, T. Jia, C. Duan, H. Cai, Y. Li, and G. Huang, "LLMeLog: An approach for anomaly detection based on LLM-enriched log events," in *Proc. 2024 IEEE Int. Symp. on Software Reliability Engineering (ISSRE)*, pp. 132–143, 2024. doi: 10.1109/ISSRE62328.2024.00023

[43] L. Zhang, T. Jia, M. Jia, Y. Wu, A. Liu, Y. Yang, Z. Wu, X. Hu, P. S. Yu, and Y. Li, "A survey of AIOps for failure management in the era of large language models," *arXiv preprint arXiv:2406.11213*, 2024. Available: https://arxiv.org/abs/2406.11213

[44] D. Roy, X. Zhang, R. Bhave, C. Bansal, P. Las-Casas, R. Fonseca, and S. Rajmohan, "Exploring LLM-based agents for root cause analysis," in *Companion Proc. 32nd ACM Int. Conf. Found. Softw. Eng.*, Porto de Galinhas, Brazil, 2024, pp. 208–219. doi: 10.1145/3663529.3663841

[45] Z. Wang, Z. Liu, Y. Zhang, A. Zhong, J. Wang, F. Yin, L. Fan, L. Wu, and Q. Wen, "RCAgent: Cloud root cause analysis by autonomous agents with tool-augmented large language models," in *Proc. 33rd ACM Int. Conf. on Information and Knowledge Management (CIKM)*, Boise, ID, USA, 2024, pp. 4966–4974. doi: 10.1145/3627673.3680016

[46] X. Zhang, S. Ghosh, C. Bansal, R. Wang, M. Ma, Y. Kang, and S. Rajmohan, "Automated root causing of cloud incidents using in-context learning with GPT-4," in *Companion Proc. 32nd ACM Int. Conf. Found. Softw. Eng.*, Porto de Galinhas, Brazil, 2024, pp. 266–277. doi: 10.1145/3663529.3663846

[47] D. Goel, F. Husain, A. Singh, S. Ghosh, A. Parayil, C. Bansal, X. Zhang, and S. Rajmohan, "X-lifecycle learning for cloud incident management using LLMs," arXiv preprint, arXiv:2404.03662, 2024. Available: https://arxiv.org/abs/2404.03662

[48] K. Sarda, Z. Namrud, M. Litoiu, L. Shwartz, and I. Watts, "Leveraging large language models for the auto-remediation of microservice applications: An experimental study," in *Companion Proc. of the 32nd ACM Int. Conf. on the Foundations of Software Engineering (FSE)*, Porto de Galinhas, Brazil, 2024, pp. 358–369. doi: 10.1145/3663529.3663855

[49] K. Sarda, Z. Namrud, R. Rouf, H. Ahuja, M. Rasolroveicy, M. Litoiu, L. Shwartz, and I. Watts, "ADARMA: Auto-detection and auto-remediation of microservice anomalies by leveraging large language models," in *Proc. 33rd Annu. Int. Conf. on Computer Science and Software Engineering (CASCON)*, Las Vegas, NV, USA, 2023, pp. 200–205. doi: 10.5555/3615924.3615949

[50] T. Ahmed, S. Ghosh, C. Bansal, T. Zimmermann, X. Zhang, and S. Rajmohan, "Recommending root-cause and mitigation steps for cloud incidents using large language models," in *Proc. 45th Int. Conf. Software Engineering (ICSE)*, Melbourne, Victoria, Australia, pp. 1737–1749, 2023. doi: 10.1109/ICSE48619.2023.00149

[51] J. Wang, G. Chu, J. Wang, H. Sun, Q. Qi, Y. Wang, J. Qi, and J. Liao, "LogExpert: log-based recommended resolutions generation using large language model," in *Proc. 2024 ACM/IEEE 44th Int. Conf. Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, Lisbon, Portugal, pp. 42–46, 2024. doi: 10.1145/3639476.3639773

[52] P. Khlaisamniang, P. Khomduean, K. Saetan, and S. Wonglapsuwan, "Generative AI for self-healing systems," in *Proc. 2023 18th Int. Joint Symp. on Artificial Intelligence and Natural Language Processing (iSAI-NLP)*, 2023, pp. 1–6. doi: 10.1109/iSAI-NLP60301.2023.10354608

[53] D. Chaudhary, S. L. Vadlamani, D. Thomas, S. Nejati, and M. Sabetzadeh, "Developing a Llama-based chatbot for CI/CD question answering: A case study at Ericsson," in *Proc. 2024 IEEE Int. Conf. on Software Maintenance and Evolution (ICSME)*, Oct. 2024, pp. 707–718. doi: 10.1109/ICSME58944.2024.00075

[54] P. Sharma and M. S. Kulkarni, "A study on unlocking the potential of different AI in continuous integration and continuous delivery (CI/CD)," in *Proc. 2024 4th Int. Conf. on Innovative Practices in Technology and Management (ICIPTM)*, 2024, pp. 1–6. doi: 10.1109/ICIPTM59628.2024.10563618

[55] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *Proc. 2017 IEEE Int. Conf. on Web Services (ICWS)*, 2017, pp. 33–40. doi: 10.1109/ICWS.2017.13

[56] C. E. Brandt, A. Panichella, A. Zaidman, and M. Beller, "LogChunks: A data set for build log analysis," in *Proc. 17th Int. Conf. on Mining Software Repositories (MSR)*, Seoul, Republic of Korea, 2020, pp. 583–587. doi: 10.1145/3379597.3387485

APPENDIX A

Dataset Availability: The dataset is publicly available at https://github.com/ByteLuo1029/dataset.

Manual Evaluation Criteria for Solutions

2 Points (E2E Good Case):
- **Non-code issues:** The solution provides *direct and actionable instructions*, such as specific tools or recommended container images.
- **Code issues:** The solution clearly identifies the exact file and location of the issue based on logs, and proposes a concrete fix or a well-referenced example.

1 Point:
- **Non-code issues:** The solution is closely aligned with the root cause but only provides *indirect suggestions* (e.g., vague instructions without actionable configuration or tooling).
- **Code issues:** The solution references the relevant logs and proposes a plausible fix, but *fails to specify the exact file or code location*.

0 Point:
- The solution is irrelevant to the true root cause, lacks sufficient log grounding, or includes incorrect content possibly due to hallucinated knowledge (e.g., fabricated repository names).

Penalty:
- If any document link in the solution is invalid or broken, *deduct 1 point* from the overall score.

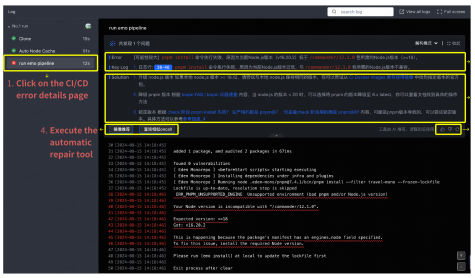

Fig. 8: Screenshot of the online user interface.