

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

---

```

1: Input: failed_log, success_log_templates {from offline
   preparation}
2: Output: filtered_log
3: Initialize candidate_pool  $\leftarrow \{\}$ 
4: for each line l in failed_log do
5:   template  $\leftarrow$  extract_template(l)
6:   position  $\leftarrow$  get_position(l, failed_log)
7:   if template  $\notin$  success_log_templates then
8:     Add l to candidate_pool {log diff}
9:   end if
10:  if contains_keyword(l) then
11:    Add l to candidate_pool {keyword matching}
12:  end if
13:  if is_in_log_tail(position) then
14:    Add l to candidate_pool {log tail prioritization}
15:  end if
16: end for
17: filtered_log  $\leftarrow$  deduplicate(candidate_pool)
18: 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*  $L = \{l_1, l_2, \dots, l_n\}$  and log lines from the candidate pool  $I = \{i_1, i_2, \dots, i_m\}$ , we define a weight list  $W = \{w_1, w_2, \dots, w_n\}$ , where each  $w_j$  represents the weight assigned to line  $l_j$ :

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

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  $\alpha$  and  $\beta$  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  $\theta$  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  $\theta$  is adaptively defined by Equation 2 according to two different situations:

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

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  $\gamma$ , 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:

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

After computing the weight density for all candidate log blocks, they are ranked in descending order of density and indexed as  $B_1, B_2, \dots, B_K$ , 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:

<code>query2doc_viking_knn</code>	<code>query2query_viking_knn</code>
<code>query2doc_es_knn</code>	<code>query2doc_es_keyword</code>
<code>query2query_es_keyword</code>	<code>query2query_es_knn</code>
<code>lark_wiki</code>	<code>rds_match</code>

**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.