Automated Proactive Logging Quality Improvement for Large-Scale Codebases

Yichen Li[¶], Jinyang Liu[¶], Junsong Pu[*], Zhihan Jiang[‡], Zhuangbin Chen[*],
Xiao He[¶], Tieying Zhang[¶†], Jianjun Chen[¶], Yi Li[¶], Rui Shi[¶], Michael R. Lyu[‡],
[¶]ByteDance, {liyichen.325, jinyang.liu, xiao.hx, tieying.zhang, jianjun.chen, liyi.ly, shirui}@bytedance.com
[‡]The Chinese University of Hong Kong, {zhjiang22, lyu}@cse.cuhk.edu.hk
[*]Sun Yat-sen University, pujs@mail2.sysu.edu.cn, chenzhb36@mail.sysu.edu.cn

***Abstract*—High-quality logging is critical for the reliability of cloud services, yet the industrial process for improving it is typically manual, reactive, and unscalable. Existing automated tools inherit this reactive nature, failing to answer the crucial *whether-to-log* question and are constrained to simple logging statement insertion, thus addressing only a fraction of the real-world logging improvement.**

To address these gaps and cope with logging debt in large-scale codebases, we propose LOGIMPROVER, a framework powered by Large Language Models (LLMs) that automates proactive logging quality improvement. LOGIMPROVER introduces two paradigm shifts: from reactive generation to proactive discovery, and from simple insertion to holistic logging patch generation. First, it identifies potential logging gaps based on principles distilled from industrial best practices. Then, it grounds each candidate through a cascading, structure-aware RAG module. Next, it prunes false positives by analyzing call-stack logging responsibilities and implicit logger inheritance. Finally, it generates holistic and explainable logging patches that reflect real-world development practices.

Our evaluation provides dual confirmation of its effectiveness: LOGIMPROVER significantly outperforms state-of-the-art baselines in closed-world experiments and achieves 68.12% developer acceptance rate in its real-world deployment. This success demonstrates the practical value of automating the entire logging quality improvement lifecycle, from discovery to recommendation.

***Index Terms*—Logging Debt; Observability; Code Generation.**

I. INTRODUCTION

Reliability is essential for cloud services, as it ensures consistent availability and expected performance, thereby preventing revenue loss and maintaining user trust [1]. Observability, which refers to the ability to monitor and understand a system's internal states through the runtime data, such as logs, traces and metrics, plays a vital role in maintaining the reliability. Logs, in particular, are often the primary resource for on-call engineers to diagnose and resolve production issues [2], [3], [4], [5], [6], making the quality of logging instrumentation a cornerstone of system maintainability.

Despite its importance, the process of improving logging quality is often inefficient and unscalable. The typical industrial process is manual, meeting-driven, and fundamentally reactive. Developers, Site Reliability Engineers (SREs), and other teams engage in regular meetings for post-incident reviews and best practice discussions to identify logging gaps. This manual-first approach cannot keep pace with the rapid evolution of large-scale codebases, creating a clear need for automated solutions to systematically address what we term *logging debt* [7].

While the need for a proactive solution is clear, existing academic studies for suggesting *where-to-log* [7], [8] or *what-to-log* [9], [10], [11] are fundamentally reactive; they require a developer to first select a specific method to improve. Simply applying these tools across an entire codebase is not practical. This is because **observability is not a free lunch**: every added logging statement incurs more I/O, storage, and computational costs [12], [13], [14]. The cost of logging indicates that an automated tool cannot simply insert logging statement everywhere. It must be both precise in identifying where a new logging statement is needed and holistic in its ability to improve existing logs. This reality exposes two fundamental limitations in existing tools.

The first limitation stems from the reactive design of current tools. This is particularly problematic given the sparse nature of logging, where our analysis reveals that only 14.0% of methods contain a logging statement (detailed in Section II-C). Because these models are designed to generate a logging statement, they are inherently biased towards insertion and lack the pruning mechanisms for proactive discovery. This observation motivates our **first paradigm shift**: from reactive generation to proactive discovery.

The second limitation is the narrow scope of existing tools, which focus exclusively on inserting new logging statements. This focus is misaligned with how developers improve logging quality in practice. Our study found that insertion accounts for only 44.1% of logging-related code changes (detailed in Section II-C). By modeling only insertion, current tools fail to learn from the patterns embedded in real-world logging improvements, which prevents them from generating practical suggestions. This finding motivates our **second paradigm shift**: from insertion to holistic logging patch generation.

To address these limitations, we propose LOGIMPROVER, an automated proactive logging quality improvement framework powered by Large Language Models (LLMs). LOGIMPROVER realizes our two paradigm shifts through a multi-stage pipeline that mirrors the reasoning of an expert SRE. The pipeline begins with a best-practice-driven scan to proactively detect

[†]Tieying Zhang is the corresponding author.

potential logging missing, directly tackling the *whether-to-log* question at scale. Next, a structure-aware RAG module contextualizes each identified candidate. LOGIMPROVER retrieves similar cases from constructed knowledge base containing post-incident fixes and industrial show house repositories. These examples demonstrate the silent practice beyond written rules Finally, a pruning agent refines the results by analyzing call stack logging responsibility and implicit logger inheritance. This step filters out false positives, such as cases where logging is already handled outside the method. The final output is a set of explainable, multi-line logging patches.

Our evaluation provides dual confirmation of its effectiveness. In closed-world experiments, LOGIMPROVER significantly outperforms state-of-the-art baselines across all tasks. More critically, its industrial deployment at ByteDance achieved a developer acceptance rate of 68.12%. Ultimately, LOGIMPROVER inverts the traditional mode by automating the entire process from discovery to recommendation, answering the two fundamental questions: (*i*) whether and where a logging improvement is needed, and (*ii*) what specific, high-quality, and convincing recommendation should be provided.

In summary, our key contributions are:

- **Novelty:** We are the first to shift logging automation from reactive, single-action insertion to proactive, holistic logging patch generation.
- **Industrial Practice Study:** We present an in-depth study of logging quality improvement in a industrial scenario, sharing real-world deployment experiences, lessons learned and insights.
- **Methodology:** We design and build LOGIMPROVER, an end-to-end LLM-powered framework with a four-stage pipeline, integrating proactive scanning with context-aware retrieval from knowledge bases to generate explainable, high-quality logging patches.
- **Evaluation:** We demonstrate the superiority and practical value of LOGIMPROVER through evaluation in both closed-world and real-world environments. Our results show state-of-the-art performance and achieve 68.12% developer acceptance rate in real-world deployment.

II. INDUSTRIAL LOGGING QUALITY IMPROVEMENT: EXPERIENCE & REQUIREMENT

To motivate our work, we first illustrate the severe consequences of inadequate logging through a real-world industrial incident. We then analyze the limitations of the current manual, reactive processes for improving logging quality. Finally, we present an empirical study of logging practices at ByteDance to demonstrate practical requirements for an automated logging quality improvement tool.

A. The Cost of Insufficient Logging: A Motivating Case

We present a real-world incident from the service discovery component of our internal system X to illustrate how insufficient logging can lead to significant operational costs. The system uses a centralized aggregator for multi-cluster service discovery. As described in Figure 1, a central Aggregator

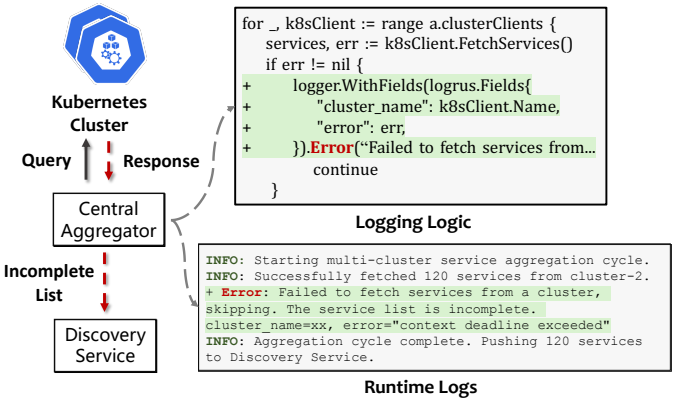


Fig. 1: The cost of missing logging statement.

queries all Kubernetes clusters and merges their responses into a list. This list is then cached by a Discovery Service, which acts as the source of truth for the system. The system relies on one key assumption: if a service is missing from the aggregator's list, it is considered deleted. Therefore, the completeness of the aggregator's data is important.

The incident began with a routine resource scaling operation, which caused a brief communication failure between the aggregator and one of the Kubernetes clusters. Although the aggregator caught the error, it did not log any warning or message. Instead, it silently skipped the failed cluster and continued processing. This missing log made the failure completely invisible to engineers and monitoring tools. From the outside, the aggregator appeared to be functioning normally, making it extremely difficult to detect that a cluster had been skipped. The aggregator then returned an incomplete list of services to the discovery service. Following its logic, the discovery service assumed the missing services had been intentionally deleted. This led to a system-wide outage: the system X could no longer find or connect to those services, even though they were still running. This incident highlights how the absence of even a single log entry can obscure critical failures and delay root cause analysis [15].

Motivation 1: Insufficient logging leads to silent failures, which can convert minor issues into costly, system-wide outages, underscoring the critical need for automated methods to improve logging debt.

B. The Manual Logging Improvement Process

While the importance of logging quality is acknowledged, and the diffusion of responsibilities among roles impedes logging quality governance, often leading to observability initiatives being de-prioritized across most services. This situation highlights the need for a fundamental shift in the improvement strategy.

As shown in Figure 2, the current industrial process for improving observability is highly dependent on manual efforts, typically organized through recurring review meetings by the reliability team (*i.e.*, SREs). This process involves a complex interaction between multiple roles. Specifically, developers are responsible for implementing concrete logging code during

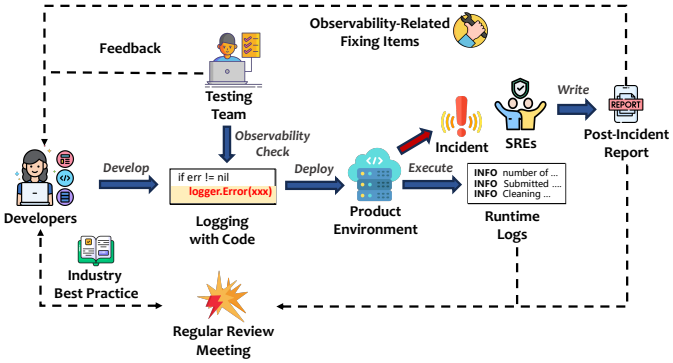


Fig. 2: The traditional, meeting-driven process for observability improvement.

```
...
+ logs.Logger.WithFields(logrus.Fields{"object_type": fmt.Sprintf("%T",
obj)}).Errorf("failed to get object GVK: %v", err)
...
```

(a) Example of logging statement insertion.

```
if err != nil {
-   logger.Errorf(err)
+   logger.WithFields(logrus.Fields{"resource": resource.Name,
"namespace": getObjNamespace(obj)}).Errorf(err)...
```

(b) Example of logging statement modification.

```
-   return nil, err
+   return nil, fmt.Errorf("failed to get modified configuration for %s:
%w", info.Name, err)        - - → Thrown for logging
```

(c) Example of logging variable enhancement.

Fig. 3: Logging quality improvement examples.

development. The testing team is then tasked with verifying the accuracy and stability of these outcomes; however, unlike functional testing which can yield precise results, this often involves a high-level manual review and subjective advice. Subsequently, SREs drive improvements from a reactive standpoint. After handling cloud incidents [1], [16], they review the root cause, evaluate metrics like Time to Detect (TTD) and Time to Resolve (TTR) [1], and provide direct insights into missing observability points through post-incident reports.

However, this manual, meeting-driven approach proves to be inefficient and unscalable for vast and evolving codebases. The high coordination overhead required for cross-team reviews makes manual governance impractical. As a result, the process becomes reactive in practice. This reveals the limitation: written best practices lack real contextualized knowledge. Therefore, there is an urgent need for an automated solution capable of mining the *silent practices*: the patterns learned from the reviewed codebase and incident-driven improvements.

Motivation 2: Current logging improvement relies on a manual, unscalable and reactive process. A proactive, automated practice-following approach is required to systematically address logging debt.

C. An Empirical Study of Industrial Logging Practices

To ground our research in real-world challenges, we conducted an empirical study of four large-scale industrial systems at ByteDance (anonymized as System-A, B, C, and D). As detailed in Table I, these are mature, global-facing cloud services built on a collective 3.8M lines of Go code. We specifically selected these systems because their logging practices are considered a benchmark for quality and reliability. Having undergone years of evolution in a live production environment, their logs have been continuously refined to effectively meet the demanding requirements of operational analysis, troubleshooting, and system monitoring.

Our analysis relies on data gathered from two sources: the current state of the source code and its historical evolution. To analyze the code's structure, we used our internal Go static analyzer to parse the 3.8M lines of code. The analyzer systematically constructed an Abstract Syntax Tree (AST) to identify all method declarations and, within each method body, detected invocations to logging libraries. To analyze how developers improve logging, we mined the complete Git commit history for each system. We identified logging-related commits by programmatically analyzing the diff of each change, isolating those that specifically added or modified lines containing logging statements. We then filtered out commits that also contained functional code changes using an LLM, thereby isolating commits that represent pure logging quality improvements. This comprehensive data allowed us to observe two essential gaps in logging strategies that limit existing tools and motivate our research.

Logging is Sparse. Our first finding is that logging statements are sparsely distributed. As Table I shows, across all methods in our dataset, only 14.0% contain logging statements. This sparsity is a direct consequence of Go's idiomatic error handling. Since errors (*i.e.*, `err` in Go language) are treated as first-class values [17], a common and accepted practice is for developers to propagate them up the call stack, often enhancing them with more context, rather than logging them immediately at their point of origin. This practice makes it extremely challenging for any automated tool to proactively determine where a log statement is truly required.

Diverse Logging Improvement Patterns. Our second finding, derived from analyzing logging improvement commits distribution, reveals that real-world improvement are not monolithic. As Table I shows, *insertion* of new log statements accounted for only 44.1% of these changes. As shown in Figure 3, the logging quality improvement patterns are diverse. The majority of developer effort was invested in more nuanced tasks: *modification* of existing logs (*e.g.*, adding contextual key-value pairs, as shown in Figure 3b) and *error enhancement* (*e.g.*, improving variable propagation wrapping, as shown in Figure 3c). This finding highlights a critical gap: existing automated tools, with their narrow focus on insertion, fail to address the majority of real-world logging improvement needs.

Our empirical findings highlight a fundamental disconnect with existing automated tools, proposing practical requirements for new paradigms:

TABLE I: A Summary of the Logging Practice of Studied Systems

Systems	LOC	# Logging Statements	# Methods	# Methods with Logging Statements	Logging Change Action Rate %		
					Insertion	Modification	Variable Enhancement
System-A	2.6M	6,163	31,486	3,180	41.5	6.9	51.6
System-B	43k	386	1,262	149	38.7	3.4	57.0
System-C	314k	830	2,984	387	33.0	3.5	63.5
System-D	845k	5,824	10,509	2,738	49.0	9.1	41.9
Total	3.8M	13203	46241	6454	44.1	7.5	48.4

Practical Requirement: An effective solution must replace the current manual process (Section II-B) with an automated approach driven by contextualized practices learned from the codebase. This approach must be *(i)* Precise in order to address the sparse nature of logging (Section II-C) and prevent costly silent failures (Section II-A); *(ii)* Holistic, moving beyond insertion to generate comprehensive logging patches (Section II-C), thus helping developers improve logging quality in practice.

III. Methodology

A. Overview

To address the challenges laid out in our empirical study, we propose LogImprover, a framework that embodies our two paradigm shifts. Our methodology is explicitly designed to overcome the two fundamental limitations of prior automated tools: their inability to answer the *whether-to-log* question and their narrow focus on logging statement insertion.

As illustrated in Figure 4, the pipeline begins with *Proactive Candidate Identification*, which scans the entire codebase against best practices to find all potential improvement candidates with high recall. These candidates then enter the *Cascading Contextualized Refinement* stage, where they are enriched with contextual knowledge from positive and negative knowledge base to improve precision. Subsequently, the *Inheritance-Aware False Positive Pruning* stage employs a specialized agent to analyze the service's custom logger infrastructure, pruning suggestions that are redundant due to implicit logging behavior inherited from wrappers or base classes. In the final stage, *Holistic Logging Patch Generation*, LogImprover synthesizes all gathered evidence to generate a holistic and concrete faceted logging patch and a detailed, evidence-based explanation.

B. Stage 1: Proactive Candidate Identification

A fundamental limitation of previous works [10], [18], [9] is its reactive nature, which sidesteps the critical *whether-to-log* question. LogImprover inverts this paradigm by initiating a proactive screening process. At its core, this stage uses a set of codified best practices [19], [20], which are distilled from the collective experience of expert SREs and post-incident reviews by years, to identify potential logging gaps.

The primary objective of this stage is to act as a powerful focusing mechanism. Large-scale codebases contain vast amounts of boilerplate code where logging is unnecessary [7]. To avoid the cost of deeply analyzing every method, we leverage a comprehensive set of industrial best practices. As shown in Figure 5, each principle within these practices targets a specific type of critical operation, such as data access. To ensure high recall, these principles are translated into lenient, syntax-based patterns. For example, the pattern for Principle 1 flags any unlogged error block following a function call with data-access substrings (*e.g.*,, `DataBase`, `Cache`). This strategy allows LogImprover to efficiently pinpoint methods that perform critical operations while filtering out irrelevant code, ensuring that all candidates with high potential are identified for subsequent stages.

LogImprover applies this screening process across two scopes: scanning entire legacy repositories to address accumulated logging debt, and incrementally analyzing new merge requests to prevent new gaps. The resulting high-recall, focused set of candidates is then passed to the subsequent stages for further contextualized refinement and false positive pruning.

C. Stage 2: Cascading Contextualized Refinement

While Stage 1 excels at recall, its reliance on generalized patterns inevitably introduces a significant number of logging noises. Presenting developers with generic suggestions (*e.g.*, `Please log errors`) would hurt trust, as abstract principles are often difficult to apply correctly and are easy to dismiss. To convince developers to act, we must move from the abstract to the concrete. This means substantiating every potential violation with evidence: a concrete, peer-validated code change from a similar context. Therefore, the goal of this stage is to shift from rule-based flagging to evidence-based refinement. By finding the most contextually similar cases for each candidate, we ground our suggestions in the existing "silent practices" of the organization's own codebase.

To bridge this gap, LogImprover grounds its recommendations by constructing and leveraging two continuously evolving method-level knowledge bases:

- **The Positive Knowledge Base (K_{pos})** serves as a collection of validated, high-quality logging implementations in a specific code change or method, sourced from: (i) *Post-incident patches*, which capture proven logging quality fixes for real-world incidents for better diagnosis; (ii) *"Show House" repositories*, which represent mature codebases reviewered by experts that serve as the gold standard; and (iii) *Accepted recommendations* from LogImprover itself, creating a powerful self-

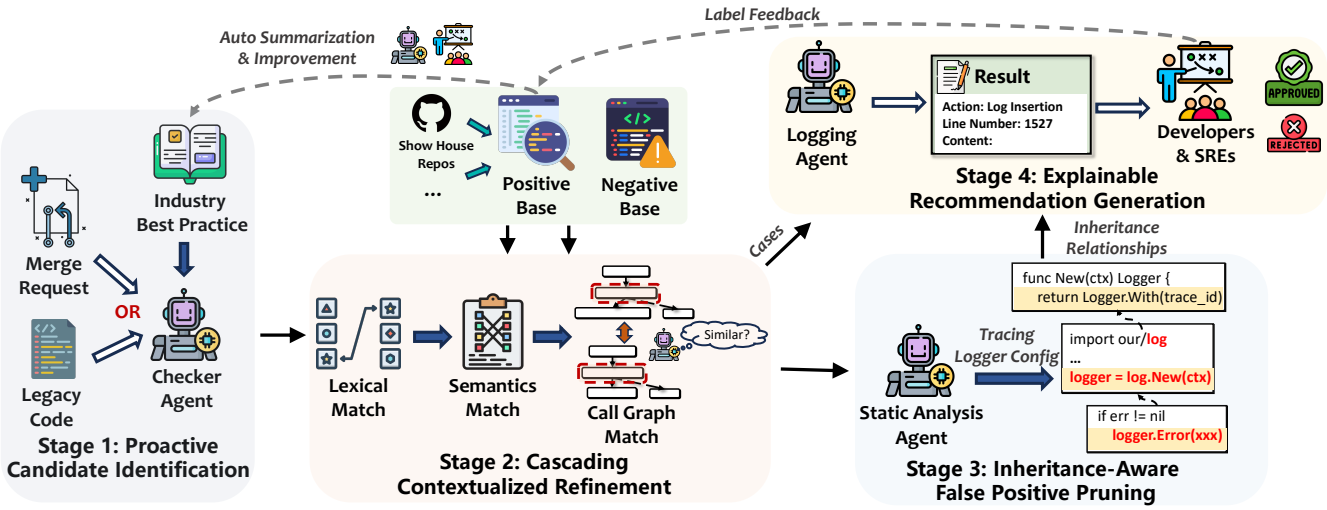


Fig. 4: The workflow of LOGIMPROVER.

Principle 1 (Data Access Failure): Failures to access a primary data source (e.g., database) require an ERROR log, while failures for a recoverable source (e.g., cache) require a WARN log.

Principle 2 (Asynchronous Failure): Errors within asynchronous tasks (e.g., goroutines) must be logged to prevent silent failures and inconsistent system states.

...

Fig. 5: The internal best practice example for logging instrumentation.

reinforcing learning loop by taking real-world deployment feedbacks.

- **The Negative Knowledge Base** (K_{neg}) is populated with explicitly rejected recommendations. This provides an unambiguous signal that prevents the system from repeating suggestions that developers have found to be irrelevant or incorrect.

Crucially, every entry in these knowledge bases is linked to its source in the repository. This ensures that each candidate can be traced back to retrieve contextualized information, such as its callers and callees, for the following process. Furthermore, each entry is a method-level artifact that pairs the contextualized code change with its textual data, such as a post-incident review explaining its diagnostic value.

However, effectively retrieving the most relevant examples from large-scale codebases poses a significant challenge. Simple lexical or semantic similarity is insufficient[21], [22], [23], as two methods can have similar text and function while operating in vastly different architectural contexts. To overcome this, LOGIMPROVER employs a *Cascading Contextualized Refinement* pipeline: a multi-stage Retrieval-Augmented Generation (RAG) [24] strategy designed to balance efficiency and contextualized precision.

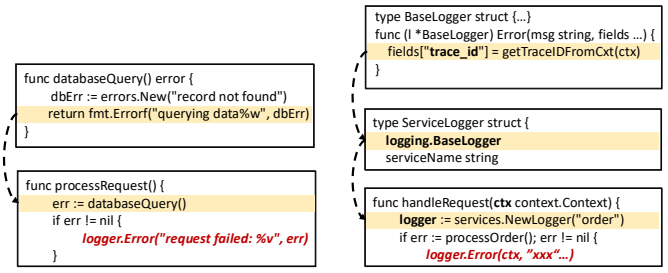
1) **Phase 1: Lexical Filtering.** This initial phase acts as a high-speed, coarse-grained filter. It uses sparse retrieval methods (i.e., BM25 [25]) to rapidly scan the knowledge bases, reducing the search space from potentially millions of entries to a few hundred candidates that share significant identifier overlap with the target method.
2) **Phase 2: Semantic Re-ranking.** The candidates from Phase 1 are then re-ranked using dense retrieval. By computing embeddings with a fine-tuned model, this phase moves beyond keyword matching to identify precedents with a similar functional intent, re-ordering the candidates based on their semantic relevance.
3) **Phase 3: Dependency-Aware Contextual Re-ranking.** This final, crucial phase ensures true contextualized similarity. Recognizing that even semantically similar methods can be inappropriate due to differing contexts, we employ an LLM to perform a structural comparison. For the top candidates, we construct local call graphs for both the candidate and the retrieved examples. The LLM is prompted to assess the structural isomorphism of these graphs, considering upstream callers, downstream callees, and error-handling logic. A high score is assigned only when a contextual match is found.

This cascading process ensures that the final selected precedent is not just functionally similar but contextualized similarly to the candidate method. This match is the cornerstone of LOGIMPROVER's ability to generate a trustworthy recommendation, coupled with concrete, approved cases drawn from post-incident patches, mature and reviewed repositories, or previously accepted suggestions.

D. Stage 3: Inheritance-Aware False Positive Pruning

The candidates refined in Stage 2, while contextually relevant, can still be incorrect due to sophisticated logging patterns in practice. The primary aim of this stage is to prune these remaining false positives by analyzing the candidate from two distinct perspectives: its logging responsibility within the call stack and the implicit pattern of its logging framework.

The first source of false positives arises from the division of logging responsibility across the call stack [7]. In practice, as illustrated in Figure 6a, if a caller function is responsible for



(a) An example of logging responsibility.

(b) An example of logger inheritance.

Fig. 6: Two noise-introducing logging patterns.

logging an error, the callee could only enhance the logger variable (*e.g.*, via `fmt.Errorf("...: %w", err)`) rather than generating another statement. Therefore, LOGIMPROVER prunes candidates where an upstream function in the call stack already provides adequate logging for the same error path according to the corresponding logging boundaries.

The second source of false positives comes from modern logging frameworks that automatically enrich log entries with contextual data (*i.e.*, `ctx` in Go language). As illustrated in Figure 6b, a specialized logger is configured automatically inject fields like `trace_id` into every inheritanced logger. A model that only observes the call to `logger.Error(...)` will fail to see this implicit behavior and will incorrectly suggest adding the `trace_id`, introducing the noise.

To address this, we introduce the **Log Config Agent**, a hybrid component that surpasses the precision limitations of traditional static analysis by combining it with LLM-driven refinement. While static analysis has been explored for logging [10], it can struggle with dependency injection or reflection, leading to inaccuracies. Our agent overcomes this by combing it with semantics understanding capability of LLM.

First, the agent performs a static tracing process, starting from the logger instance in the candidate method. It follows all possible inheritance and composition chains to identify every inheritance *path*. This initial step is designed for high recall, ensuring no potential inheritance path is missed.

However, static analysis alone cannot definitively validate these paths. Therefore, the agent sends the discovered paths to an LLM for refinement. The input to the LLM consists of the full inheritance paths along with the source code for each class or struct in those paths. The LLM analyzes the provided code to determine which of the paths are actually viable and what fields (*e.g.*, `trace_id`) are implicitly added. The final output is a validated profile of the logger's automatic behavior.

By creating this definitive profile, this two-fold pruning strategy ensures that the final recommendations are not only contextually relevant but also aware of both repo-level logging conventions and framework-level implicit behavior.

E. Stage 4: Holistic Logging Patch Generation

This final stage synthesizes all the gathered evidence into a concrete, actionable logging patch. It directly fulfills our second paradigm shift, moving from simple insertion to diverse logging improvement patterns a s we detailed in Section II-C.

As demonstrated by our findings in Figure 3 and Table I, real-world logging requires a mix of insertion, modification, and variable enhancement. To mirror this reality, we define a corresponding *action space*. The model's decision of which action to take is not based on a simple rule, but on a holistic synthesis of the knowledge gained throughout the pipeline. This knowledge integrates the specific best practice violated from Stage 1 with the most contextually similar cases retrieved in Stage 2, including its source code, rationale, and call graph analysis. It is then informed by the final validity checks from Stage 3, which confirm both the call stack's logging responsibility and the logger's implicit behavior.

Based on this informed decision, the LLM generates a concrete logging patch and a corresponding explanation, falling into one of the three categories of our action space:

- **Logging Statement Insertion**: Suggests the insertion of new logging statements into critical code paths, as shown in Figure 3a.
- **Logging Statement Modification**: Refines existing logging statements to improve their quality. This includes adding more contextual key-value pairs (*e.g.*, via `log.withFields`), as shown in Figure 3b.
- **Logging Variable Enhancement**: Improves variable propagation wrapping. Instead of returning raw variables, LOGIMPROVER suggests wrapping them with operational details (*e.g.*, using `fmt.Errorf("...: %w", err)`), as shown in Figure 3c.

Furthermore, the effectiveness of this entire process relies on its ability to continuously learn and adapt. Developer feedback directly updates the positive (K_{pos}) and negative (K_{neg}) knowledge bases for immediate use in RAG. Over time, as feedback accumulates, we will further employ alignment techniques, such as Direct Preference Optimization (DPO) [26], to tune the LLMs so that they can better internalize the real preferences of developers. This multi-tiered learning loop ensures that LOGIMPROVER remains responsive to immediate needs while continuously evolving its understanding of logging practices.

IV. EXPERIMENT SETUP

Our evaluation is divided into *(i)* a controlled, closed-world experiment to precisely measure performance against existing baselines, and *(ii)* a real-world deployment study to evaluate practical utility and developer acceptance in the wild.

Our evaluation is guided by the following RQs:

- **RQ1**: How effective is LOGIMPROVER at logging quality improvement?
- **RQ2:** How do the different phases affect the effectiveness of LOGIMPROVER?

A. Dataset

Our evaluation dataset is derived from a comprehensive snapshot of 10 mature, actively-maintained cloud services within the company with their corresponding repositories. To

ensure high data quality and minimize potential noise, we first applied a series of filtering rules:

- We removed duplicate files and code generated automatically by frameworks (*e.g.*, Thrift[27]-generated files) to reduce noise in the dataset.
- Files containing non-English logs were excluded to maintain language consistency.
- We selected mature repositories that are still maintained and have been running online for more than one year.

This rigorous process yielded a high-quality dataset of 20,525 non-auto-generated Go methods, which serve as the foundation for our experiments. Both the retrieval and tuning-based components of our baselines use them as training set. From this foundation, we constructed the ground-truth test sets for our three logging improvement tasks as follows:

For the *logging statement insertion* task, we created a large-scale test set by systematically removing all 7,703 existing logging statements from the codebase, following the prior works [28], [10], [11]. This process is straightforward and provides a substantial dataset where the model's task is to correctly predict the removed statements.

For the *logging statement modification* and *logging variable enhancement* tasks, constructing the ground truth is far more challenging, as it requires identifying real improvements from historical data. Simple removal is not an option. Instead, we reuse the mining process in Section II-C. This comprehensive process yielded a smaller but highly valuable and realistic set of ground truths: 153 confirmed logging modification changes and 848 confirmed logging variable enhancement changes. These serve as the positive examples for their respective tasks. To construct a test set that realistically evaluates the decision-making capability, we then introduced negative samples. For each positive case, we sampled seven methods that already contained logging statements (or variables) but did not require modification or enhancement, according to the real-world distribution learned in Section II-C. While these numbers are smaller, they represent real, developer-authored logging improvements.

B. Baselines and Evaluation Scenarios

Due to the lack of logging quality improvement (*e.g.*, modification) baselines, we primarily include logging statement insertion methods to evaluate the effectiveness of LOGIMPROVER. For confidentiality and safety reasons, LOGIMPROVER and all LLM-based models share the same internal LLM backbone: *Doubao-Seed-1.6-thinking*.

We selected SCLogger [10], LANCE [9] (tuned on training set) and the raw LLM model and raw LLM with typical RAG capability [18] as our primary baselines. It is important to note that none of these baselines have the capability to conduct *whether-to-log* task and other logging patch generation tasks; they simply insert logging statement when given a method. For LLM-based baselines, we provided the relevant instructions for LLM (voted by developers in ByteDance), such as *Evaluate whether to insert logging statement to this method according to the practices you have learned*. For models that cannot take instructions, we tuned the model on the training dataset for *whether-to-log* task.

C. Evaluation Metrics

Following existing works [10], [11], we evaluate each suggestion by comparing it against the ground truth across a sequential, three-stage process: (1) Decision, (2) Position, and (3) Content. The evaluation of each subsequent stage is strictly **conditioned on the success of the previous one**. This cascading evaluation logic means that a suggestion is only evaluated for its *Position* if its *Decision* is correct. If the model incorrectly decides to insert a logging statement, the subsequent stages for that candidate are not considered. Similarly, the *Content* metrics are only measured if the logging statement is placed at the correct *Position*.

- **Decision:** The task of deciding *whether* to give the suggestion in a given method. We treat this as a binary classification problem and report the *F1-score* as *D-F1* to balance precision and recall.
- **Position:** The correctness of the log's location. We report *Position Accuracy (P-ACC)*, which is 1 if the predicted line number is right location, and 0 otherwise.
- **Content:** The quality of the generated log statement itself, which we break down into its constituent parts:
  - Level: The accuracy of the predicted logging level (e.g., Info, Error), reported as *Level Accuracy (L-ACC)*.
  - Variable: The correctness of the set of variables included in the logging statement, measured by their *F1-score* [10] as *V-F1*.
  - Text: The quality of the descriptive log message. We report two standard text similarity metrics: *ROUGE-L* [29] and *BLEU-1* [30].
- **Overall Quality (Judge):** To capture the holistic quality of a suggestion, we employ an LLM-as-a-Judge approach[31], [32] to compare the suggestion to the ground truth with all contextualized knowledge. We ask the *DeepSeek-R1* [33], which demonstrates its capability as a judge [34], [35], to rate the overall quality of each generated log statement on a scale of *0-1* with *5* fixed human-labeled examples. *Judge* is only measured for suggestions when *Decision* and *Position* are both right, making it an indicator of overall quality for valid suggestions. We then report the average score.

V. CLOSED-WORLD EVALUATION RESULTS

A. RQ1: How effective is LOGIMPROVER in logging quality improvement?

To answer RQ1, we evaluate LOGIMPROVER on three key logging improvement tasks: *logging insertion*, *logging modification*, and *error enhancement* as we defined in Section III-E.

Logging Statement Insertion. We first evaluate the core task of inserting new logging statements. As shown in Table II, LOGIMPROVER establishes a new state-of-the-art, significantly outperforming all baselines. Its ability to accurately answer the crucial *whether-to-log* question is demonstrated by a *D-F1* score of 0.714, a substantial 28.2% improvement over the

TABLE II: Evaluation results for the three primary logging improvement tasks.

Task	Model	Decision (D-F1)	Position (P-ACC)	Levels (L-ACC)	Variables (V-F1)	Texts		Judge
						ROUGE-L	BLEU-1	
Logging Statement Insertion	LANCE	0.192	0.285	0.549	0.357	0.244	0.183	0.196
	Raw LLM	0.485	0.521	0.682	0.632	0.335	0.305	0.657
	Raw LLM + RAG	0.548	0.579	0.739	0.702	0.418	0.426	0.694
	SCLogger	0.557	0.623	0.795	0.725	**0.585**	0.553	0.725
	LogImprover	**0.714**	**0.651**	**0.804**	**0.784**	0.572	**0.584**	**0.768**
Logging Statement Modification	Raw LLM	0.594	—	0.732	0.682	0.684	0.671	0.708
	Raw LLM + RAG	0.658	—	0.810	0.723	**0.706**	**0.675**	0.735
	LogImprover	**0.766**	—	**0.876**	**0.817**	0.687	0.664	**0.792**
Logging Variable Enhancement	Raw LLM	0.564	—	—	—	0.515	0.481	0.657
	Raw LLM + RAG	0.591	—	—	—	0.547	0.498	0.691
	LogImprover	**0.648**	—	—	—	**0.551**	**0.532**	**0.738**

strongest baseline, SCLogger. This is a direct result of our multi-stage pipeline, where Stage 1's proactive discovery and Stage 2's contextual refinement work in concert to identify valid logging improvement decision with high precision. For *where-to-log*, LogImprover achieves a *P-ACC* of 0.651, precisely localizing the log statement by leveraging the contextually identical examples from its knowledge base.

In the *what-to-log* dimension, LogImprover again shows superior performance in generating correct logging Levels (0.804 *L-ACC*) and Variables (0.784 *V-F1*). While SCLogger achieves a slightly higher *ROUGE-L* score, this metric can be misleading as it rewards simple lexical overlap. The superior *BLEU-1* score of LogImprover suggests its generated text is more precise. Most importantly, its top-ranked *Judge* score (0.768) confirms that its suggestions are perceived as holistically superior and better aligned with the expectations of professional developers.

Beyond its superior effectiveness, LogImprover also offers a significant efficiency advantage for large-scale scanning. In contrast to baselines like SCLogger, which apply a costly linting-based refinement phase to every scanned method, employs a cascading filtering strategy where each stage of the pipeline progressively narrows the set of candidates. This ensures that the most resource-intensive, LLM-based analyses are reserved for only a small set of high-potential candidates. Our measurements show that the average scanning time per method for LogImprover is only 23.76% of that required by our adapted SCLogger. This efficiency is critical, making proactive codebase analysis practical at scale.

Logging Statement Modification. This task involves deciding *whether* to modify an existing logging statement (*D-F1*) and then generating improved content. P-ACC is not applicable as the location is fixed. As shown in Table II, while all models perform reasonably well on decision-making due to the strong signal of an existing statement, LogImprover still leads with a *D-F1* of 0.766. For content, LogImprover excels in predicting Levels (0.876) and Variables (0.817). While baselines show slightly higher text similarity, this is because many ground-truth modifications do not alter the text, rewarding a passive strategy of leaving it unchanged.

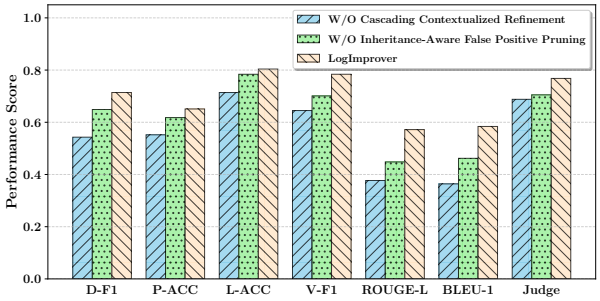


Fig. 7: The ablation study result of LogImprover.

In contrast, LogImprover often suggests refining the text for clarity and quality as part of a holistic improvement. The superior *Judge* score of 0.792 for LogImprover confirms that its more comprehensive suggestions are considered to be of higher overall quality

Logging Variable Enhancement. This task evaluates the model's ability to decide *whether* to wrap a bare logging variable (*e.g.*, error) with more context (*D-F1*) and then generate the wrapper text. Other metrics are not applicable. Table II shows that LogImprover significantly outperforms baselines in identifying enhancement opportunities, with a *D-F1* of 0.648. It also generates the most precise messages, reflected by its leading scores in *ROUGE-L*, *BLEU-1*, and most importantly, the *Judge* score (0.738), which confirms the practical value of its generated enhancements.

Answer to RQ1: LogImprover demonstrates superior performance across all evaluated logging improvement tasks: insertion, modification, and error enhancement, significantly outperforming baselines in closed-world experiments.

B. RQ2: How do the different phases affect the effectiveness of LogImprover*?*

To investigate the individual contribution of each stage in our pipeline, we conduct an ablation study by creating three variants of LogImprover, each with one key stage removed. We evaluate these variants on the logging insertion task. The first variant removes the *Proactive Candidate Identification* stage and instead treats every method as a candidate. The

second removes the *Cascading Contextualized Refinement* stage. The third removes the *Inheritance-Aware False Positive Pruning* stage and does not check for implicit logger behavior.

Proactive Candidate Identification stage serves as the fundamental scanning component of our pipeline. Its responsibility is to act as an initial, high-recall filter, sifting through the entire codebase to identify a manageable set of potential logging candidates for the subsequent stages. To quantify this impact, we conducted an efficiency-focused ablation by removing *Proactive Candidate Identification*. Our experiments confirm this: without proactive filtering, the variant must pass every scanned method through the resource-intensive RAG pipeline, increasing the average token cost per scanned method from 6,592 to 26,769, a nearly four-fold increase, confirming the value as a critical cost-saving filter.

The impact of the other stages on effectiveness is illustrated in Figure 7. Removing the contextual refinement stage causes the most severe performance degradation across all metrics, especially the holistic *D-F1* score. This confirms that generating suggestions from abstract principles alone is insufficient; grounding them in concrete, peer-validated examples is essential for quality. Furthermore, while the third variant performs better, it is still significantly outperformed by the complete LOGIMPROVER framework. This demonstrates that even with contextualized examples, failing to analyze implicit framework behavior introduces critical noise. The superior performance of the full LOGIMPROVER model clearly shows that each refinement and pruning stage provides a distinct and cumulative benefit to the final recommendation quality.

Answer to RQ2: The ablation study confirms that each stage is crucial. Stage 1 acts as an essential efficiency filter, while Stages 2 and 3 provide contextual refinement and structural pruning, contributing to generate high-quality, trustworthy recommendations.

VI. REAL-WORLD DEPLOYMENT EXPERIENCES

We evaluated the real-world effectiveness of LOGIMPROVER through its deployment at ByteDance, starting in April 2025. It has since been adopted for two primary use cases: incremental scanning of new merge requests and comprehensive, SRE-driven analysis of many critical legacy repositories for logging quality improvement.

At the time of writing, LOGIMPROVER has generated over 3,096 recommendations. The composition of these suggestions underscores the need for a holistic approach that moves beyond simple insertion: 38.57% were for insertion, 15.02% for modification, and 46.41% for logging variable enhancement. LOGIMPROVER achieved an overall labeled developer acceptance rate of 68.12%. This acceptance in a production environment validates the practical value of its suggestions and demonstrates the direct industrial impact of this research.

While a direct measurement of the impact on downstream direct reliability metrics, such as Mean Time to Resolution (MTTR), is a critical next step for future work, the developer acceptance rate serves as a crucial leading indicator. This level

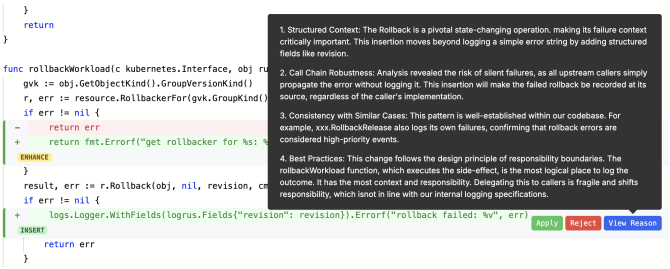


Fig. 8: A case study from a Kubernetes controller.

of adoption is the prerequisite for achieving a broader, long-term impact on the overall observability and reliability of the real-world system.

A. Case Study

To demonstrate how our methodology translates into holistic recommendations, we present a real-world case study. This example showcases how the evidence gathered from each stage is synthesized into a holistic logging patch for the `rollbackWorkload` method.

First, LOGIMPROVER identified an error path (*i.e.*, `return err`) in which a raw, non-informative error was returned. It suggested an enhancement to wrap the error with critical context (the resource's `GroupVersionKind`). The provided explanation justified this recommendation by industrial best practices and existing patterns within the codebase, making the error self-explanatory for any upstream consumer.

Second, as prominently featured in Figure 8, LOGIMPROVER identified another error path where a critical rollback operation could fail silently. Instead of a generic suggestion, the tool presents the developer with a multi-faceted argument, directly visible along with the logging patch. This argument is the direct synthesis of the evidence gathered throughout the pipeline with four reasons: (*i*) the need for *structured context* (*e.g.*, the `revision` number), the conclusion substantiated by contextualized examples retrieved; (*ii*) the perspective from call stack, which is the direct output of the call-stack logging responsibility analysis, confirming that no upstream callers were logging this error and thus ensure the current method's responsibility to record the failure; (*iii*) the pattern and silent practices learned from the similar examples, another insight derived from the RAG pipeline, which convinces the developer by showing similar rollback operations are already logged; and (*iv*) the best practice points out the logging responsibility, linking back to the principle identified in the first stage.

This case study reveals the core principle that drives LOGIMPROVER's high developer acceptance: its ability to convince developers. Rather than simply generating a patch, LOGIMPROVER constructs a compelling, evidence-based argument for why the patch is necessary. This ability to synthesize evidence from every stage makes its suggestions from simple alerts to trustworthy, expert-level recommendations.

B. Lessons Learned

The real-world deployment of LOGIMPROVER offered several critical lessons, demonstrating the challenge of managing

logging debt. We learned that logging debt is not merely a technical problem of missing logs, rooted in evolving team practices and the difficulty of maintaining context in large-scale codebases. This understanding led to two key insights.

First, abstract or outdated best practices are ineffective for addressing logging debt. Convincing a developer to modify their code requires grounding recommendations in the current, living context of the codebase. We found the most effective recommendations were those derived from the silent practices discovered within reviewed, mature repositories. This evidence-based approach of providing relevant, peer-validated examples proved far more persuasive than appealing to a generic rulebook.

Second, building a tool for logging debt requires prioritizing developer trust. Unlike other tools where a *deploy-and-iterate* feedback model might work, the tolerance for low-quality, non-functional suggestions is actually low. A high initial rate of false positives will cause developers to unsubscribe the tool, severing the feedback loop necessary for improvement. This underscores our crucial insight: *Establishing developer trust with high-precision, contextually relevant suggestions is a prerequisite for any proactive code improvement framework.*

C. Threats to Validity

The primary threat to our closed-world evaluation is that logging quality is inherently subjective. Unlike functional code, which can be verified through execution [36], logging code is evaluated against a ground truth from existing repositories. Our evaluation, in line with prior work, measures performance by comparing generated code against a ground truth extracted from existing, mature codebases. This approach is limited because the ground truth itself may not represent the optimal standard of logging practices. However, this concern is substantially mitigated by the results of our real-world deployment. The high acceptance rate of LOGIMPROVER's suggestions by developers in practice validates its effectiveness, a strong indicator that seasoned engineers find the generated suggestions valuable.

VII. RELATED WORK

Our work is situated at the intersection of two major research areas: the generation of observability instrumentation and its subsequent analysis for system reliability.

Automated Logging Statement Generation: Generally, the process of logging automation can be categorized into two stages [37], [38]: identifying logging locations and the generation of logging statements, which we summarize as *where-to-log* and *what-to-log*.

In terms of *where to log*, researchers have explored various methods [39], [8], [40], [12], [7], [41], [42] to select logging positions within the source code from different aspects and at different levels of granularity. Regarding *what-to-log*, the generation of logging statements is typically broken down into three subtasks: generating logging text [43], [9], [18], [44], selecting logging variables [45], [46], [47], and predicting the logging level [48], [49], [50], [51]. The most recent approaches based on LLM (UniLog [18], FastLog [44], SCLogger [10], LANCE [9]) provide an all-in-one solution to generate log statements, including deciding log locations, levels, content and variables in one step by leveraging knowledge of LLM. To simplify the problem, they approach this task by focusing on generating a logging statement for a given file or function, rather than scanning and assessing the entire codebase and identifing the methods need logging.

Our framework introduces a paradigm shift by solving the critical *whether-to-log* problem and introducing comprehensive tasks beyond logging statement insertion. Unlike prior work, LOGIMPROVER proactively scans entire codebases using industrial best practices and contextual knowledge retrieval. It provide an all-in-one solution for recommending multi-apsect logging quality improvement suggestions.

Log-Driven AIOps Techniques: Once logs are collected, they become the data foundation for AIOps [52], [53], [54], [55], [56], [57], [58], which aims to automate IT operations. These techniques analyze logs to perform operation tasks, such as anomaly detection, root cause analysis, and incident diagnosis.

The goal of these techniques is to learn patterns from logs. For instance, in log-based anomaly detection, pioneering unsupervised models like DeepLog [59] and LogAnomaly [60] learn the sequence of normal log events to identify deviations. Supervised approaches [61], [62] further leverage labeled data to train more accurate classifiers. To combat data scarcity, semi- supervised [63] and active learning [64] methods have also been proposed. In parallel, techniques for root cause analysis aim to construct causal graphs or trace dependencies from log event sequences to pinpoint the source of a failure [52], [58], [56]. In recent years, the emergence of LLMs has opened new avenues in this area. For example, SeaLog [4], an LLM-powered trie-based log detection agent, was proposed to take advantage of the ability to understand logs for system diagnosis.

However, a critical assumption underpins this entire body of work: the existence of high-quality log data. The performance of all these analysis techniques is fundamentally capped by the quality of logs. LOGIMPROVER is designed to fill this foundational gap. By proactively and holistically improving the quality of logging at the source, LOGIMPROVER provides the data foundation necessary for the entire ecosystem of log-driven AIOps tools.

VIII. CONCLUSION

This paper introduced LOGIMPROVER, an automated framework to address the manual, reactive and unscalable process of logging quality improvement in large-scale codebases. We demonstrated the effectiveness of two paradigm shifts: from reactive generation to proactive discovery, and from simple insertion to holistic patch generation. This was achieved through a multi-stage pipeline that combines proactive scanning with contextual knowledge retrieval and structural pruning. Our evaluation provides dual confirmation of LOGIMPROVER's performance. In controlled experiments, it significantly outperforms all baselines. Moreover, its industrial deployment

at ByteDance achieved a high developer acceptance rate of 68.12%, confirming the practical value and feasibility of using a proactive, evidence-based framework to automate the logging quality improvement lifecycle.

REFERENCES

[1] Z. Chen, Y. Kang, L. Li, X. Zhang, H. Zhang, H. Xu, Y. Zhou, L. Yang, J. Sun, Z. Xu *et al.*, "Towards intelligent incident management: why we need it and how we make it," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 1487–1497.

[2] W. Yuan, S. Lu, H. Sun, and X. Liu, "How are distributed bugs diagnosed and fixed through system logs?" *Information and Software Technology*, vol. 119, p. 106234, 2020.

[3] Z. Jiang, J. Liu, Z. Chen, Y. Li, J. Huang, Y. Huo, P. He, J. Gu, and M. R. Lyu, "Lilac: Log parsing using llms with adaptive parsing cache," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 137–160, 2024.

[4] J. Liu, J. Huang, Y. Huo, Z. Jiang, J. Gu, Z. Chen, C. Feng, M. Yan, and M. R. Lyu, "Scalable and adaptive log-based anomaly detection with expert in the loop," *arXiv preprint arXiv:2306.05032*, 2023.

[5] J. Huang, Z. Jiang, J. Liu, Y. Huo, J. Gu, Z. Chen, C. Feng, H. Dong, Z. Yang, and M. R. Lyu, "Demystifying and extracting fault-indicating information from logs for failure diagnosis," in *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2024, pp. 511–522.

[6] Z. Jiang, J. Huang, G. Yu, Z. Chen, Y. Li, R. Zhong, C. Feng, Y. Yang, Z. Yang, and M. Lyu, "L4: Diagnosing large-scale llm training failures via automated log analysis," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, 2025, pp. 51–63.

[7] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to Log: Helping Developers Make Informed Logging Decisions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*. Florence, Italy: IEEE, 2015, pp. 415–425.

[8] Z. Li, T.-H. P. Chen, and W. Shang, "Where shall we log?: Studying and suggesting logging locations in code blocks," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Virtual Event Australia: ACM, 2020, pp. 361–372.

[9] A. Mastropaolo, L. Pascarella, and G. Bavota, "Using deep learning to generate complete log statements," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 2279–2290.

[10] Y. Li, Y. Huo, R. Zhong, Z. Jiang, J. Liu, J. Huang, J. Gu, P. He, and M. R. Lyu, "Go static: Contextualized logging statement generation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 609–630, 2024.

[11] Y. Li, Y. Huo, Z. Jiang, R. Zhong, P. He, Y. Su, and M. R. Lyu, "Exploring the effectiveness of llms in automated logging generation: An empirical study," *arXiv preprint arXiv:2307.05950*, 2023.

[12] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. Shanghai China: ACM, 2017, pp. 565–581.

[13] C. Sridharan, *Distributed systems observability: a guide to building robust systems*. O'Reilly Media, 2018.

[14] N. Alshahwan, M. Harman, A. Marginean, R. Tal, and E. Wang, "Observation-based unit test generation at meta," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 173–184.

[15] Y. Li, Y. Wu, J. Liu, Z. Jiang, Z. Chen, G. Yu, and M. R. Lyu, "Coca: Generative root cause analysis for distributed systems with code knowledge," *arXiv preprint arXiv:2503.23051*, 2025.

[16] J. Huang, J. Liu, Z. Chen, Z. Jiang, Y. Li, J. Gu, C. Feng, Z. Yang, Y. Yang, and M. R. Lyu, "Faultprofit: Hierarchical fault profiling of incident tickets in large-scale cloud systems," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, 2024, pp. 392–404.

[17] O. H. Veileborg, G.-V. Saioc, and A. Møller, "Detecting blocking errors in go programs using localized abstract interpretation," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[18] J. Xu, Z. Cui, Y. Zhao, X. Zhang, S. He, P. He, L. Li, Y. Kang, Q. Lin, Y. Dang *et al.*, "Unilog: Automatic logging via llm and in-context learning," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024, pp. 1–12.

[19] G. Rong, Q. Zhang, X. Liu, and S. Gu, "A systematic review of logging practice in software engineering," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2017, pp. 534–539.

[20] S. Gu, G. Rong, H. Zhang, and H. Shen, "Logging practices in software engineering: A systematic mapping study," *IEEE transactions on software engineering*, vol. 49, no. 2, pp. 902–923, 2022.

[21] X. Liu, B. Lan, Z. Hu, Y. Liu, Z. Zhang, F. Wang, M. Q. Shieh, and W. Zhou, "Codexgraph: Bridging large language models and code repositories via code graph databases," in *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 2025, pp. 142–160.

[22] H. Tao, Y. Zhang, Z. Tang, H. Peng, X. Zhu, B. Liu, Y. Yang, Z. Zhang, Z. Xu, H. Zhang *et al.*, "Code graph model (cgm): A graph-integrated large language model for repository-level software engineering tasks," *arXiv preprint arXiv:2505.16901*, 2025.

[23] Q. Zhang, S. Chen, Y. Bei, Z. Yuan, H. Zhou, Z. Hong, J. Dong, H. Chen, Y. Chang, and X. Huang, "A survey of graph retrieval-augmented generation for customized large language models," *arXiv preprint arXiv:2501.13958*, 2025.

[24] S. Gao, X.-C. Wen, C. Gao, W. Wang, H. Zhang, and M. R. Lyu, "What makes good in-context demonstrations for code intelligence tasks with llms?" in *Proceedings of the 38th International Conference on Automated Software Engineering (ASE)*, 2023.

[25] S. Robertson, H. Zaragoza *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[26] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn, "Direct preference optimization: Your language model is secretly a reward model," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[27] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," *Facebook white paper*, vol. 5, no. 8, p. 127, 2007.

[28] B. Tan, J. Xu, Z. Zhu, and P. He, "Al-bench: A benchmark for automatic logging," *arXiv preprint arXiv:2502.03160*, 2025.

[29] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[30] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics (ACL)*, 2002, pp. 311–318.

[31] J. Gu, X. Jiang, Z. Shi, H. Tan, X. Zhai, C. Xu, W. Li, Y. Shen, S. Ma, H. Liu *et al.*, "A survey on llm-as-a-judge," *arXiv preprint arXiv:2411.15594*, 2024.

[32] Y. R. Dong, T. Hu, and N. Collier, "Can llm be a personalized judge?" *arXiv preprint arXiv:2406.11657*, 2024.

[33] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[34] N. Chen, Z. Hu, Q. Zou, J. Wu, Q. Wang, B. Hooi, and B. He, "Judgelrm: Large reasoning models as a judge," *arXiv preprint arXiv:2504.00050*, 2025.

[35] A. Xu, S. Bansal, Y. Ming, S. Yavuz, and S. Joty, "Does context matter? contextualjudgebench for evaluating llm-based judges in contextual settings," *arXiv preprint arXiv:2503.15620*, 2025.

[36] Y. Peng, J. Wan, Y. Li, and X. Ren, "Coffe: A code efficiency benchmark for code generation," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 242–265, 2025.

[37] B. Chen and Z. M. Jiang, "A survey of software log instrumentation," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–34, 2021.

[38] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu, "A survey on automated log analysis for reliability engineering," *ACM computing surveys (CSUR)*, vol. 54, no. 6, pp. 1–37, 2021.

[39] Z. Jia, S. Li, X. Liu, X. Liao, and Y. Liu, "SMARTLOG: Place error log statement by deep understanding of log intention," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Campobasso: IEEE, 2018, pp. 61–71.

[40] K. Yao, G. B. de Pádua, W. Shang, S. Sporea, A. Toma, and S. Sajedi, "Log4Perf: Suggesting Logging Locations for Web-based Systems' Performance Monitoring," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. Berlin Germany: ACM, 2018, pp. 127–138.

[41] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: Enhancing failure diagnosis with proactive logging," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 293–306.

[42] H. Li, T.-H. Chen, W. Shang, and A. E. Hassan, "Studying software logging using topic models," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2655–2694, 2018.

[43] Z. Ding, H. Li, and W. Shang, "LoGenText: Automatically Generating Logging Texts Using Neural Machine Translation," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Honolulu, HI, USA: IEEE, 2022, pp. 349–360.

[44] X. Xie, Z. Cai, S. Chen, and J. Xuan, "Fastlog: An end-to-end method to efficiently generate and insert logging statements," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 26–37.

[45] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Which Variables Should I Log?" *IEEE Transactions on Software Engineering (TSE)*, pp. 1–1, 2019.

[46] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving Software Diagnosability via Log Enhancement," *ACM Transactions on Computer Systems*, vol. 30, no. 1, pp. 1–28, 2012.

[47] S. Dai, Z. Luan, S. Huang, C. Fung, H. Wang, H. Yang, and D. Qian, "Reval: Recommend which variables to log with pretrained model and graph neural network," *IEEE Transactions on Network and Service Management*, vol. 19, no. 4, pp. 4045–4057, 2022.

[48] H. Li, W. Shang, and A. E. Hassan, "Which log level should developers choose for a new logging statement?" *Empirical Software Engineering*, vol. 22, no. 4, pp. 1684–1716, 2017.

[49] Z. Li, H. Li, T.-H. Chen, and W. Shang, "DeepLV: Suggesting Log Levels Using Ordinal Based Neural Networks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. Madrid, ES: IEEE, 2021, pp. 1461–1472.

[50] J. Liu, J. Zeng, X. Wang, K. Ji, and Z. Liang, "TeLL: Log level suggestions via modeling multi-level code block information," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. Virtual South Korea: ACM, 2022, pp. 27–38.

[51] T. Mizouchi, K. Shimari, T. Ishio, and K. Inoue, "PADLA: A Dynamic Log Level Adapter Using Online Phase Detection," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. Montreal, QC, Canada: IEEE, 2019, pp. 135–138.

[52] P. Chen, Y. Qi, P. Zheng, and D. Hou, "Causeinfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems," in *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 2014, pp. 1887–1895.

[53] A. Amar and P. C. Rigby, "Mining historical test logs to predict bugs and localize faults in the test logs," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 140–151.

[54] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 102–111.

[55] C. M. Rosenberg and L. Moonen, "Spectrum-based log diagnosis," in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–12.

[56] F. Lin, K. Muzumdar, N. P. Laptev, M.-V. Curelea, S. Lee, and S. Sankar, "Fast dimensional analysis for root cause investigation in a large-scale service environment," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 2, pp. 1–23, 2020.

[57] C. Lee, T. Yang, Z. Chen, Y. Su, and M. R. Lyu, "Eadro: An end-to-end troubleshooting framework for microservices on multi-source data," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1750–1762.

[58] G. Yu, P. Chen, Y. Li, H. Chen, X. Li, and Z. Zheng, "Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 553–565.

[59] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 1285–1298.

[60] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun *et al.*, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs." in *IJCAI*, vol. 19, no. 7, 2019, pp. 4739–4745.

[61] V.-H. Le and H. Zhang, "Log-based anomaly detection with deep learning: How far are we?" in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1356–1367.

[62] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li *et al.*, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 807–817.

[63] 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 *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1448–1460.

[64] C. Duan, T. Jia, Y. Li, and G. Huang, "Aclog: An approach to detecting anomalies from system logs with active learning," in *2023 IEEE International Conference on Web Services (ICWS)*, 2023, pp. 436–443.