LogUpdater: Automated Detection and Repair of Specific Defects in Logging Statements

RENYI ZHONG, The Chinese University of Hong Kong, Hong Kong
YICHEN LI, The Chinese University of Hong Kong, Hong Kong
JINXI KUANG, The Chinese University of Hong Kong, Hong Kong
WENWEI GU, The Chinese University of Hong Kong, Hong Kong
YINTONG HUO*, Singapore Management University, Singapore
MICHAEL R. LYU, The Chinese University of Hong Kong, Hong Kong

Developers write logging statements to monitor software runtime behaviors and system state. However, poorly constructed or misleading log messages can inadvertently obfuscate actual program execution patterns, thereby impeding effective software maintenance. Existing research on analyzing issues within logging statements is limited, primarily focusing on detecting a singular type of defect and relying on manual intervention for fixes rather than automated solutions.

To address the limitation, we initiate a systematic study that pinpoints four specific types of defects in logging statements (i.e., statement code inconsistency, static dynamic inconsistency, temporal relation inconsistency, and readability issues) through the analysis of real-world log-centric changes. We then propose LogUpdater, a two-stage framework for automatically detecting and updating logging statements for these specific defects. In the offline stage, LogUpdater constructs a similarity-based classifier on a set of synthetic defective logging statements to identify specific defect types. During the online testing phase, this classifier first evaluates logging statements in a given code snippet to determine the necessity and type of improvements required. Then, LogUpdater constructs type-aware prompts from historical logging update changes for an LLM-based recommendation framework to suggest updates addressing these specific defects.

We evaluate the effectiveness of LogUpdater on a dataset containing real-world logging changes, a synthetic dataset, and a new real-world project dataset. The results indicate that our approach is highly effective in detecting logging defects, achieving an F1 score of 0.625. Additionally, it exhibits significant improvements in suggesting precise static text and dynamic variables, with enhancements of 48.12% and 24.90%, respectively. Furthermore, LogUpdater achieves a 61.49% success rate in recommending correct updates on new real-world projects. We reported 40 problematic logging statements and their fixes to GitHub via pull requests, resulting in 25 changes confirmed and merged across 11 different projects.

CCS Concepts: • **Software and its engineering** → *Maintaining Software*.

Additional Key Words and Phrases: Logging Statement, Logging Practice, Large Language Model

*Yintong Huo is the corresponding author.

Authors' Contact Information: Renyi Zhong, renyizhong@link.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong; Yichen Li, ycli21@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong; Jinxi Kuang, jxkuang22@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong; Wenwei Gu, wwgu21@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong; Yintong Huo, ythuo@smu.edu.sg, Singapore Management University, Singapore; Michael R. Lyu, lyu@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong.

ACM 1557-7392/2025/4-ART
https://doi.org/10.1145/3731754

1 INTRODUCTION

Logging is a fundamental practice to provide visibility of systems status, especially for revealing complicated software activities in the production environments [26, 32]. Listing 1 shows a logging statement in if-block containing three ingredients: logging level (i.e., *debug*), static contents (i.e., *Left sub-range fully inconsistent*), and dynamic variables (i.e., *depth* and *right*) [29, 33].

When executed, these logging statements generate log messages that serve as primary resources for observing the behavior of systems during runtime. Such messages are instrumental in tasks like identifying anomalies [51, 81, 87] and probing into root cause analyses [11, 38]. Therefore, the accuracy and clarity of these logging statements are crucial for providing reliable system information to engineers. However, logging statements containing defects—such as inconsistencies in semantics, incorrect variable usage, or tense mismatches—can hinder their monitoring purpose and result in the dissemination of erroneous information. This, in turn, can prolong error diagnosis procedures and delay anomaly resolution [78]. While logging defects encompasses various aspects, including appropriate log levels and information verbosity, the defect mentioned in the following paper focuses specifically on factual defects related to the content and context of logging statements. Taking the above Cassandra code in Listing 1 as an example[1], the static content of this logging statement exhibits a semantic inconsistency with the dynamic variable: the static content indicates the left tree is recorded, but the dynamic variable actually represents the right tree. The corresponding produced log message will mislead engineers into looking at the wrong tree when executed.

Listing 1. A logging statement example from Cassandra.

```
static int differenceHelper(...) { ...
  else if (!lreso) {
    log.debug("({}) Left sub-range fully inconsistent {}", depth, right);
    ldiff = FULLY_INCONSISTENT;
```

Despite the prevalence of logging in modern software systems, the issues in existing logging statements receive insufficient attention due to the following challenges [22]. Firstly, the large volume of logging statements within a codebase makes manual inspection impractical [12, 84]. Secondly, because defects in logging statements do not directly affect program functionality, the functional testing suites fail to detect such issues [4]. Consequently, logging statements written by individual developers may not undergo a thorough review during the development and deployment process [93].

While most studies have developed tools to decide logging locations [5, 47, 91] and suggest logging content [16, 44, 48, 53, 63, 77], only a few have looked into identifying defective logging statements in existing software systems [4, 6, 17]. For instance, Ding et al. [17] studied the temporal relation between logging statements and the actual code sequence, proposing a detection framework for log-code temporal issues. Li et al. [45] analyzed the readability of logging statements and attempted an automated readability issue detection process to mitigate it. However, existing works suffer from two main drawbacks: **(1) Limited scope of analysis:** Previous studies focused on a single, specific category of logging statement defect (e.g., exclusively detect temporal inconsistencies [17] or only focus on readability issues [45]), failing to provide a comprehensive view of logging statement defects. While combining the outputs of these specialized tools is theoretically possible, this approach presents practical limitations. Firstly, it requires developers to manage and execute multiple disparate tools, increasing workflow complexity and potentially analysis time. Secondly, integrating results from different tools for a unified diagnosis can be cumbersome. Most importantly, such a combination does not readily lend itself to a unified automated repair mechanism. Effective automated repair often requires understanding the specific type of defect present to apply the correct fixing strategy. Building a cohesive, type-aware repair engine on top of

[1] https://issues.apache.org/jira/browse/CASSANDRA-15173

independent detectors is a significant challenge. **(2) Detection without repairing:** Existing works solely focused on detecting certain defects in logging statements without offering suggestions for repairs. Therefore, these approaches require extra human intervention with domain-specific knowledge to correct the defects. Automating the process of updating logging statements remains an unexplored area.

To tackle the first limitation, we begin with a pilot study to uncover the potential defects that can affect the accuracy and clarity of logging statements by analyzing the existing logging-related software updates. Unlike previous studies [6, 17] focused on a small number of projects (~10), our pilot study spans 641 repositories, providing a broader range to analyze defective logging practices across various real-world software development environments. Upon analysis, we identify four distinct objective defects, each requiring precise detection and *defect-specific remediation strategies*. In addition to our pilot study findings, through a survey of related work, we observe two important considerations that inform our approach: (1) while defective logging statements frequently appear in logging-related updates, they represent only a minor fraction of the overall logging statement corpus. This observation is supported by multiple empirical studies: Kabinna et al.[35] found that only 30% of logging statements undergo changes during their lifetime, while Chen et al.[7] reported that approximately 60% of logging statement changes are after-thought modifications not directly related to core feature implementation, with only about a third addressing misleading content. This *imbalanced distribution of data* highlights the need to *consider the resource overhead* of our approach, particularly when employing large language models (LLM) for fixing; and (2) recent research shows that learning-based methods can detect defects in certain categories of logging statements [4, 17]. Additionally, repairing defects with contextual defect information yields better results compared to arbitrary fixing [79]. Therefore, an effective strategy involves designing a *defect detector, followed by suggesting repair updates* based on the detection results.

Based on the above insights and to address the second limitation, we propose LogUpdater, a framework for automatically detecting and updating defective logging statements. LogUpdater contains an offline fine-tuning phase and an online phase for updating based on detection. The offline part aims to develop a defective logging statement detector. For the four distinct defect types, we first design three efficient mutation strategies and then fine-tune a similarity-based classifier for defect type identification on a set of synthetic defective logging statements. During the online phase, LogUpdater first applies the above classifier to determine whether the input code snippet needs improvement and identify the specific types of defects. Finally, LogUpdater constructs defect type-aware prompts and incorporates surrounding code contexts for suggesting logging statement updates.

We extensively evaluate LogUpdater in three datasets, including an existing log patches dataset collected from real-world software history, a synthetic dataset containing artificial defects, and a new dataset with unidentified defects for human evaluation. Our approach achieves promising detection ability (0.625 at F1) and a significant boost of 48.12% and 24.90% in static text and dynamic variables update ability, respectively. LogUpdater also demonstrates effective detection and updating capabilities on new project data, achieving an overall successful rate of 61.49%. To date, 25 out of 40 changes submitted to the GitHub project developers have been merged, underlining the practicality of LogUpdater.

In summary, this paper makes the following contributions:

- We conduct a broad-scope pilot study spanning 641 repositories, surpassing previous studies that focused on specific libraries. Our broad-scope analysis of defective logging practices across diverse development environments identify four types of common defects that developers are concerned about, providing more representative defects for real-world software.
- We propose LogUpdater, a novel end-to-end framework that automatically detects and repairs four categories of defective logging statements. Unlike existing tools limited to specific defect types, our solution addresses a wider range of defects, effectively identifying subtle and complex issues often missed by category-specific approaches.

- Experimental validation across three datasets, including successful merging of 25 PRs in popular projects demonstrates our framework's practical effectiveness. The tool's automated repair capability overcomes manual maintenance limitations in large-scale projects. To facilitate reproducibility, we open-sourced relevant code, prompts, and datasets [55].

Paper Organization Section 2 conducts a pilot study to investigate the defects in logging statements. Section 3 introduces our framework for automating detecting and updating defective logging statements. Section 4 shows the implementation details. Section 5 describes the experimental results. Section 6 presents a discussion on the practicality of LogUpdater and threats to validity. Section 7 discusses the related work. Section 8 concludes the paper.

2 PILOT STUDY

In this section, we investigate the defects in existing logging statements in real-world software applications. Considering that modifications to logging statements typically occur concurrently with functional code updates [83], we posit that isolated commit changes related to existing statements are likely aimed at addressing defect issues. Consequently, our analysis centres around examining what is referred to as the *log-centric change (i.e., LCC)*. This term describes software commits that exclusively involve the modification, excluding both addition and deletion, of one or more logging statements without accompanying modifications to functional code. Similar to the approach of related studies [6], we exclude the addition and deletion of logging statements from our analysis. This exclusion is primarily due to the unique challenges involved in modifying existing logging statements, which necessitate an analysis of semantic alignment between code and logs. In contrast, scenarios involving the addition of logging statements require distinct analytical frameworks to pinpoint potential logging locations [47], while situations involving the deletion of these statements raise separate concerns, such as redundancy [46].

2.1 Dataset Construction

The majority of current research works [8, 17, 42, 46] predominantly focus their investigations on a handful, specifically between four to six, of well-established repositories such as Hadoop and Zookeeper. This narrow focus results in a lack of an inclusive perspective on logging statements within newer and developing software systems. In contrast, our preliminary study is intended to encompass a larger number of repositories, with a broader domain, lifecycle, and scale, thereby providing a more diverse study.

2.1.1 Select target repositories. We first collect projects from popular and developing Java repositories on Github, denoted as target repositories. Specifically, the target repositories satisfy the following criteria [74]:

- The number of commits for the projects should range between 1,000 and 100,000, ensuring sufficient community interest and maintenance.
- Projects must have been created before December 31, 2019, and the last commit must be later than January 1, 2023. This criterion ensures that the projects have a significant history and sustained maintenance.
- The number of stars for the projects must exceed 1,000, indicating the popularity of the projects.

We utilized the tool provided by Dabic et al. [14] to filter GitHub repositories based on the aforementioned criteria. After excluding forks and merging repositories with the same name from different contributors, we obtained a total of 749 repositories that met the specified conditions. To ensure that each target repository contains at least one logging statement, we employed a static analysis method based on JavaParser to decompose the corresponding Java class files into units of methods. This process allowed us to eliminate repositories that did not contain any logging statements. Finally, we obtain a total of 641 repositories as the target datasets. Table 1 shows statistics of these repositories.

Table 1. Statistics of the studied repositories.

Avg #	Code line	Fork	Commit	Issue
Target repositories	454,157	1,886	7,618	1,772

2.1.2 Extracting log-centric changes. We then track changes in logging statements based on the evolution history of the target repository. In specific, we start with developing scripts to execute the git log command on the historical codebase from target repositories. These scripts extract the source code and relevant metadata for each commit, including identifiers such as the commit hash and content. Then, we employ ChangeDistiller [19] to localize modifications between consecutive versions. Furthermore, considering that logging statements typically describe the code context within a method [20], we isolate functions containing logging statements from the repository. This aligns with previous intra-function logging practices research settings [70, 83]. Next, we filter out the commits including functional code changes, reserving those where all modifications are related to logging lines. Ultimately, we identified 10,137 LCCs.

2.2 Manual Study Design

To execute a comprehensive manual analysis, we adhere to the sampling methodology outlined in prior research efforts [10, 43]. In detail, we apply a random sampling technique [67] for the selection of 500 instances of LCC, ensuring a 95% confidence level and a margin of error at 5% confidence interval [3]. We enlist two proficient co-authors, referred to as annotators A1 and A2, who each possess more than seven years of programming expertise. Given the intricate nature of this research, we administer a two-stage examination procedure. This dual-phase approach enables us to ultimately obtain 500 LCCs, each manually categorized according to their respective defect types.

Stage 1: Identifying categories of defect. In this stage, we first randomly select 100 logging statements with their surrounding code, where A1 and A2 first derive a draft list of categories of logging statements separately. Then they discuss the different cases until reaching a consensus. During this phase, the categories are revised and refined.

Stage 2: Categorizing all sampled data. In this stage, A1 and A2 independently categorize the remaining 400 logging statements by investigating the reason for changes by analyzing the data flow and structural context surrounding each logging statement. After this, their categorizations are compared. When both agree on a label, it's accepted as the final category. Before any discussion, the results yield a Cohen's Kappa of 0.78, which is a substantial level of agreement. However, when there are disagreements, the two annotators discuss collaboratively to decide on the final label for the disputed logging statements. Ultimately, they come to a mutual agreement on all categorizations.

2.3 Defects Type in Log-centric Changes

Table 2 provides an overview of the results obtained from the manual analysis. This table outlines seven distinct reasons for updating logging statements. Our research specifically concentrates on the first four categories of these reasons because they relate to factual defects. These defects result in errors in the objective description of system behaviors and have a direct influence on the accuracy and clarity of the logging statements. We have intentionally omitted subjective alterations and project-specific conventions from our focus. By doing so, we aim to concentrate on defects that have verifiable consequences on logging statements, thereby enhancing the generalizability of our taxonomy. This approach helps us to avoid merging stylistic preferences with factual inaccuracies. The choice to *add or delete statement information* relies heavily on subjective interpretations concerning what is deemed as "sufficient" information. As our methodology emphasizes objective and verifiable defects, the inclusion of

Table 2. Summarized rationals for log-centric changes.

Type	Number	Percentage
Correct statement-code inconsis.	73	14.6%
Correct static-dynamic inconsis.	45	9.0%
Correct temporal inconsis.	11	2.2%
Correct typos/capitalization	129	25.8%
Add/delete statement info	105	21.0%
Change log level	124	24.8%
Refactor output format	8	1.6%
Others	5	1.0%
Total	500	100%

highly subjective categories could jeopardize the uniformity and dependability of the processes for detecting and correcting these issues. Alterations in *log levels*, such as changing a log level from debug to info, can exhibit considerable variation between projects. These changes are often driven by organizational or project-specific logging methods. This degree of variability poses a significant challenge in defining a universal criterion for identifying what constitutes a defect in log-level settings, which is why it remains outside the scope of our main focus. Notably, we observe that 51.6% (258/500) of the changes relate to the first four factual defects, highlighting the significant attention developers devote to factual defect issues. More specifically, we detail the symptoms and explain how each defect weakens the accuracy and clarity of logging statements as follows. Figure 1 offers examples of each defect from four different repositories.

2.3.1 Statement-code inconsistency. This defect indicates that a logging statement whose semantic meaning conflicts with the execution logic within its enclosing method. The fundamental purpose of logging practice is to capture the runtime system activities of the current program for operation. A mismatch between the content of the logging statements and their code context leads to inconsistencies that can mislead engineers in failure diagnosis. For example, in Fig. 1a, the *DDL success* event was mistakenly logged as *Catalog update success*.

2.3.2 Static-dynamic inconsistency. It represents a mismatch between the static text description and the actual content of dynamic variables. The static-dynamic inconsistency is considered if either: variable's name/usage in code contradicts the static text description (e.g., logging "Counter A: " while passing counter B variable) or the variable is missing while static text requires one (e.g. logging three variables in text but give two variables). Static text within logging statements must carry the meaning and implications of the associated dynamic variables. Otherwise, the logged variable strings are meaningless and hard to comprehend for log inspectors. As in Fig. 1b, the static text *"Current mDatasetCounter:"* incorrectly describes the dynamic variable *mInodeCounter.get()*. This misalignment could lead to confusion when debugging or reviewing logs.

2.3.3 Temporal inconsistency. To address temporal inconsistency, we adhere to the definition proposed by Ding et al. [17]. This definition characterizes temporal inconsistency as arising when there is a mismatch between the logical temporal relations, which pertain to the sequence in which a logging statement is executed in relation to its associated source code (i.e., what the actual order is in the code), and the semantic temporal relations, which can be deduced from the semantic interpretation of the logging text (i.e., what the order is inferred from the generated logs). The tense used in logging statements should align with the temporal context of the events. This adherence ensures clarity for tracking system events. In Fig. 1c, the original logging statement uses the present continuous tense *"Creating"*, which implies that the action of creating the *exec probe* is still ongoing. However,

```
public class DdlThread extends BenchmarkThread {
      if (cr.getStatus() != ClientResponse.SUCCESS) {
            hardStop("DDL failed: " + cr.getStatusString());
      } else {
      -   log.info("Catalog update success #" + Long.toString(progressInd.get()) + " : " + createOrDrop[count]);
      +   log.info("DDL success #" + Long.toString(progressInd.get()) + " : " + createOrDrop[count]);
            ...
```

(a) Statement code inconsistency at DdlThread.java in voltdb.

```
public class MasterInfo {
      CommonUtils.deleteFile(Config.MASTER_LOG_FILE); }
-     LOG.info("Files recovery done. Current mDatasetCounter: " + mInodeCounter.get());
+     LOG.info("Files recovery done. Current mInodeCounter: " + mInodeCounter.get());
      ...
```

(b) Static dynamic inconsistency at MasterInfo.java in alluxio.

```
public abstract class AbstractResource implements Resource {
       .withInitialDelaySeconds(initialDelay)
      . withTimeoutSeconds(timeout) .build();
-   log.trace("Creating exec probe {}", probe);
+   log.trace("Created exec probe {}", probe);
    return probe; }
```

(c) Temporal relation inconsistency at AbstractResource.java in strimzi-kafka-operator.

```
public static QuorumVerifier parseDynamicConfig(Properties dynamicConfigProp, in
      if (numParticipators <= 2) {
            LOG.warn("No server failure will be tolerated. You need at least 3 servers."); }
      else if (numParticipators % 2 == 0) {
-           LOG.warn("Non-optimial configuration, consider an odd number of servers.");
+           LOG.warn("Non-optimal configuration, consider an odd number of servers."); }
      ...
```

(d) Readability issue at QuorumPeerConfig.java in zookeeper.

Fig. 1. Real-world examples for four defects.

the logging statement is placed immediately before the return statement, indicating that the creation of the probe has been completed by the time of the logging statement execution.

2.3.4 Readability issues. Readability issues are linguistic defects that impairing logging message clarity, which includes spelling errors and capitalization errors. Eliminating these errors is crucial to prevent misunderstandings and ensure that logs serve as clear and effective communication tools for developers and operators. Taking Fig. 1d as an example, *"Non-optimial"* is a typo error. The readability issue of logging statements is significant since many logs generated by logging statements are processed by automated tools for downstream tasks like anomaly detection and root cause analysis. While current models used in these tools can often tolerate minor typos, such defects can still degrade the performance of embeddings derived from logs, ultimately impacting the accuracy of automated analyses.

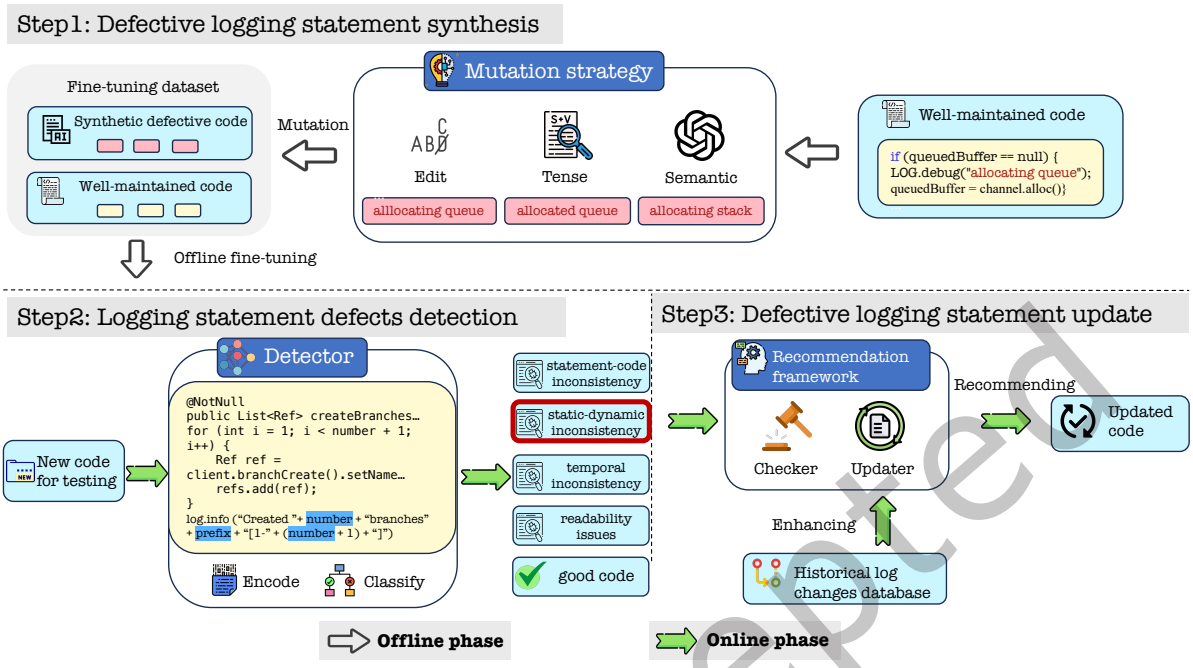

Fig. 2. The overview framework of LogUpdater.

Summary: The study uncovers four categories of factual defects (statement-code inconsistency, static-dynamic inconsistency, temporal inconsistency, and readability issues), which account for 51.6% log-centric commits.

3 FRAMEWORK

3.1 Overview

This paper resolves the defective logging statement detection and updating task, described as follows: Let s_{original} denote the given target statement. The goal is to first classify s_{original} into one of the following categories using a multi-class classification model f. This classification can be represented as: $f(s_{\text{original}}) = type_s$, where $type_s$ belongs to the set {non-defect, $d_{\text{statement\_code}}$, $d_{\text{static\_dynamic}}$, d_{temporal}, $d_{\text{readability}}$}. If $type_s \neq$ non-defect, the target statement s_{original} is updated to obtain an improved statement s_{update}.

To this end, we propose LogUpdater, a framework for automatically detecting and updating defective logging statements. Fig. 2 illustrates the high-level view of LogUpdater, which operates in two stages: offline fine-tuning and online updating. During the **offline stage**, the goal is to develop a defective logging statement detector. We first synthesize the defective logging statements with corpus collected from well-maintained repositories, ensuring high-quality code and maintenance standards. Then we use the synthetic data to fine-tune a similarity-based logging defect detector. The **online phase** involves two steps: detecting defective logging statements and updating them. We first use the fine-tuned detector to determine the $type_s$. Based on the defect information, we apply the analyzed knowledge into the LLM-based updater for logging statement update.

3.2 Defective Logging Statement Synthesis

To prepare the corpus for fine-tuning the detector, the initial step involves synthesizing negative samples. These samples are sourced from well-maintained repositories and mutated by text mutation strategies, as the corpus for tuning defective logging statement detectors. This foundational step is crucial as the training data quality employed in developing the model significantly influences its overall performance. Previous studies [4] emphasize three crucial objectives in the data generation methodology: *realism*, *diversity*, and *scalability*. *Realism* ensures that the dataset mirrors actual errors that developers might encounter. *Diversity* is vital for enabling the model to recognize various types of issues. Lastly, *scalability* is necessary since training a robust model typically demands a large dataset. In alignment with these objectives, we develop three techniques to synthesize defective logging statements for different defects, detailed as follows.

3.2.1 Mutation of words. For **readability issues**, we synthesize defective logging statements by making straightforward modifications to one word in the static constant part of the logging statement. Specifically, for simulating typos, we randomly select a word and alter its spelling based on the known typo dataset [25, 62]. If the chosen word is not in the dataset, we then randomly add, delete, or change one of its characters. For issues with capitalization abuse, we randomly select a word and convert it to uppercase. We ensure that only one type of word mutation (e.g., *executor* → *executer*) is applied to each logging statement in modification.

Listing 2. A word mutation case for readability issue.

```
public void stop() {
    for (InstanceExecutor executor : Executors){
    executor.stop();}
    LOG.info("To stop Stream executer");
    // *** executor -> executer ***
    streamExecutor.stop();
```

3.2.2 Mutation of tense. Inspired by previous work [17], for **temporal inconsistency**, we use spaCy [28], which is an open-source NLP library, to alter the tense of logging statements. Specifically, we employ dependency parsing [95] and part-of-speech (POS) tagging [13] to identify the main verbs in the logging statements. We consider the verb tagged as a head token in the dependency tree as the main verb. Finally, we change the main verbs to random tenses to mutate examples where logging statements do not align with the system events. For instance, we mutate *starting* to *started* before the sequence generator source starts in the following examples.

Listing 3. A tense mutation case for temporal inconsistency.

```
protected void doStart() throws FlumeException {
    logger.info("Sequence generator source do started");
    // *** starting -> started ***
    sourceCounter.start(); }
```

3.2.3 LLM-based mutation. For defects requiring semantic mutation (**statement-code inconsistency** and **static-dynamic inconsistency**), using rule-based or heuristic methods can be challenging. Therefore, we employ LLM-based techniques to synthesize defective logging statements. These LLMs, trained on vast amounts of code, have been shown to handle code-related downstream tasks effectively [75, 79, 88]. We use GPT-4o [65] to mutate the semantics for static content or dynamic variables. Specifically, for *statement-code inconsistency*, we prompt the LLM to carefully analyze the surrounding code and then alter the static content of logging statements to create semantic inconsistencies (e.g., recording *opened* while the actual event is *close*). For *static-dynamic inconsistency*,

we instruct the LLM to randomly mutate the static content or dynamic variables to generate potential mismatches (e.g., recording *exc* while a directory path *dir* is expected). The prompt details are shown in [60].

Listing 4. A LLM-based mutation case for statement-code inconsistency.

```
if (channelRef.get() != null) {
  channelRef.get().close();
  LOG.debug("channel {} opened", remoteAddr);
  // *** closed -> opened *** }
```

Listing 5. A LLM-based mutation case for static-dynamic inconsistency.

```
public postVisitDirectory(Path dir, Exception exc) {
  LOGGER.info("> Deleting folder {}", exc);
  // *** dir -> exc ***
  Files.delete(dir); }
```

After generating defective logging statements using above techniques, we filter out the identical samples. This step is crucial as various data mutation techniques may inadvertently generate identical data. The presence of such duplicate data could potentially skew our detector's learning process, leading to an overemphasis on homogenous data features.

3.3 Logging Statement Defects Detection

Considering that only a small portion of logging statements require update, and scanning all methods in a repository with LLM is resource-intensive, LogUpdater employs a similarity-based classifier as the detector. This detector identifies defective logging statements and determines the types of defects. The introduction of the fine-tuned detector significantly reduces costs and enables subsequent LLM components to update the logging statements in a more efficient and targeted manner. Specifically, this phase is responsible for handling a multi-class classification task. Given a logging statement $s_{original}$ with its associated context (i.e., code snippet), the goal is to determine whether $s_{original}$ has a specific type of defect. We resolve the problem through the following steps:

3.3.1 Tokenization and embedding. This step converts the input code snippet into a series of tokens. In this study, we use UniXcoder [24], which is a unified cross-modal pre-trained model for programming languages that are based on a multi-layer transformer and utilizes mask attention matrices [18] with prefix adapters to control the access to context for each token. We adapt the pre-trained tokenizer provided by the Unixcoder and embed the series of tokens into fixed-size vectors. The fine-tuning process is end-to-end where the parameters of the encoder layers are adjusted to enhance the representation vectors, allowing these vectors to better adapt to detection task.

3.3.2 Similarity-based multiple classification. After obtaining the corresponding representations for the target method and logging statement, we then devise a multi-class classifier to identify defects. Intuitively, a "good" logging statement should closely align with the semantic meaning of its adjacent code. Hence, we have developed a similarity-based classification model instead of applying a basic multi-classifier with fully connected layers. Specifically, the classifier concatenates the embedding of the logging statement l, and its corresponding code snippet s. Here are several benefits of doing so. First, explicitly identifying the target logging statements allows the model to focus more on the semantic meaning related to target logging statements. Second, it addresses the issue of multiple logging statements within a single method, as specifying l informs the model which statement to analyze. The loss function, shown in Equation 1, includes a cross-entropy loss and a similarity-based term. The first part is the standard cross-entropy loss [69], commonly applied in machine learning and optimization areas. The second part is a similarity-based loss term, which forces the target logging statement vectors to be similar

to the surrounding code snippet vectors by maximizing their cosine similarity. α is a weighting parameter to control the influence of each term. The last term is added to ensure the total loss function remains above zero.

$$loss = -(\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\log(p_{i,c}) + \alpha\frac{1}{N}\sum_{i=1}^{N}\frac{l_i \cdot s_i}{||l_i||||s_i||} - \alpha) \tag{1}$$

In the equation, C, N represent the total number of defect types and samples, respectively. The variable $y_{i,c}$ is an indicator for the ground truth label for the c-th type of the i-th sample, while $p_{i,c}$ donates the predicted probability for the same type. The vector l_i and s_i correspond to the vector representations of the target logging statement and the surrounding code snippet for the i-th sample, respectively.

3.4 Updating Detected Defective Logging Statement

Following the defect classification phase, where potential defective logging statements are identified, we proceed with a multi-expert collaboration framework to repair these statements, inspired by the cognitive synergy [23]. In particular, we employ two LLM-based experts: a *checker* and an *updater*. In Fig. 3 and Fig. 4, we see a representation of the collaborative dynamics between two domain experts. These figures illustrate not only the specialized responsibilities assigned to each expert but also the corresponding prompts tailored for them. Our approach to designing prompts for the updating component is meticulously informed by well-established best practices in LLM prompting. This involves a thoughtful task-specific structuring and the application of few-shot learning, utilizing historical examples which have demonstrated enhancements in task performance in existing literature [21, 73]. Initially, the *checker* plays a crucial role by verifying the classification outcomes generated by the detector. Subsequently, it employs LLMs to augment these results with additional semantic insights. Following this, the role of the *updater* is to diligently execute updates to logging statements in accordance with the semantic guidance provided by the *checker*.

3.4.1 Checker. The role of *checker* is to reduce the false positives brought by the similarity-based detector, where non-defective logging statements could be mistakenly marked as requiring updates due to the extremely imbalanced data in real-world situations. To address this issue, we have designed a *checker* with two objectives. The first objective is to determine whether the detector's decisions on updating logging statements are necessary, aiming to eliminate false positives. The second objective is to leverage the extensive knowledge of the LLM to offer additional semantic insights, thereby enhancing the effectiveness of subsequent experts.

Specifically, we use the information generated by the detector (i.e., target logging statement, surrounding code, defect type) as the input of the *checker*. Ultimately, the *checker* is expected to generate its judgment, rationale, and semantic information of the logging statements and the surrounding code. This information will be passed on to the next expert for reference.

3.4.2 Updater. Upon checking, we identified the target logging statements and their corresponding defect types. The next step involves employing the *updater* to modify these logging statements.

We utilize log-centric changes as supplementary knowledge to guide the LLM in learning the past update patterns from maintainers. We apply a defect type-aware strategy to select proper historical LCC examples for the *updater*'s reference. In specific, for defect types sensitive to project characteristics (e.g., temporal relation inconsistency and readability issues), we exclusively select the LCCs from the same project to maintain the consistency in the project's logging style [44]. In contrast, for defects related to semantic information inconsistency that are unaffected by logging style and inter-project variations, we incorporate LCCs from multiple projects for the *updater*. We employ the BM25 [68] ranking algorithm to identify the LCC that are the most similar to the target logging statements.

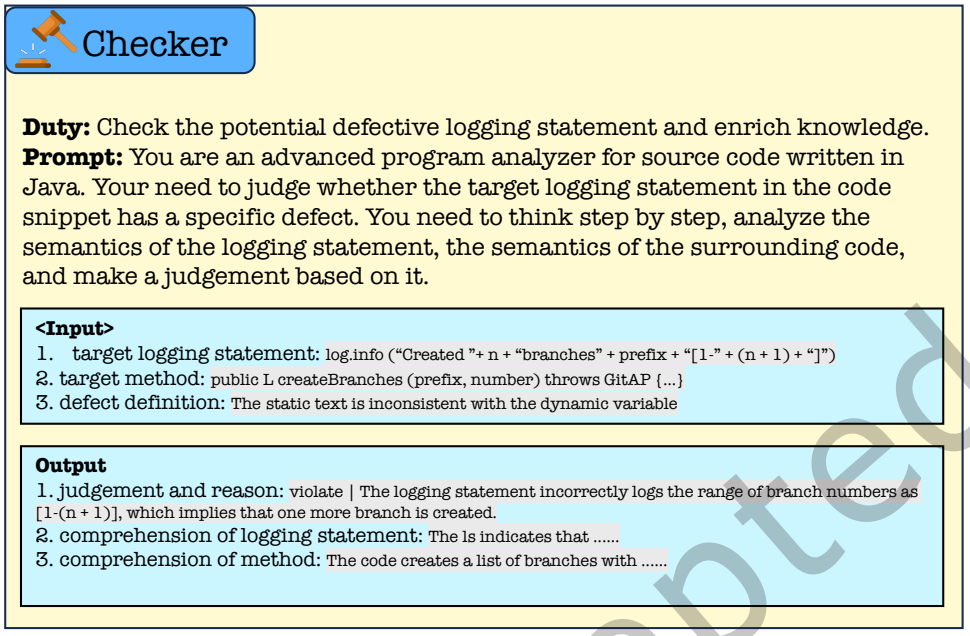

Fig. 3. The Checker framework on a static-dynamic inconsistency case.

4 IMPLEMENTATION

4.1 Fine-tuning Dataset

To develop a high-quality dataset for the detector fine-tuning process, our initial step involves the random selection of 20,000 cases that include logging statements, subsequently dividing them into four distinct categories. As a second step, the method detailed in Section 3.2 is utilized to produce 5,000 defective instances for each categorized type. Logging statements sourced from repositories that are meticulously maintained are assumed to be devoid of defects due to their compliance with rigorous community standards and comprehensive maintenance practices. Specifically, we select **well-maintained repositories** from the target repositories based on the following criteria:

- The initiators of the repository must be companies, such as those from Google, Apache, Facebook, etc. This ensures that the projects are backed by experienced teams leading to better development practices and higher-quality code.
- The repository must allow for the submission of issues; open community discussions are beneficial for developers to maintain the project.
- Projects must have a license, ensuring the availability of the code for widespread use.

We ultimately considered a total of 161 projects, encompassing 128K methods with logging statements, as well-maintained repositories. In addition to 20K defect instances, we also randomly selected 20K examples from the well-maintained repositories as positive samples, leading to 40K samples in total. We split the data into training, validation, and test sets with a ratio of 8:1:1 using stratified sampling [67]. This results in a total of 32K samples for fine-tuning the detector.

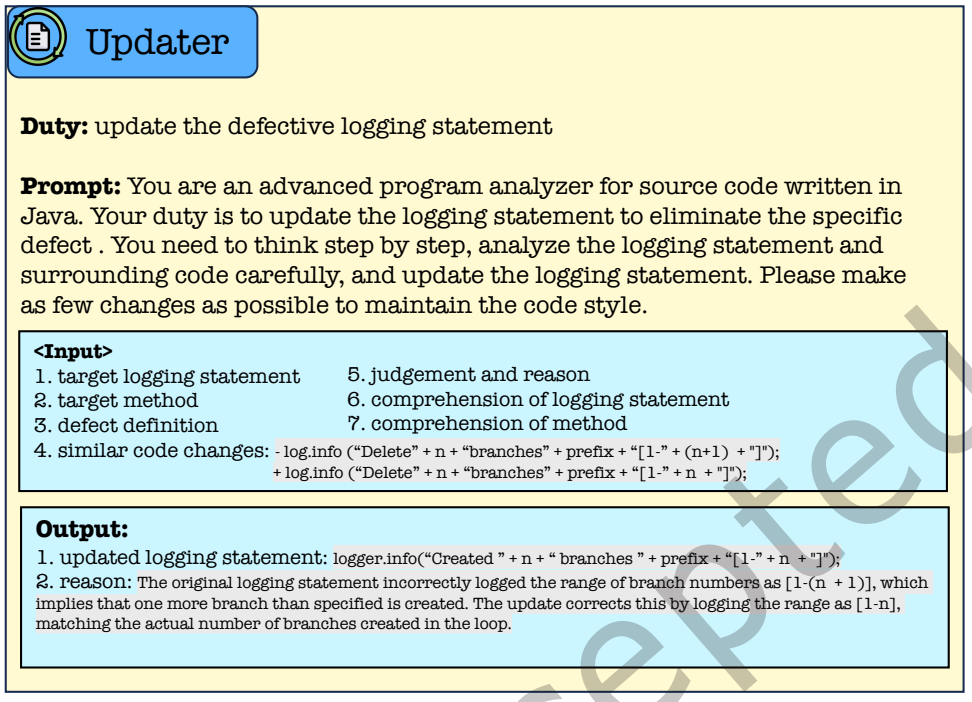

Fig. 4. The Updater framework on a static-dynamic inconsistency case.

4.2 Implementation Details

During LLM-based defective logging statement synthesis, we use the GPT-4o [65] API provided by Openai as the default LLM model. We use UniXcoder-base as the tokenization and embedding tool. We use Adam [36] as the optimizer and set the learning rate to 5e-5, Adam epsilon to 1e-8, and dropout rate to 0.1 when fine-tuning the similarity-based classification model. We fine-tune the model for 10 epochs and set the max token number to 1024. We run them on a server with Dual Intel(R) Gold 6326 CPU, 512GB physical memory, and 4x NVIDIA RTX 3090 GPU. The OS version is Ubuntu 18.04. For all experiments related to LLM inference, we set the temperature to 0 so that LLM would generate the same output for the same query to ensure reproducibility.

5 EVALUATION

We evaluate LogUpdater by answering the following research questions:

RQ1: How effective is LogUpdater at detecting and enhancing defective logging statements?
RQ2: How effective is LogUpdater at end-to-end detect and update defective logging statements?
RQ3: How do different components affect the performance of LogUpdater?
RQ4: How generalizable is LogUpdater for different LLM backbones?
RQ5: How is the practitioners' feedback on LogUpdater?

5.1 Evaluation Dataset

To evaluate the performance of LogUpdater, we conducted experiments on three distinct datasets.

5.1.1 Existing logpatches. In our study, we employ the dataset annotated during the manual investigation presented in (2.2) as a source of existing logpatches. These consist of 258 defective instances derived from the manually categorized dataset. Furthermore, we enhance our dataset by sampling an additional 258 logging statements that are defect-free from repositories known for their meticulous maintenance as outlined in Section 4.1. This procedure results in a comprehensive dataset comprising a total of 516 cases. It is important to highlight that these specific data points are deliberately excluded from the training dataset.

5.1.2 Synthetic test data. As mentioned by 4.1, we used a stratified sampled test set including 4,000 samples, as the synthetic test data.

5.1.3 New repositories. To evaluate the effectiveness of LogUpdater in addressing previously unidentified defects, we randomly sampled 100K log-code pairs from the target repositories but outside the collected well-maintained dataset for human evaluation. These samples were deliberately excluded from all previous processes, thereby guaranteeing that the evaluation was conducted on entirely fresh data.

5.2 Metrics

We evaluate LogUpdater from two perspectives: the detection ability and updating ability.

5.2.1 Evaluation metrics for detection ability. For the detection ability of LogUpdater, we use Precision, Recall, and F1-macro to measure multiple classification performance. Formally, $Precision = \frac{TP}{TP+FP}$, $Recall = \frac{TP}{TP+FN}$, $F_1 = 2 * (\frac{Precision*Recall}{Precision+Recall})$, and $F_1^{macro} = \frac{1}{N}\sum_{i=1}^{N} F_1^i$. We use F_1^{macro} to give equal weight to each class, avoiding the issue of inflating the weight of the class with more instances.

5.2.2 Evaluation metrics for updating ability. For the updating ability, we use similar metrics as logging statement generation task. Different metrics are used to evaluate different parts of the logging statements.

(1) Static content. In alignment with recent research efforts [16, 43, 61], the quality of generated logging texts is evaluated using two established metrics from the field of machine translation: BLEU [66] and ROUGE [50]. These n-gram-based metrics quantify the similarity between the generated log messages and those crafted by developers, producing a normalized score ranging from 0 to 1, where higher scores denote greater textual correspondence. Specifically, we utilize variations of these metrics, namely BLEU-K (where K = 1, 2, 4) and ROUGE-K (where K = 1, 2, L), to measure the overlap in terms of K-grams between the generated and actual log texts.

(2) Dynamic variables. Following the recent work [43], we utilize special Precision, Recall, and F1 to assess the accuracy of updated logging variables . For each logging statement, let S_{ud} represent the variables in the LogUpdater updates, and S_{gt} denote the variables in the actual logging statement. We calculate and report the proportion of correctly updated variables ($precision$=$\frac{S_{ud} \cap S_{gt}}{S_{ud}}$), the proportion of actual variables that are correctly predicted by the model ($recall$=$\frac{S_{ud} \cap S_{gt}}{S_{gt}}$), and their harmonic mean (F_1=$2 * \frac{Precision*Recall}{Precision+Recall}$).

The above metrics of updating evaluation, are widely used to assess the quality of text generation by calculating the similarity between generated log text and actual ones. However, a high similarity score in this task is not necessarily related to a significant improvement, as the defective logging statements prior to the update could already bear a close resemblance to the ground truth (i.e., the logging statements updated by developers). **Consequently, we propose a novel metric named Improvement Coefficient (IC), which focuses on the change in these metrics before and after the updates.** The formula 2 reflects the proportion of the improvement achieved after the updates relative to the maximum possible improvement. Specifically, $m \in \{BLEU, ROUGE, Precision, Recall, F_1\}$. m_{origin} and m_{update} denote the metrics before and after the updates, respectively.

$$\text{Improvement Coefficient (IC)} = \frac{m_{updated} - m_{origin}}{1 - m_{origin}} \tag{2}$$

Table 3. The detection result on existing logpatches and synthetic data

	Existing logpatches			Synthetic data		
	Precision	Recall	F1	Precision	Recall	F1
CodeT5+	0.314	0.062	0.096	0.366	0.075	0.115
Claude3.5-sonnet	0.441	0.436	0.413	0.575	0.492	0.530
Deepseek-coder-v2	0.406	0.345	0.351	0.648	0.479	0.512
Deepseek-R1	0.381	0.426	0.391	0.419	0.564	0.446
GPT3.5	0.256	0.204	0.167	0.474	0.258	0.270
GPT4o	0.468	0.401	0.410	0.512	0.467	0.476
LogUpdater	0.725	0.552	0.625	0.925	0.673	0.776

Table 4. The static texts updating results on existing logpatches and synthetic data

	BLEU-1	BLEU-2	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L
Existing logpatches						
GPT4o-Instruct	0.690 (4.90%↑)	0.587 (12.86%↑)	0.268 (0.94%↑)	0.746 (13.01%↑)	0.593 (17.44%↑)	0.745 (12.96%↑)
GPT4o-ICL	0.681 (2.14%↑)	0.588 (13.08%↑)	0.262 (0.14%↑)	0.750 (14.38%↑)	0.603 (19.47%↑)	0.748 (13.99%↑)
GPT4o-DET	0.701 (8.28%↑)	0.620 (19.83%↑)	0.327 (8.93%↑)	0.781 (25.00%↑)	0.650 (29.01%↑)	0.779 (24.57%↑)
LogUpdater	0.799 (38.34%↑)	0.721 (41.13%↑)	0.384 (16.64%↑)	0.849 (48.28%↑)	0.738 (46.85%↑)	0.848 (48.12%↑)
Synthetic data						
GPT4o-Instruct	0.546 (-7.32%↓)	0.465 (-7.86%↓)	0.316 (-2.54%↓)	0.634 (-6.08%↓)	0.499 (-9.15%↓)	0.632 (-6.35%↓)
GPT4o-ICL	0.545 (-7.56%↓)	0.468 (-7.25%↓)	0.313 (-3.00%↓)	0.639 (-4.64%↓)	0.508 (-7.19%↓)	0.637 (-4.91%↓)
GPT4o-DET	0.640 (14.89%↑)	0.579 (15.12%↑)	0.371 (5.69%↑)	0.715 (17.39%↑)	0.636 (20.69%↑)	0.714 (17.34%↑)
LogUpdater	0.659 (19.39%↑)	0.592 (17.74%↑)	0.436 (15.44%↑)	0.753 (28.40%↑)	0.648 (23.31%↑)	0.751 (28.03%↑)

5.3 RQ1: Effectiveness of LogUpdater

Approach. The effectiveness of LogUpdater should be evaluated from two aspects: first, the detection capability, specifically the capability to identify defective logging statements; second, the updating capability, which pertains to the similarity between the revised logging statements (i.e., prediction) and the actual ground truth (i.e., repository update history).

To evaluate detection capabilities, we employ a **CodeT5+** [72] model pre-trained following the CMI-Finder [4] methodology as a baseline for comparison. This model has demonstrated effective performance in binary classification tasks concerning message-condition inconsistency. We adhered to the same pretraining methodology as our proposed detector, utilizing an identical fine-tuning dataset consisting of 32K samples, conducted fine-tuning over 10 epochs, and capped the maximum token length at 1024. Furthermore, given the progress in the development of LLM and their potential use in similar tasks, we also consider serval prominent LLMs as baselines, including gerneral-purpose LLMs (Claude3.5-sonnet, GPT3.5, GPT4o), code-based LLMs (Deepseek-coder-v2), and reasoning LLMs (Deepseek-R1). For the LLM-based baselines, we provide five fixed examples for task demonstration. The prompt we use for the baselines can be found in [59].

For the updating ability, due to the lack of related research, we employed several different prompting methods as baselines. Specifically, we utilized instruction prompting **(Instruct)**, in-context learning (**ICL**), and prompts with detection information (**DET**). Instruction prompting involves directly instructing the LLM to update defective logging statements without providing additional context. In ICL, we retrieve examples to help the model better

Table 5. The dynamic variables updating result on existing logpatches and synthetic data

	Precision	Recall	F1
Existing logpatches			
GPT4o-Instruct	0.272 (0.13%↑)	0.256 (1.59%↑)	0.264 (1.15%↑)
GPT4o-ICL	0.300 (3.97%↑)	0.307 (21.82%↑)	0.304 (16.47%↑)
GPT4o-DET	0.315 (6.03%↑)	0.296 (17.46%↑)	0.305 (16.85%↑)
LogUpdater	0.439 (23.04%↑)	0.451 (26.60%↑)	0.445 (24.90%↑)
Synthetic data			
GPT4o-Instruct	0.696 (14.85%↑)	0.707 (42.20%↑)	0.702 (32.57%↑)
GPT4o-ICL	0.683 (11.20%↑)	0.664 (33.73%↑)	0.673 (26.02%↑)
GPT4o-DET	0.752 (30.53%↑)	0.761 (52.85%↑)	0.756 (44.79%↑)
LogUpdater	0.795 (42.57%↑)	0.778 (56.21%↑)	0.787 (51.80%↑)

understand the objective; we randomly sampled an add-delete pair from the LCC to create these examples. For DET, we provided the LLM relevant information about the type of defect and used prompts to guide the LLM to first explain why the target logging statement contains the specific defect before updating it.

Result. Table 3 shows the detection results with different classification models. The experimental results demonstrate that LogUpdater significantly outperforms all baseline models in detecting defective logging statements across both existing logpatches and synthetic datasets. With F1 scores of 0.625 and 0.776 respectively, LogUpdater achieves substantial improvements over the next best performers Claude3.5-sonnet. This superior performance is particularly evident in precision metrics, where LogUpdater reaches 0.725 and 0.925 across the two datasets, indicating its high reliability in correctly identifying defective logging statements. General-purpose LLMs like Claude3.5-sonnet and GPT4o show moderate effectiveness (F1 scores around 0.41-0.47), while code-specialized models like Deepseek-coder-v2 perform comparably on synthetic data but lag on real-world logpatches. Furthermore, all models perform better on synthetic data than on existing logpatches, suggesting that real-world logging defects present more complex patterns than synthetically generated ones. These findings validate the effectiveness of our specialized approach for detecting logging inconsistencies in production code.

Table 4 and Table 5 illustrate the updating performance on static content and dynamic variables, respectively. Compared to the three baselines, LogUpdater demonstrates significant performance improvements across various metrics on two datasets. Specifically, for GPT4o-Instruct and GPT4o-ICL, which lack defect information, the improvement on the logpatches dataset is minimal (0.14%↑ IC on BLEU-4), and there is even a regression (3.00%↓) on synthetic data. However, GPT4o-DET, which includes defect information, shows a certain level of improvement, highlighting the importance of defect information.

Answer to RQ1: LogUpdater demonstrates strong capabilities in defect detection and updating. It outperforms all baselines in detection with an F1 score of 0.625. Regarding its update ability, it improves static text and dynamic variables by 48.12% and 24.90%, respectively.

5.4 RQ2: The end-to-end effectiveness of LogUpdater

Approach. In this RQ, our goal is to explore the efficacy of LogUpdater once it integrates the tasks of detecting and updating defective logging statement. We conduct a thorough evaluation and comparison between LogUpdater

Table 6. The end-to-end comparisons of LogUpdater with the baselines.

	Precision	Recall	F1-score	ROUGE-L	Variable-F1	CPI
GPT4o	0.401	0.403	0.373	0.905 (68.33%↑)	0.967 (29.79%↑)	0.0194
Deepseek-coder-v2	0.383	0.365	0.372	0.861 (51.90%↑)	0.979 (0.00%)	0.0022
Deepseek-R1	0.358	0.414	0.337	0.897 (62.68%↑)	0.905 (12.03%↑)	0.0044
LogUpdater	0.725	0.552	0.625	0.928 (81.29%↑)	0.977 (55.77%↑)	0.0011

and several leading LLMs: a general-purpose model (GPT4o), a code-based model (Deepseek-coder-v2), and a reasoning model (Deepseek-R1). For each of these LLM-based benchmark models, we supply five examples for ICL to facilitate task demonstration. Details of the prompt used for the end-to-end task are available in [59]. We assess these models using the existing logpatches dataset, as elaborated in Section 5.1.1, focusing on metrics such as Detection Precision, Recall, F1-Score, ROUGE-L, and Variable-F1. It should be noted that for each model, the computation of updating ability metrics is limited to cases in which a defective detection is successfully achieved. Additionally, to appraise the financial cost associated with each method, we introduce the concept of cost per input (CPI), representing the average cost of using the official API of LLM for each input that aims at the detection and revision of defective logging statements. Note that the base LLM model for LogUpdater is GPT4o.

Result. Table 6 presents the evaluation results for end-to-end comparisons of LogUpdater with the baselines. We can see the LogUpdater outperforms all baselines in terms of all metrics. LogUpdater demonstrates significant performance enhancements compared to all baseline models when evaluated using the metrics of Precision, Recall, and F1-score. Specifically, LogUpdater achieves a precision of 0.725 and recall of 0.552, resulting in an F1-score of 0.625, which significantly surpasses the baseline models (0.373, 0.372, and 0.337 for GPT4o, Deepseek-coder-v2, and Deepseek-R1, respectively). That means, for the detection performance, LogUpdater detects more defectifve logging statements with higher precision.

In terms of the updating metrics, we observe that LogUpdater demonstrates competitive performance in ROUGE-L (0.928) and Variable-F1 (0.977) metrics. Notably, LogUpdater shows substantial improvements over original logging statements as indicated by the percentage increases: 81.29% in ROUGE-L and 55.77% in Variable-F1. This suggests that LogUpdater is particularly effective at fixing the defective logging statements at both the static content aspect and dynamic variable aspect.

From an efficiency perspective, LogUpdater demonstrates remarkable cost-effectiveness compared to its base model GPT4o. With a CPI of 0.0011, LogUpdater is approximately 16.2 times more cost-efficient than the direct use of GPT4o (0.0194). This substantial cost reduction while maintaining superior performance is particularly noteworthy since both approaches utilize GPT4o as their foundation. The significant cost advantage of LogUpdater can be attributed to its specialized architecture (i.e., use small model to develop a detector), which minimizes the number of API calls required for each input cases analysis. This cost-effectiveness, combined with LogUpdater's superior detection and generation performance, makes it a more viable solution for large-scale logging statement maintenance in real-world applications.

Answer to RQ2: In summary, LogUpdater significantly outperforms baseline models in terms of all metrics by significant margins. Moreover, LogUpdater is 16.2 times more cost-efficient than GPT4o, enhancing its viability for practical applications.

5.5 RQ3: Effects of Different Components

Approach. We consider the contribution of different components to LogUpdater 's performance from two perspectives.

For detection capability, we first experiment with several different configurations on PLM to understand how hyperparameters affect our detection results. Specifically, we change the epoch in fine-tuning and the α in loss function to control the importance of traditional cross-entropy loss and cosine similarity regularization term. Then we remove the checker LLM agent and use only the PLM for detection as a comparison. This will demonstrate the necessity of using the checker.

To assess the updating capability, we begin by comparing the outcomes with a scenario in which the detection module is completely omitted, allowing the LLM to directly update the logging statements requiring enhancement. This approach will highlight the importance of including the detection module. Subsequently, we conduct an analysis to evaluate how varying the number of similar LCCs presented in the prompt influences the effectiveness of the Updater.

Result. For hyperparameters in PLM, the default is ten epochs and α is 0.5. Experimental results in figure 5 indicate that both reducing the training by 5 epochs (0.05↓ | 0.08↓) or extending it by 5 epochs (0.03↓ | 0.04↓) lead to a certain degree of performance degradation on both existing logpatches and synthetic data. Regarding the hyperparameter α, setting it to 0 (using BCE loss directly) or setting it to 2 (giving higher weight to the similarity-based term) also results in significant performance degradation. This demonstrates the effectiveness of our designed loss function.

Figure 6 illustrates the detection performance of the framework with and without the checker. We observe that using the checker does not result in a significant change in the F1 score. Additionally, the precision on both datasets is significantly improved (0.15↑ | 0.11↑), while the recall is notably reduced (0.20↓ | 0.04↓). This aligns with our design rationale for the checker, which is to filter out the massive false positives generated by the similarity-based classifier.

Figure 7 illustrates the update performance of the framework with and without the detection component. We observe a significant decrease in various evaluation metrics across both datasets when the detection component is not used. Additionally, even without the detection component, the update component still provides some degree of repair (13.99%↑ IC on ROUGE-L) on the existing logpatches. However, its performance on the synthetic dataset is worse than not performing updates at all (10.98%↓ IC on ROUGE-L).

Table 7 demonstrates the performance during updating across varying numbers of similar LCCs as outlined in the prompt. It is evident that all performance metrics show a gradual increase when the number of shots is incremented from 1 to 4, for the case of existing log patches, and up to 3 for synthetic data. This trend reaches its highest point at 4 shots, recording a peak value of 0.395 for BLEU-4, which signifies a 2.8% increase compared to the 1-shot scenario. Such findings indicate that the updater gains from a more substantial provision of examples in the prompt, thereby facilitating an improved contextual comprehension and consequently enhancing the preciseness of the logging statements. Nonetheless, beyond this optimal range, when the number of shots is further escalated to 5 and 10, a slight decrease is observed across all metrics.

Answer to RQ3: Each component of LogUpdater contributes variously to its overall performance. Removing the detection module notably degrades performance, underscoring its critical role in enhancing logging statement accuracy. 4-shot LCCs in prompt maximizes the update performance of LogUpdater.

5.6 RQ4: Generalizability of LogUpdater

Approach. We evaluate the capability of LogUpdater using code-specific (Phind-CodeLlama [30], Deepseek-coder [94]) and general (Claude3.5-sonnet [2], Llama3 [1], GPT3.5 [64], GPT4o [65]) LLMs on both synthetic data

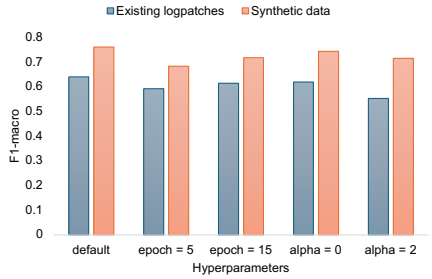

Fig. 5. Hyperparameter analysis.

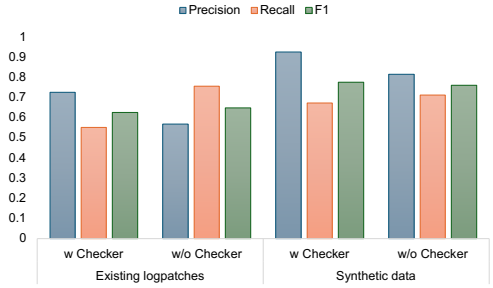

Fig. 6. Detection results w/wo checker.

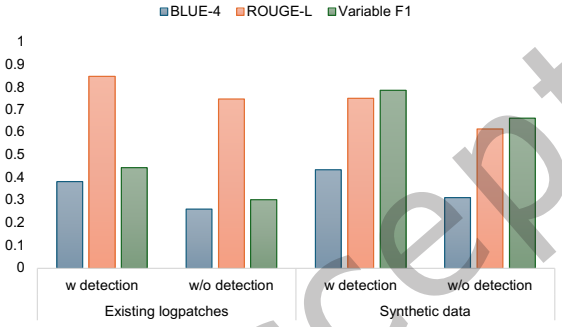

Fig. 7. Updating results w/wo detection.

Table 7. Shot Number Analysis

Shot Number	1	2	3	4	5	10
Existing logpatches						
BLEU-4	0.384	0.387	0.388	**0.395**	0.391	0.385
ROUGE-L	0.848	0.850	0.855	**0.864**	0.858	0.844
Variable-F1	0.445	0.438	0.447	**0.462**	0.449	0.434
Synthetic data						
BLEU-4	0.436	0.440	**0.446**	0.443	0.442	0.438
ROUGE-L	0.751	0.757	**0.762**	0.759	0.760	0.754
Variable-F1	0.787	0.792	0.792	**0.796**	0.789	0.784

and existing logpatches. This can illustrate the generalizability of our framework, which means that we don't rely on the ability of specific LLM backbones.

Result. Table 8 shows the performance of LogUpdater with six different backbone models. First, we observe that our framework can consistently perform well on most backbones in terms of all metrics by a large margin. On average, all models get 0.614 F1-macro for detection ability. From the perspective of update ability, models

Table 8. The performance of LogUpdater with different backbone models.

	Detection	Updating text		Updating variable
	F1-macro	BLEU-4	ROUGE-L	F1
CodeLlama-34b-v2	0.613	0.377 (15.70%↑)	**0.856 (50.85%↑)**	0.319 (7.85%↑)
Deepseek-coder-v2	**0.642**	0.371 (14.88%↑)	0.818 (37.88%↑)	0.309 (6.50%↑)
Claude3.5-sonnet	0.637	0.355 (12.72%↑)	0.810 (36.15%↑)	0.359 (13.26%↑)
Llama3-70b	0.594	0.353 (12.45%↑)	0.819 (38.23%↑)	0.359 (13.26%↑)
GPT3.5	0.573	0.311 (6.77%↑)	0.771 (21.84%↑)	0.330 (9.34%↑)
GPT4o (original)	0.625	**0.384 (16.64%↑)**	0.848 (48.12%↑)	**0.445 (24.90%↑)**

have an average of IC 13.19%↑ (reflected by BLEU-4), 38.68%↑ (reflected by ROUGE-L), and 12.52%↑ (reflected by F1) respectively.

The results demonstrate the generalizability of LogUpdater across various backbone models. We anticipate that the performance of LogUpdater can be further enhanced with advancements in LLMs.

Answer to RQ4: LogUpdater showcases great generalization ability by consistently enhancing the logging statements across various LLMs, including code-specific and general ones.

5.7 RQ5: Practitioners' feedback on LogUpdater

In this RQ, we evaluate LogUpdater based on the practitioners' feedback from two perspectives.

1) Human evaluation

Approach. We aim to assess if LogUpdater can accurately identify and update unseen defective logging statements while achieving a reasonable successful rate (SR). For this, we employ the new repositories dataset (Sec. 5.1.3), which has *no prior* identified logging statement defects and modifications. Two authors manually verified each case to confirm whether the detection is accurate and if the update is correct. If both annotators unanimously agree, we consider it a success case. Finally, we compute the success rate as the ratio of the number of successful cases to the total number of cases.

Result. LogUpdater reports 149 defective logging statements out of 100K data. The two authors achieve a Cohen's Kappa of 0.84 and finally identify 91 successful detections and updates after reaching a consensus, resulting in a 61.49% SR. This outcome demonstrates the low false positive rate of LogUpdater, enabling developers to conserve time in scrutinizing meaningless updates. Table 9 demonstrates SR for LogUpdater across different types of defects, with the highest SR of 88.57% observed for readability issues, and the lowest at 46.15% for static-dynamic inconsistency.

2) Developer's feedback

Approach. In order to comprehensively grasp the significance of rectifying identified defects from the viewpoint of developers, we conduct a randomized sampling of **40 cases** from the previously 91 instances of successfully updated logging cases. We then submit pull requests (PRs) to the repository maintainers for each sampled instance. Each pull request was accompanied by detailed information regarding the types of defects encountered, in addition to the enhanced logging statements suggested by LogUpdater. The developers' readiness and commitment to dedicate their efforts towards testing, validating, and ultimately accepting our submitted PRs underscores the effectiveness and practical value of our proposed strategy. The full merged cases are shown in our replication package [55].

Table 9. The successful rate for LogUpdater in new data.

Defect type	statement-code incon.	static-dynamic incon.	temporal incon.	readability issues
Successful rate	62.07%	46.15%	63.16%	88.57%

Result. To date, **25** out of 40 proposed PRs have been accepted and merged into the codebases, and an additional 3 cases have been recognized as bugs and are incorporated in future versions.

Answer to RQ5: In repositories containing previously unidentified defects, LogUpdater suggests 61.49% precise logging fixes according to human evaluation, illustrating its real-world practicality. While this demonstrates considerable effectiveness on unseen data, it also highlights that expert validation remains an essential part of the workflow, although for these successful cases, the effort is largely reduced to reviewing the proposed fix. Furthermore, out of 40 changes reported, developers have merged 25 cases, further validating its usefulness.

5.8 Case Study

Table 10 shows five representative real-world defective logging statements updated by LogUpdater (four successful cases and one failure case). In the first four success cases: (1) The statement-code inconsistency merged by infinispan [56] fixes the misleading as the older logging statement implies that a commit operation was undertaken, whereas the transaction actually involves rolling back. (2) In the static-dynamic inconsistency case merged by trino [57], the original statement logged variable *sortChannels* twice, mistakenly applying it to record both the channels and orders in the message. (3) For the temporal inconsistency case merged by openhab-addons [58], The old statement used the past tense ("*started*"), suggesting that the thread had already started. However, in reality, the thread initiation occurs later in the code execution. (4) In the readability issue merged by openhab-addons [54], the original one contains a typo *Intellflo*. LogUpdater successfully update it to *IntelliFlo* (a pool pump) rather than *Intelliflo* (a software platform). These real-world updates, which were confirmed by developers, indicate the practicality of LogUpdater in detecting and fixing various defective logging statements.

The last row shows a failure (false positive) case offered by LogUpdater, where it erroneously reports the logging statement as defective due to a misinterpretation of *next+1* as the current number. This error highlights LogUpdater's difficulty in handling numerical variable changes, likely due to the limited mathematical capabilities of LLMs. Despite several false positive cases, LogUpdater has significantly reduced the manual effort required to inspect all logging statements in a project, enhancing development efficiency and reliability. To eliminate these issues in the future direction, an advanced program logical comprehension module could be introduced.

6 DISCUSSION

6.1 Advantages of LogUpdater

The effectiveness of LogUpdater has been evidenced by the new project dataset, with 25 out of 40 suggested modifications being accepted and integrated into widely-used, real-world repositories. We expand upon LogUpdater in the context of actual logging practices.

6.1.1 Benefits of utilizing Checker component. The Checker is designed to mitigate false positives from the similarity-based detector, which are prevalent due to the highly imbalanced nature of real-world data (i.e., most logging statements are not defective). By comparing the effectiveness of Checker in both balanced data and unbalanced data, we argue that Checker is quite beneficial for the actual development scenario. For the balanced dataset, Figure 5 shows a noticeable improvement in precision (increases of 0.15 and 0.11) with a modest

Table 10. The real-world defective logging statement update cases suggested by LogUpdater.

Information	Update code†

Statement-code inconsistency **merged** by infinispan [56]	`public CompletionStage<Void> rollbackAsync (){`
	`- log.tracef("Transaction.commit() invoked with Xid=%s", xid);`
	`+ log.tracef("Transaction.rollback() invoked with Xid=%s", xid);`
	`status = Status.STATUS_MARKED_ROLLBACK;`

Static-dynamic inconsistency **merged** by trino [57]	`private Comparator internalCompilePageWithPositionComparator(List <Type> types, List<Integer> sortChannels, List<SortOrder> sortOrders){`
	`- log.error(t, "Error compiling comparator for channels %s with order %s", sortChannels, sortChannels);`
	`+ log.error(t, "Error compiling comparator for channels %s with order %s", sortChannels, sortOrders);`
	`comparator = new SimplePageWithPositionComparator(types, sortChannels, sortOrders, typeOperators);`

Temporal inconsistency **merged** by openhab-addons [58]	`public void run() {`
	`- logger.debug("Receiver thread started");`
	`+ logger.debug("Starting receiver thread");`
	`while (!interrupted()) {`
	`Optional<String> message = readLineBlocking();}`
	`logger.debug("Receiver thread finished");`

Readability issue **merged** by openhab-addons [54]	`public void handleCommand(ChannelUID cUID, Command cmd) {`
	`if (command instanceof RefreshType) {`
	`logger.debug("Intellflo received refresh command");`
	`logger.debug("IntelliFlo received refresh command");`
	`updateChannel(channelUID.getId(), null);}}`

A **false positive case** offered from LogUpdater.	`public Future scaleDown(Reconciliation reconciliation, String namespace, String name, int scaleTo, long timeoutMs) {`
	`Integer nextReplicas = currentScale(namespace, name);`
	`while (nextReplicas > scaleTo) {`
	`nextReplicas--;`
	`LOGGER.infoCr(reconciliation, "Scaling down from {} to {}", nextReplicas + 1, nextReplicas);`
	`resource(namespace, name).withTimeoutInMillis(timeoutMs).scale(nextReplicas);}`
	`return nextReplicas;}`

† To save space, some parts of the update code snippets are abbreviated.

reduction in recall, confirming its role in reducing false positives as intended. The value of the Checker emerges in unbalanced datasets. For example, in our new project dataset with 100K entries, the similarity-based classifier initially identified 1,771 potential defective logging statements, and only 149 of them remained after passing Checker. We further manually checked 100 randomly selected cases considered false positives by the Checker and revealed only 4 false negatives, demonstrating a 96% filtering accuracy while saving the cost of LLM invocation in the subsequent step.

6.1.2 Applicability of automatically generated logging statements. Although recent works have investigated the capabilities of LLMs in generating logging statements automatically [43, 44], LogUpdater continues to offer unique benefits from the following two aspects. First, the generated logging statements is far from mature. LLMs exhibit potential limitations, particularly in accurately generating static text intertwined with dynamic variables — a key area prone to defects [43]. For example, the logging statement generated by the SOTA solution [44] achieved a ROUGE-L score of 0.509. However, the ROUGE-L score comparing the logging statements before and after commit changes was 0.707. The performance of LLMs in automatically writing logging statements is still far from being applicable, let alone enabling them to avoid various complex defects. Our LogUpdater is designed to address these intricacies, identifying and rectifying such defects. Second, while LLM-based generation approaches are effective in generating new logging statements to some extent, they are not typically utilized to retrospectively analyze and update large amounts of existing logging statements within codebases due to computational and practical constraints. LogUpdater fills this gap by efficiently processing vast quantities of existing logging statements, ensuring they meet current standards without significant overhead.

6.1.3 Capability to solve multiple defects in one logging statement situation. During our manual analysis, we find one defective logging statement case (1/258, 0.3%) containing multiple defects—specifically, a combination of statement-code inconsistency and readability issues. This finding indicates that it is indeed quite rare for a logging statement to simultaneously exhibit multiple types of defects. Our approach is capable of addressing related issues. The detection model still provides classification results for these special cases, following which Updater performs updates based on the corresponding prompt.

6.1.4 Capability for extending new defect categories. In this study, we conducted a manual analysis of log-centric commits sampled from 641 repositories, identifying four distinct defect types, which collectively account for 51.6%—over half of the cases. It is possible that specific repositories may exhibit unique new defect types. Our proposed method is well-suited to adapt to such scenarios. By retraining the similarity-based classifier and making minor adjustments to the prompt, our approach can seamlessly extend to incorporate one or more additional defect categories.

6.1.5 Cost-management for LogUpdater's pipeline design. LogUpdater's architecture prioritizes cost-effectiveness, particularly in environments with numerous logging statements. By separating the detection and updating phases, LogUpdater efficiently filters out non-defective logging statements, minimizing the need for expensive LLM invocations. With a CPI of 0.0011, LogUpdater is approximately 16.2 times more cost-efficient than the baseline model GPT4o (CPI of 0.0194). This significant cost reduction, achieved while maintaining superior performance metrics, is primarily due to fewer API calls required for processing. In practical scenarios, the potential savings could exceed those measured in RQ2, as most logging statements do not need LLM processing. By focusing only on relevant context, LogUpdater reduces the information sent to the LLM, enhancing cost-efficiency. This makes LogUpdater a compelling solution for practitioners seeking to improve logging practices without incurring high costs.

6.2 Threats to Validity

6.2.1 Construct validity. **Fine-tuning dataset construction.** The fine-tuning dataset construction process may raise concerns. Firstly, similar to logging-related works [47, 52, 61], the basic assumption is that the logging statements in the latest programs are defect-free. Secondly, due to the required fine-tuning set scale, manually labeling a large number of negative examples would be impractical. Consequently, we employed mutation techniques to automatically synthesize defective examples, potentially leading to the inclusion of suboptimal negative examples. To mitigate this, the selection of data source is limited to well-maintained codebases (details in 4.1), which are considered as positive data. Additionally, we devised various mutation methods to ensure

the diversity, realism, and scalability of synthetic data. **Limitation in scope of defect types.** While our work focuses on four factually verifiable defect types (e.g., semantic inconsistencies), we intentionally excluded logging-level changes (24.8%) and other subjective practices. This exclusion introduces a threat to the method's comprehensiveness, as misuse logging-level issues are critical to system observability. However, our decision aligns with the prioritization of objective, context-agnostic defects to build a robust automated framework. To mitigate this threat, we plan to enhance our framework by supporting context-aware misuse log-level detection in the future work. **Repair Effectiveness Measurement.** A potential concern pertains to the practical implication of the 61.49% success rate reported in RQ5. We acknowledge that this rate, while demonstrating the tool's capability, signifies that manual intervention by domain experts is still necessary to validate the suggestions and correct the remaining 38.51% of cases where the detection or the proposed fix might be inaccurate. It is important to clarify that LogUpdater is designed to significantly reduce and shift the manual workload associated with logging statement maintenance, rather than eliminating it entirely. Compared to fully manual inspection or approaches relying solely on defect detection tools (which necessitate manual diagnosis and fixing for 100% of identified potential issues), LogUpdater offers substantial effort savings. Firstly, for the 61.49% successful cases, the developer effort is reduced from diagnosis and correction to primarily validation of the automatically generated fix. Secondly, even for the unsuccessful suggestions, the tool focuses developer attention on specific problematic statements and provides a starting point for correction. Therefore, while human oversight remains crucial, LogUpdater represents a significant step towards automating log quality improvement by handling a substantial portion of the detection and repair tasks, thereby reducing the overall manual burden.

6.2.2 Internal validity. **Potential data leakage.** We conduct experiments using various LLM backbones. Because the training data for these models are undisclosed, there is a risk of potential data leakage. To mitigate this, we initiate our data collection in April 2024, postdating the training of most LLMs. However, we recognize that for the existing logpatches dataset, there may be some risk of data leakage when testing the updating effects. To mitigate data leakage, we add synthetic test data and new repositories data. The synthetic test data are generated by the proposed synthetic strategies, which can ensure that LLMs cannot include these data as part of their training dataset. For new repositories, code snippets are current as of April 2024. The LLM updates defective logging statements without relying on April 2024 information, reducing data leakage risk. Experimental results demonstrate that LogUpdater can achieve a 61.49% success rate with new repository data, proving our method's effectiveness. **Human involvement.** We conduct manual studies to identify defects in logging statements and human evaluation of new repositories. This human-involved process may introduce bias during annotation. To minimize this threat, we engage two independent annotators with expertise in the field, and ensure that any discrepancies are resolved through discussion until a consensus is reached. Besides, we get 0.78 Cohen's Kappa for manual defect identification and 0.84 for human evaluation, which is a substantial level of agreement.

6.2.3 External validity. **The selection of programming languages.** We source data from 641 Java projects, which may impact the generalizability of our method for other languages. Following previous study [17], we choose Java because it is a mature language with multiple built-in logging and third-party logging frameworks. Nevertheless, applying LogUpdater to other programming languages may affect its performance because of distinct language features. While fine-tuning detectors on other language-specific datasets is a promising solution, we aim to incorporate more languages in future work. **The selection of LLM backbone.** We select GPT4o as our basic LLM backbone and employ six different popular instruction-taken LLMs for experiments to demonstrate the effectiveness of LogUpdater. However, different LLM models and model versions exhibit varying levels of comprehension regarding prompts, leading to fluctuations in model performance during experiments. To ensure reproducibility, we set the temperature to 0, theoretically allowing the model to return the same results for identical queries. Additionally, we plan to extend the relevant experiments to newer emerging LLMs to further assess the generalizability of our approach.

7 RELATED WORK

7.1 Studies on Maintaining High-quality Logging Statement

In numerous related studies, researchers have concentrated on identifying potential problems associated with logging statements. Li et al. [45] dedicated their research to analyzing the readability of log messages within these logging statements, formulating three dimensions pertinent to readability. They delved into the feasibility of implementing automatic classification systems for the readability of log messages. Conversely, Ding et al. [17] embarked on a pioneering investigation into the temporal relationship between logging statements and their corresponding source code. Their efforts culminated in the development of a tool designed to pinpoint and address temporal inconsistencies. In a different vein, Bouzenia et al. [4] directed their focus towards detecting discrepancies between conditions and logging statements, unveiling CMI-Finder—a model grounded in neural networks capable of identifying inconsistencies in condition-statement pairs. Zhang et al. [89] undertook a comprehensive analysis of 21 open-source projects, encompassing 70,000 logging statements, which include 48,000 production logging statements and 22,000 test logging statements. This study provided developers and researchers with substantive insights into the interplay between test and production logging. Chen et al. [6, 8] explored and pinpointed anti-patterns manifested in logging statements, incorporating six of these anti-patterns into a static code analysis tool to unearth previously undetected anti-patterns in source code. Additionally, Li et al. [42, 46] pursued an in-depth investigation into duplicate logging statements. By meticulously examining over 3,000 duplicate logging statements and their corresponding code across four expansive systems, they delineated five patterns indicative of duplicate logging smells. Furthermore, they introduced DLFinder, a tool engineered to automatically identify problematic duplicate logging code smells.

The primary distinction between our work and the aforementioned studies lies in the fact that while these studies solely detected specific categories, our approach not only identifies these issues but also provides solutions for their rectification.

7.2 Empirical Studies on Existing Logging Practices

Owing to advancements in logging within software engineering, there has been a surge in research exploring logging practices. Li et al. [40] conducted an in-depth qualitative analysis of both the benefits and limitations of logging in software development. Zhou et al. [92] delved into the ramifications of logging practices on data leaks within mobile apps. Kabinna et al. [34] identified that aspects like bug corrections, feature augmentations, and code restructuring often led to modifications in logging statements. Recently, Li et al. [43] pioneered a study on LLMs for logging statement generation and discovered that employing prompt-based strategies with zero-shot or few-shot learning enhances the generalization capabilities of LLMs. Zhao et al. [90] conducted an empirical study on the IDs in logging statements, proposed a simple approach to inject IDs in logging statements to mitigate information loss issues, and studied the information gained by injecting IDs into logs. Zeng et al. [86] and Chen [6] extended the study of Yuan et al. [83] to Android and Java systems, finding a massive presence of log statements in the analyzed systems. Li et al. [49] investigate the characteristics of dynamic variables and their importance in practice, and then explore the potential of a variable-aware log abstraction technique. Lai et al. [37] perform an analysis of logging code constructs at two levels. They answer nine research questions related to statistical and content analysis at file level and block level. Additionally, significant studies have been made in determining log levels [48, 52, 71] and content [16, 53].

7.3 Logging Statement Automation

Conventionally, the process of logging statement automation can be categorized into two stages [9, 26]: identifying logging locations and creating logging statements. We refer to these steps as where-to-log and what-to-log [44]. To address the challenge of determining where to log, researchers have explored various methods [31, 39, 47, 80,

82, 91, 93]. to identify suitable logging locations within the source code. In terms of what to log, the generation of logging statements is typically broken down into three subtasks: generating logging text [16, 61, 76, 77], selecting logging variables [15, 53, 85], and predicting the logging level [27, 41, 48, 52, 63]. The most recent approach (Unilog [77], Fastlog [76], SCLogger [44], LANCE [61]) provides automatically logging statements generation solution of deciding logging places, statements level, content, and variables at one-step by leveraging the ability of LLMs.

As we detailed discussion in Section 6.1.2, although recent works have investigated the capabilities of LLMs in generating logging statements automatically, LogUpdater continues to offer unique benefits by lower costs and better performance in detecting and updating defective logging statements.

8 CONCLUSION AND FUTURE WORK

In this paper, we find that existing research on analyzing existing logging statements is limited, primarily focusing on detecting a singular type of defect (limited scope of analysis) and relying on manual intervention for fixes rather than automated solutions (detection without repairing).

First, we conduct a broad-scope pilot study and identify four types of common defects that developers are concerned about from 641 targeted repositories. This coverage enables us to analyze defective logging practices across varied development environments, unveiling new insights into developer priorities and defect patterns that are more broadly representative of real-world software development.

Then, we propose a two-phase end-to-end framework to automatically detect and update potential defective logging statements. This is a pioneering advancement in the field, especially valuable for large-scale projects where manual repair is not feasible. By automating repairs, our tool saves significant manual effort, addressing a gap not covered by existing literature. Extensive experiment results demonstrate the effective detection and updating capabilities and practicality of LogUpdater.

In future work, we will extend LogUpdater to handle log-level optimization using context-aware techniques, addressing project-specific conventions while maintaining the framework's reliability. Specifically, a potential path is to develop a context-aware detection framework combining organizational guidelines and runtime execution frequency. By learning cross-project commit histories and logging statements, the model will learn optimal level assignments that balance observability needs with overhead constraints. The framework will adaptively detect misuse log-level based on code context, enabling project-specific yet automated existing logging statement checking.

ACKNOWLEDGEMENT

The work described in this paper was supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14206921 of the General Research Fund).

REFERENCES

[1] AI@Meta. 2024. Llama 3 Model Card. (2024). https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md
[2] Anthropic. 2024. Claude 3.5: Sonnet. https://www.anthropic.com/news/claude-3-5-sonnet Accessed: 2024-07-10.
[3] Sarah Boslaugh. 2012. *Statistics in a Nutshell.* "O'Reilly Media, Inc.".
[4] Islem Bouzenia and Michael Pradel. 2023. When to Say What: Learning to Find Condition-Message Inconsistencies. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE).* IEEE, Melbourne, Australia, 868–880. https://doi.org/10.1109/ICSE48619.2023.00081
[5] Jeanderson Cândido, Jan Haesen, Maurício Aniche, and Arie van Deursen. 2021. An Exploratory Study of Log Placement Recommendation in an Enterprise System. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR).* 143–154. https://doi.org/10.1109/MSR52588.2021.00027
[6] Boyuan Chen and Zhen Ming Jiang. 2017. Characterizing and Detecting Anti-Patterns in the Logging Code. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).* IEEE, Buenos Aires, 71–81. https://doi.org/10.1109/ICSE.2017.15

[7] Boyuan Chen and Zhen Ming Jiang. 2017. Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation. *Empirical Software Engineering* 22 (2017), 330–374.

[8] Boyuan Chen and Zhen Ming Jiang. 2019. Extracting and studying the Logging-Code-Issue-Introducing changes in Java-based large-scale open source software systems. *Empirical Software Engineering (ESE)* 24 (2019), 2285–2322.

[9] Boyuan Chen and Zhen Ming Jiang. 2021. A survey of software log instrumentation. *ACM Computing Surveys (CSUR)* 54, 4 (2021), 1–34.

[10] Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming (Jack) Jiang. 2018. An Automated Approach to Estimating Code Coverage Measures via Execution Logs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)* (Montpellier, France). Association for Computing Machinery, New York, NY, USA, 305–316. https://doi.org/10.1145/3238147.3238214

[11] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, et al. 2024. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys)*. 674–688.

[12] Zhuangbin Chen, Zhihan Jiang, Yuxin Su, Michael R Lyu, and Zibin Zheng. 2024. TraceMesh: Scalable and Streaming Sampling for Distributed Traces. *arXiv preprint arXiv:2406.06975* (2024).

[13] John S Coleman. 2005. *Introducing speech and language processing*. Cambridge university press.

[14] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *18th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. IEEE, 560–564.

[15] Shaozhi Dai, Zhongzhi Luan, Shaohan Huang, Carol Fung, He Wang, Hailong Yang, and Depei Qian. 2022. REVAL: Recommend Which Variables to Log With Pretrained Model and Graph Neural Network. *IEEE Transactions on Network and Service Management* 19, 4 (2022), 4045–4057.

[16] Zishuo Ding, Heng Li, and Weiyi Shang. 2022. LoGenText: Automatically Generating Logging Texts Using Neural Machine Translation. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Honolulu, HI, USA, 349–360. https://doi.org/10.1109/SANER53432.2022.00051

[17] Zishuo Ding, Yiming Tang, Yang Li, Heng Li, and Weiyi Shang. 2023. On the Temporal Relations between Logging and Code. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, Melbourne, Australia, 843–854. https://doi.org/10.1109/ICSE48619.2023.00079

[18] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. 2019. Unified language model pre-training for natural language understanding and generation. *Advances in neural information processing systems (NIPS)* 32 (2019).

[19] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. 2007. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering (TSE)* 33, 11 (2007), 725–743.

[20] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where Do Developers Log? An Empirical Study on Logging Practices in Industry. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, Hyderabad India, 24–33. https://doi.org/10.1145/2591062.2591175

[21] Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael Lyu. 2024. Search-based llms for code optimization. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 254–266.

[22] Sina Gholamian and Paul AS Ward. 2021. A comprehensive survey of logging in software: From logging statements automation to log mining and analysis. *arXiv preprint arXiv:2110.12489* (2021).

[23] Ben Goertzel. 2009. Cognitive synergy: A universal principle for feasible general intelligence. In *2009 8th IEEE International Conference on Cognitive Informatics*. IEEE, 464–468.

[24] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Dublin, Ireland, 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499

[25] Masato Hagiwara and Masato Mita. 2020. GitHub Typo Corpus: A Large-Scale Multilingual Dataset of Misspellings and Grammatical Errors. In *Proceedings of the Twelfth Language Resources and Evaluation Conference*. European Language Resources Association, Marseille, France, 6761–6768. https://aclanthology.org/2020.lrec-1.835

[26] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R Lyu. 2021. A survey on automated log analysis for reliability engineering. *ACM computing surveys (CSUR)* 54, 6 (2021), 1–37.

[27] Yi Wen Heng, Zeyang Ma, Zhenhao Li, Dong Jae Kim, et al. 2024. Studying and Benchmarking Large Language Models For Log Level Suggestion. *arXiv preprint arXiv:2410.08499* (2024).

[28] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. 2020. spaCy: Industrial-strength Natural Language Processing in Python. (2020). https://doi.org/10.5281/zenodo.1212303

[29] Junjie Huang, Zhihan Jiang, Zhuangbin Chen, and Michael R Lyu. 2024. ULog: Unsupervised Log Parsing with Large Language Models through Log Contrastive Units. *arXiv preprint arXiv:2406.07174* (2024).

[30] Hugging Face. 2024. Phind-CodeLlama-34B-v2. https://huggingface.co/Phind/Phind-CodeLlama-34B-v2. Accessed: 2024-07-22.

[31] Zhouyang Jia, Shanshan Li, Xiaodong Liu, Xiangke Liao, and Yunhuai Liu. 2018. SMARTLOG: Place Error Log Statement by Deep Understanding of Log Intention. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Campobasso, 61–71. https://doi.org/10.1109/SANER.2018.8330197

[32] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R Lyu. 2024. LILAC: Log Parsing using LLMs with Adaptive Parsing Cache. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 137–160.

[33] Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jiazhen Gu, Zhuangbin Chen, Jieming Zhu, and Michael R Lyu. 2024. A Large-Scale Evaluation for Log Parsing Techniques: How Far Are We?. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[34] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, Mark D Syer, and Ahmed E Hassan. 2018. Examining the stability of logging statements. *Empirical Software Engineering (ESE)* 23 (2018), 290–333.

[35] Suhas Kabinna, Weiyi Shang, Cor-Paul Bezemer, and Ahmed E. Hassan. 2016. Examining the Stability of Logging Statements. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, Suita, 326–337. https://doi.org/10.1109/SANER.2016.29

[36] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG] https://arxiv.org/abs/1412.6980

[37] Sangeeta Lal, Neetu Sardana, and Ashish Sureka. 2015. Two level empirical study of logging statements in open source java projects. *International Journal of Open Source Software and Processes (IJOSSP)* 6, 1 (2015), 49–73.

[38] Ao Li, Shan Lu, Suman Nath, Rohan Padhye, and Vyas Sekar. 2024. ExChain: Exception Dependency Analysis for Root Cause Diagnosis. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2047–2062.

[39] Heng Li, Tse-Hsun Chen, Weiyi Shang, and Ahmed E. Hassan. 2018. Studying Software Logging Using Topic Models. *Empirical Software Engineering* 23, 5 (2018), 2655–2694. https://doi.org/10.1007/s10664-018-9595-8

[40] Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E. Hassan. 2021. A Qualitative Study of the Benefits and Costs of Logging From Developers' Perspectives. *IEEE Transactions on Software Engineering (TSE)* 47, 12 (2021), 2858–2873. https://doi.org/10.1109/TSE.2020.2970422

[41] Heng Li, Weiyi Shang, and Ahmed E. Hassan. 2017. Which Log Level Should Developers Choose for a New Logging Statement? *Empirical Software Engineering* 22, 4 (2017), 1684–1716. https://doi.org/10.1007/s10664-016-9456-2

[42] Heng Li, Haoxiang Zhang, Shaowei Wang, and Ahmed E Hassan. 2021. Studying the practices of logging exception stack traces in open-source software projects. *IEEE Transactions on Software Engineering* 48, 12 (2021), 4907–4924.

[43] Yichen Li, Yintong Huo, Zhihan Jiang, Renyi Zhong, Pinjia He, Yuxin Su, and Michael R Lyu. 2023. Exploring the Effectiveness of LLMs in Automated Logging Generation: An Empirical Study. *arXiv preprint arXiv:2307.05950* (2023).

[44] Yichen Li, Yintong Huo, Renyi Zhong, Zhihan Jiang, Jinyang Liu, Junjie Huang, Jiazhen Gu, Pinjia He, and Michael R Lyu. 2024. Go static: Contextualized logging statement generation. *Proceedings of the ACM on Software Engineering (FSE)* 1, FSE (2024), 609–630.

[45] Zhenhao Li, An Ran Chen, Xing Hu, Xin Xia, Tse-Hsun Chen, and Weiyi Shang. 2023. Are They All Good? Studying Practitioners' Expectations on the Readability of Log Messages. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Luxembourg, Luxembourg, 129–140. https://doi.org/10.1109/ASE56229.2023.00136

[46] Zhenhao Li, Tse-Hsun Chen, Jinqiu Yang, and Weiyi Shang. 2019. DLFinder: Characterizing and Detecting Duplicate Logging Code Smells. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 152–163. https://doi.org/10.1109/ICSE.2019.00032

[47] Zhenhao Li, Tse-Hsun (Peter) Chen, and Weiyi Shang. 2020. 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)*. ACM, Virtual Event Australia, 361–372. https://doi.org/10.1145/3324884.3416636

[48] Zhenhao Li, Heng Li, Tse-Hsun Chen, and Weiyi Shang. 2021. DeepLV: Suggesting Log Levels Using Ordinal Based Neural Networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, Madrid, ES, 1461–1472. https://doi.org/10.1109/ICSE43902.2021.00131

[49] Zhenhao Li, Chuan Luo, Tse-Hsun Chen, Weiyi Shang, Shilin He, Qingwei Lin, and Dongmei Zhang. 2023. Did we miss something important? studying and exploring variable-aware log abstraction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 830–842.

[50] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[51] Jinyang Liu, Junjie Huang, Yintong Huo, Zhihan Jiang, Jiazhen Gu, Zhuangbin Chen, Cong Feng, Minzhi Yan, and Michael R Lyu. 2023. Scalable and adaptive log-based anomaly detection with expert in the loop. *arXiv preprint arXiv:2306.05032* (2023).

[52] Jiahao Liu, Jun Zeng, Xiang Wang, Kaihang Ji, and Zhenkai Liang. 2022. 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)*. ACM, Virtual South Korea, 27–38. https://doi.org/10.1145/3533767.3534379

[53] Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2019. Which Variables Should I Log? *IEEE Transactions on Software Engineering (TSE)* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2941943

[54] LogUpdater. 2024. readability issue case in openhab-addons. https://github.com/openhab/openhab-addons/pull/16989/commits/59872e1d1528343d810840f9819e4f56a6e539ca. Accessed: 2024-07-27.
[55] Logupdater. 2024. Replication package of Logupdater. https://anonymous.4open.science/r/Logupdater-8879/README.md. Accessed: 2024-08-01.
[56] LogUpdater. 2024. statement code inconsistency case in infinispan. https://github.com/infinispan/infinispan/pull/12676. Accessed: 2024-07-27.
[57] LogUpdater. 2024. static dynamic inconsistency case in trino. https://github.com/trinodb/trino/pull/22789. Accessed: 2024-07-27.
[58] LogUpdater. 2024. temporal inconsistency case in openhab-addons. https://github.com/openhab/openhab-addons/pull/16989/commits/0558b5f2f9df4fdd58530fe83829754e9a44a918. Accessed: 2024-07-27.
[59] Logupdater. 2025. Prompt for baseline comparison. https://anonymous.4open.science/r/Logupdater-8879/prompt/comparison_prompt.md. Accessed: 2025-02-24.
[60] Logupdater. 2025. Prompt for mutation strategies. https://anonymous.4open.science/r/Logupdater-8879/prompt/mutation_prompt.md. Accessed: 2025-02-24.
[61] Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. 2022. Using deep learning to generate complete log statements. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. 2279–2290.
[62] Microsoft Corporation. 2016. Microsoft Research Spelling-Correction Data. https://www.microsoft.com/en-us/download/details.aspx?id=52418 Accessed: 2024-06-18.
[63] Tsuyoshi Mizouchi, Kazumasa Shimari, Takashi Ishio, and Katsuro Inoue. 2019. PADLA: A Dynamic Log Level Adapter Using Online Phase Detection. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, Montreal, QC, Canada, 135–138. https://doi.org/10.1109/ICPC.2019.00029
[64] OpenAI. 2022. GPT-3.5. https://platform.openai.com/docs/models/gpt-3-5
[65] OpenAI. 2024. GPT-4o. https://openai.com/index/hello-gpt-4o/ Accessed: 2024-06-18.
[66] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics (ACL)*. 311–318.
[67] Heidar Pirzadeh, Sara Shanian, Abdelwahab Hamou-Lhadj, and Ali Mehrabian. 2011. The concept of stratified sampling of execution traces. In *2011 IEEE 19th International Conference on Program Comprehension (ICPC)*. IEEE, 225–226.
[68] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.
[69] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature* 323, 6088 (1986), 533–536.
[70] Weiyi Shang, Meiyappan Nagappan, and Ahmed E Hassan. 2015. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering (ESE)* 20 (2015), 1–27.
[71] Yiming Tang, Allan Spektor, Raffi Khatchadourian, and Mehdi Bagherzadeh. 2022. Automated evolution of feature logging statement levels using git histories and degree of interest. *Science of Computer Programming* 214 (2022), 102724.
[72] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1069–1088.
[73] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
[74] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 53–64.
[75] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.
[76] Xiaoyuan Xie, Zhipeng Cai, Songqiang Chen, and Jifeng Xuan. 2024. 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*. 26–37.
[77] Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liqun Li, Yu Kang, Qingwei Lin, Yingnong Dang, et al. 2024. UniLog: Automatic Logging via LLM and In-Context Learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*. 1–12.
[78] Lin Yang, Junjie Chen, Shutao Gao, Zhihao Gong, Hongyu Zhang, Yue Kang, and Huaan Li. 2024. Try with Simpler-An Evaluation of Improved Principal Component Analysis in Log-based Anomaly Detection. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 33, 5 (2024), 1–27.
[79] Zhen Yang, Jacky Wai Keung, Xiao Yu, Yan Xiao, Zhi Jin, and Jingyu Zhang. 2023. On the significance of category prediction for code-comment synchronization. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 32, 2 (2023), 1–41.
[80] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Steve Sporea, Andrei Toma, and Sarah Sajedi. 2018. Log4Perf: Suggesting Logging Locations for Web-based Systems' Performance Monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance*

Engineering. ACM, Berlin Germany, 127–138. https://doi.org/10.1145/3184407.3184416

[81] Boxi Yu, Jiayi Yao, Qiuai Fu, Zhiqing Zhong, Haotian Xie, Yaoliang Wu, Yuchi Ma, and Pinjia He. 2024. Deep Learning or Classical Machine Learning? An Empirical Study on Log-Based Anomaly Detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*. 1–13.

[82] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 293–306.

[83] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing Logging Practices in Open-Source Software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, Zurich, 102–112. https://doi.org/10.1109/ICSE.2012.6227202

[84] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012. Improving Software Diagnosability via Log Enhancement. *ACM Trans. Comput. Syst. (TOCS)* 30, 1, Article 4 (feb 2012), 28 pages. https://doi.org/10.1145/2110356.2110360

[85] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012. Improving Software Diagnosability via Log Enhancement. *ACM Transactions on Computer Systems* 30, 1 (2012), 1–28. https://doi.org/10.1145/2110356.2110360

[86] Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun (Peter) Chen. 2019. Studying the Characteristics of Logging Practices in Mobile Apps: A Case Study on F-Droid. *Empirical Software Engineering (ESE)* 24, 6 (2019), 3394–3434. https://doi.org/10.1007/s10664-019-09687-9

[87] Chenyangguang Zhang, Tong Jia, Guopeng Shen, Pinyan Zhu, and Ying Li. 2024. MetaLog: Generalizable Cross-System Anomaly Detection from Logs with Meta-Learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. 1–12.

[88] Dejiao Zhang, Wasi Uddin Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. 2024. Code Representation Learning At Scale. In *The Twelfth International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=vfzRRjumpX

[89] Haonan Zhang, Yiming Tang, Maxime Lamothe, Heng Li, and Weiyi Shang. 2022. Studying logging practice in test code. *Empirical Software Engineering* 27, 4 (2022), 83.

[90] Jianchen Zhao, Yiming Tang, Sneha Sunil, and Weiyi Shang. 2023. Studying and Complementing the Use of Identifiers in Logs. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 97–107.

[91] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. ACM, Shanghai China, 565–581. https://doi.org/10.1145/3132747.3132778

[92] Rui Zhou, Mohammad Hamdaqa, Haipeng Cai, and Abdelwahab Hamou-Lhadj. 2020. Mobilogleak: A preliminary study on data leakage caused by poor logging practices. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 577–581.

[93] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2015. Learning to Log: Helping Developers Make Informed Logging Decisions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*. IEEE, Florence, Italy, 415–425. https://doi.org/10.1109/ICSE.2015.60

[94] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint arXiv:2406.11931* (2024).

[95] Ran Zmigrod, Tim Vieira, and Ryan Cotterell. 2021. On Finding the K-best Non-projective Dependency Trees. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Online, 1324–1337. https://doi.org/10.18653/v1/2021.acl-long.106