arXiv:2412.18835v2 [cs.SE] 13 Aug 2025

AUCAD: Automated Construction of Alignment Dataset from Log-Related Issues for Enhancing LLM-based Log Generation

Hao Zhang
Software Institute, Nanjing University
Nanjing, Jiangsu, China
hao-zhang@smail.nju.edu.cn

Dongjun Yu
Software Institute, Nanjing University
Nanjing, Jiangsu, China
502023320014@smail.nju.edu.cn

Lei Zhang
Software Institute, Nanjing University
Nanjing, Jiangsu, China
522023320200@smail.nju.edu.cn

Guoping Rong*
Software Institute, Nanjing University
Nanjing, Jiangsu, China
ronggp@nju.edu.cn

Yongda Yu
Software Institute, Nanjing University
Nanjing, Jiangsu, China
yuyongda@smail.nju.edu.cn

Haifeng Shen
Faculty of Science and Engineering,
Southern Cross University
Bilinga, Queensland, Australia
haifeng.shen@scu.edu.au

He Zhang
Software Institute, Nanjing University
Nanjing, Jiangsu, China
hezhang@nju.edu.cn

Dong Shao
Software Institute, Nanjing University
Nanjing, Jiangsu, China
dongshao@nju.edu.cn

Hongyu Kuang
Software Institute, Nanjing University
Nanjing, Jiangsu, China
khy@nju.edu.cn

Abstract

Log statements have become an integral part of modern software systems. Prior research efforts have focused on supporting the decisions of placing log statements, such as where/what to log. With the increasing adoption of Large Language Models (LLMs) for code-related tasks such as code completion or generation, automated approaches for generating log statements have gained much momentum. However, the performance of these approaches still has a long way to go. This paper explores enhancing the performance of LLM-based solutions for automated log statement generation by post-training LLMs with a purpose-built dataset. Thus the primary contribution is a novel approach called *AUCAD*, which automatically constructs such a dataset with information extracting from log-related issues. Researchers have long noticed that a significant portion of the issues in the open-source community are related to log statements. However, distilling this portion of data requires manual efforts, which is labor-intensive and costly, rendering it impractical. Utilizing our approach, we automatically extract log-related issues from 1,537 entries of log data across 88 projects and identify 808 code snippets (i.e., methods) with retrievable source code both before and after modification of each issue (including log statements) to construct a dataset. Each entry in the dataset consists of a data pair representing high-quality and problematic log statements, respectively. With this dataset, we proceed to post-train multiple LLMs (primarily from the Llama series) for automated log statement generation. Both human and experimental evaluations indicate that these models significantly outperform existing LLM-based solutions, thereby validating the efficacy of our method for constructing a post-training dataset to enhance LLM-based log statement generation.

CCS Concepts

• **Software and its engineering → Software creation and management**.

Keywords

LLM, Log Statement Generation, Log-related Issues, Alignment

ACM Reference Format:
Hao Zhang, Dongjun Yu, Lei Zhang, Guoping Rong, Yongda Yu, Haifeng Shen, He Zhang, Dong Shao, and Hongyu Kuang. 2025. *AUCAD*: Automated Construction of Alignment Dataset from Log-Related Issues for Enhancing LLM-based Log Generation. In *the 16th International Conference on Internetware (Internetware 2025), June 20–22, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3755881.3755889

1 Introduction

Log statements serve the essential role of documenting a series of discrete events within an active service, generating log data that is essential for software engineers to understand the operational status of software systems [17]. Log data is indispensable for performing Operations and Maintenance (O&M) tasks, including anomaly detection [23, 30] and root cause localization [21, 38, 52]. The quality of log data, which depends on the quality of log statements, has a significant impact on the effectiveness of these O&M tasks [31]. Consequently, improving the quality of log statements is vital for effective O&M, and numerous studies have concentrated on enhancing log statement quality through best logging practices [15, 47], particularly in the instrumentation of log statements [5, 65].

With advancements in machine learning, various automation tools for logging practices have been proposed in both academia

*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
Internetware 2025, Trondheim, Norway
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1926-4/25/06
https://doi.org/10.1145/3755881.3755889

and industry. These tools aim to recommend specific elements in logging practices (e.g., log location [66], log level [37], and variables recorded in the logs [39]) or provide the capability to suggest complete log statements [12, 71]. However, these tools are not yet sufficiently effective for the automated generation of log statements. Current mainstream logging practices remain largely manual, heavily relying on developers' experience and personal preferences [22, 27]. Consequently, many developers struggle to accurately log the desired information, leading to inconsistencies in the quality of the resulting log data [50] and subsequent O&M tasks.

The emergence of Large Language Models (LLMs) has created promising prospects for research and applications in related fields [35]. Researchers are investigating the advanced coding capabilities of LLMs to directly recommend complete log statements through a prompt engineering approach, such as UniLog [61] and SCLogger [33]. LANCE [42] and FastLog [59] represent another technical approach – fine-tuning – commonly employed in research concerning LLM-enabled software practices [14]. This approach is particularly prevalent in studies leveraging open-source LLMs, such as the Llama series [3]. Nevertheless, post-training of LLMs, which includes both fine-tuning and alignment stages, for automated log statement generation remains an underexplored area, particularly with respect to the alignment phase. A critical challenge in this domain is the scarcity of high-quality datasets for post-training. In the case of alignment, such a dataset necessitates the inclusion of both positive and negative samples. While it is well-established that a significant portion of issues in the open-source community pertains to log statements, constructing a dataset from these issues demands extensive manual efforts, rendering it impractical.

This paper addresses this gap by proposing ***AUCAD*** (**AU**tomated **C**onstruction of **A**lignment **D**ataset), a novel method that achieves automated construction of high-quality alignment datasets specifically designed for enhancing LLM-based log statement recommendation through post-training. Our key innovation lies in establishing explicit correspondence between log statement code contexts and their evolutionary changes through systematic analysis of log-related issues in open-source communities. A pivotal component of this method is the construction of a distilled, structured dataset composed of high-quality log statements well-recognized within the open-source community. The objective of this dataset is to improve the performance of post-training LLMs in generating log statements. The proposed method collects log-related issues from the Apache open-source community, from which log statements and their corresponding code context – post-resolution in open-source software systems – are automatically extracted and structured to create a distilled alignment dataset known as *AucadLog*. This dataset is considered to be high-quality for two main reasons. First, the contextual code snippets related to the log statements have garnered significant attention from the community. Second, the final form of the log statements and contextual code snippets reflects the approval of senior developers, indicating the prominence and quality of these log statements in comparison to others. This construction enables LLMs to learn and master better logging practices from the open-source community, thereby facilitating the recommendation of high-quality log statements.

The evaluation of the dataset indicates that the dataset constructed using the *AUCAD* method ensures the relevance and effectiveness of positive examples, thereby guaranteeing both the validity and high quality of the dataset. This finding also underscores the scalability of the *AUCAD* method. Furthermore, implementation results across various alignment algorithms, including *DPO* [48], *cDPO* [44], and *SimPO* [43], demonstrate that the *AucadLog* dataset constructed through the *AUCAD* method effectively supports the alignment of LLMs. They also exhibit significant performance improvements in enhancing the capability of open-source LLMs to automatically generate log statement.

The main contributions of this paper are as follows: (1) We propose ***AUCAD* Method**, a novel method for the automated construction of an alignment dataset with both positive and negative log statements. By automatically extracting and structuring log statements with contextual code, *AUCAD* ensures high-quality datasets that enhance log generation performance and scalability. (2) We contribute ***AucadLog* Dataset**, a custom-built dataset through the *AUCAD* method that comprises both high-quality and problematic log statements as positive and negative samples for the research community to explore the alignment of LLMs.

2 Related Work

This section discusses related works on the recommendations of log statements and the adoption of LLMs for automated software engineering tasks.

2.1 The Generation/Recommendation of Log Statements

Closely related to code generation, there has recently been a surge in research of log statement instrumentation [7]. In recent years, researchers have conducted extensive empirical studies on log statements in open-source software, constructing diverse log statement datasets for systematic analysis. For instance, Hassani et al. [24] performed a manual investigation of hundreds of issues, identifying seven root causes of log-related problems. Chen et al. [6] proposed the LCC-SZZ to automatically detect Logging-Code-Issue-Introducing Changes (LCII) from software version histories, tracing the evolutionary patterns of log statements to pinpoint issue origins. While these empirical studies provide critical insights for enhancing logging practices and understanding log maintenance patterns, they predominantly focus on recording metadata of code changes (i.e., log statement modifications) without fully leveraging the rich contextual information pervasive in open-source communities. Moreover, these datasets cannot be directly applied to log statement recommendation tasks.

The advent of numerous AI applications for code generation tasks has spurred the gradual integration of AI techniques into log-related practices [16, 28, 34, 41, 42]. For a considerable period, studies in this area were confined to addressing specific sub-problems of log statement instrumentation, e.g., the position (where to log) and the content (what to log) of log statements [22]. For instance, Zhang et al. [66] proposed DeepLog, a model that recommends log statement positions using a double-branched neural network model pre-trained with semantic and syntactic features extracted from abstract syntax trees. Zhu et al. [70] introduced LogAdvisor, which

employs feature selection, classifier learning, and noise handling techniques to assist developers in determining where to place log statements.

To determine the content of log statements, Liu et al. [39] introduced a method where each program token is labeled as a vector of real numbers through RNN and self-attention layers. This approach predicts the probability of each token being logged and recommends the variables that are suitable to be logged. Ding et al. [12] proposed LoGenText, a model that recommends log texts using neural machine translation models. Gholamian et al. [20] employed log-aware clone detection to achieve automated log position and description prediction. While the verbose level determines the information captured by log statements, Liu et al. [37] put forth TeLL, a model that leverages graph neural networks to encode features, thereby facilitating the recommendation of log levels. Li et al. [34] proposed DeepLV, which utilizes recurrent neural networks to predict appropriate verbose levels.

2.2 Adoption of LLMs for Automated Software Engineering Tasks

With the rapidly increasing ability of content comprehension and generation, the emergence of LLMs is regarded as a major disruption to software development [26]. Much of the research surrounding LLMs is focused on code-related tasks, such as code generation, completion, summarization, review, and testing [67]. Take coding task as an example. There are many solutions adopting LLMs in coding, for example, Cursor [1], GitHub Copilot [2], and Alibaba's TONGYI Lingma [3] have greatly improved developers' efficiency [60]. Meanwhile, there are also many examples of applying LLMs for non-coding purposes such as code review (e.g., LLAMA-Reviewer [40]), code summarization [2, 18, 53], and software testing [55].

Despite the extensive applications of LLMs across various domains in software engineering, their utilization in log statement generation is surprisingly sparse. To the best of our knowledge, there are only a handful of studies on this specific topic, e.g., LANCE by Mastropaolo et al. [42], FastLog by Xie et al. [59], LEONID by Mastropaolo et al. [41], and UniLog by Xu et al. [61], and SCLogger by Li et al. [33], etc. LANCE [42] is a model that attempts to generate entire log statements in an end-to-end manner using the T5-small model. FastLog [59] is an end-to-end approach for log statement generation and insertion based on the fine-tuned PLBART [1] model. LEONID [41] introduces minor modifications to the LANCE model, including the use of a larger pre-training dataset and support for multiple log statements, leading to marginal improvements in log statement recommendation. The research proposed by Xu et al. [61] employs the Codex as the core of UniLog, which has a greater number of parameters. This model leverages the in-context learning capability of the LLM for warm-up training and achieves superior log statement recommendation results on the LANCE dataset. However, it is evident that the limited amount of research conducted so far has not adequately addressed the practical challenges of automatically generating high-quality log statements within source code. SCLogger[33] utilizes the principles of prompt engineering to organize the context in alignment with a Chain-of-Thought (CoT) strategy, using refinement prompts to integrate comprehensive variable type information into the preliminary log statement. In fact, a recent empirical evaluation[32] suggests that, possibly due to data leakage, the performance of most of the existing LLMs in log statement generation may be overestimated, further revealing the necessity for an in-depth research on this topic.

In summary, LLMs have been adopted in nearly all processes in software engineering [14]. As these processes often involve specialized domain knowledge and there are concerns about privacy, security, and other factors [29], fine-tuning LLMs presents a prevailing approach to applying LLMs to software engineering tasks [36]. The prominence of using LLMs for code generation-related tasks suggests their potential for the task of automated log statement generation.

3 Methodology Overview

In this section, we outline *AUCAD*, which constructs a distilled, structured dataset with log statements from log-related issues in the open-source community for the alignment of open-source LLMs.

Generally, the recommendation of log statements needs the following three sub-tasks: (1) **Position**: recommend the optimal position of log statements within the source code; (2) **Level**: recommend the appropriate verbosity level for the log statement (e.g., TRACE, DEBUG, INFO, WARN, or ERROR); (3) **Content**: generate the content/variables of the log statement to capture the required information. Therefore, we collected log statements from the resolved log-related issues in the open-source community and organized the dataset to specifically address these three sub-tasks. The dataset provides structured information to support the recommendation of optimal positions, appropriate verbosity levels, and meaningful content of log statements, effectively guiding the automated generation of comprehensive log statements. Quality of log statements is a distinctive feature of our work that distinguishes itself from similar works [42, 70], which typically collect log statements present in the source code of the open-source software without considering their evolution over the software development lifecycle. We paid particular attention to the log statements before and after the modifications in the log-related issues, which were used as negative and positive samples of logging practice, to construct a distilled dataset for the post-training of open-source LLMs.

Fig. 1 illustrates the *AUCAD* method, which consists of two primary stages: (A) **dataset building** (Section 4), and (B) **dataset evaluation** (Section 5). Each stage is described in detail in the subsequent sections.

In the first stage, we design a three-step process to collect and filter log-related issues from the Apache Issue Tracker. We extract code snippets from associated GitHub commits to build a distilled, structured dataset, called *AucadLog*, containing log statements and their associated context code after modifications in these issues, so as to build paired samples with both positive and negative examples of log statements.

In the second stage, we evaluate the dataset through (1) **human evaluation** (Section 5.2) and then (2) **experimental evaluation** (Section 5.3). For **human evaluation**, two independent researchers assess code pairs for relevance to the corresponding issue's intent,

[1] https://www.cursor.com/
[2] https://github.com/features/copilot
[3] https://tongyi.aliyun.com/lingma/

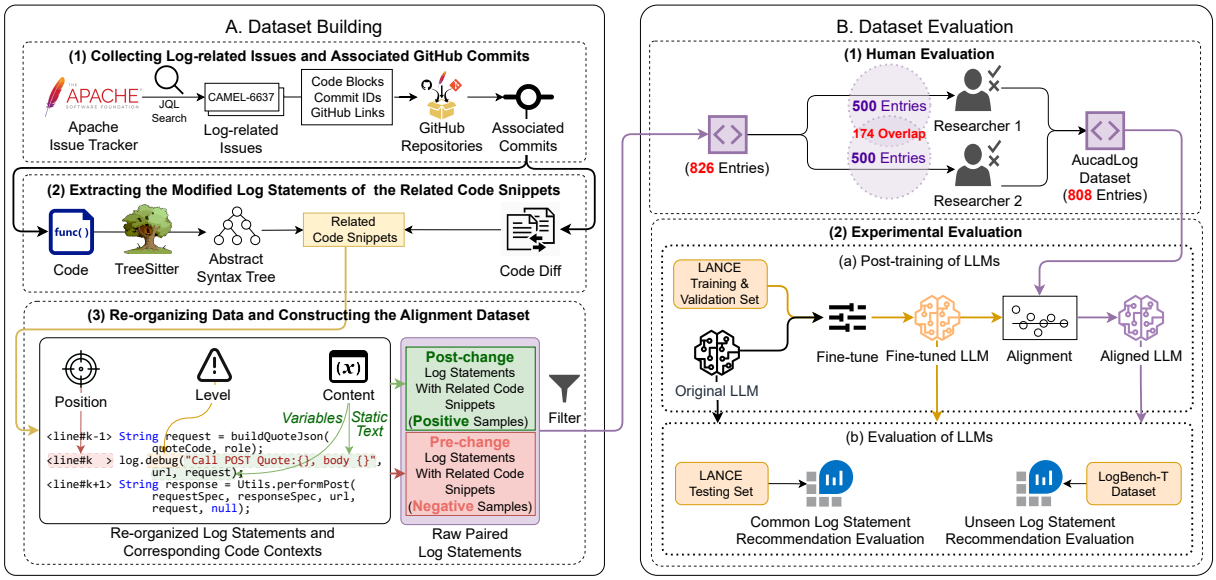

Figure 1: The *AUCAD* method for automated construction of an alignment dataset with log statements.

identifying high-quality code diffs verified by human judgment. The **experimental evaluation** defines the optimization task, followed by fine-tuning Llama 3.1 and Magicoder (leading open-source and code-specific LLMs) to develop exemplar models with enhanced log-statement generation capabilities. Subsequently, we apply preference alignment algorithms with the *AucadLog* alignment dataset to further refine the LLMs by alignment. Finally, we evaluate the original LLM, fine-tuned LLM, and aligned LLM in their ability to generate log statements for both common and unseen code contexts.

4 Dataset Building

There are a large amount of log-related issues in the open-source community, offering valuable insights for logging practices. Compared to log statements and their contexts obtained directly from the source code of the open-source software, those reviewed and discussed in the resolved issues tend to be of higher quality. Some existing studies [10] have paid attention to log-related issues and realized the significance and value embedded in this part of the data. The presence of a log-related issue often indicates suboptimal practices in the original logging approach, whereas the log statements revised through discussion are more reliable to be a good reference. Moreover, many attempts at fine-tuning LLMs have shown a significant positive effect of high-quality fine-tuning data on model performance [4, 54, 63]. Therefore, we chose to construct a high-quality distilled dataset by extracting the log statements after the modifications in the log-related issues, and the associated contextual code snippets.

As depicted in Fig. 1, we developed *AUCAD* for constructing a high-quality log statement dataset *AucadLog* from the Apache Software Foundation and GitHub. This method comprises three steps: (1) collecting a large number of log-related issues from open-source software projects of the Apache Software Foundation and retrieving their associated GitHub commit information, (2) extracting the modified log statements and the relevant code snippets as our distilled data, (3) structuring the distilled data to construct the fine-tuning dataset. Detailed steps are elaborated below.

4.1 Step 1: Collecting Log-related Issues and Associated GitHub Commits

Obviously, the more active the open-source project community means that there are more opportunities to find suitable log-related issues and a relative wealth of discussion information. With this in mind, we refer to the annual reports of the Apache Software Foundation [4] to establish the project candidate pool, and finally select 88 project teams based on the number of issues ($\geq$ 100) and participants (*watch* or *fork* or *star*$\geq$ 500).

To efficiently gather issue data from the Apache Software Foundation, we utilized the Apache Issue Tracker [5], which manages issues, tasks, and enhancement requests for Apache projects. This system, which includes both Bugzilla and Jira, enables community members to report and discuss various issues. We favored Jira for its advanced Jira Query Language (JQL), allowing precise issue filtering. We streamlined the data retrieval process by focusing on issue summaries, which concisely encapsulate the content, and employed keywords related to logging to filter relevant issues, e.g., *log*, *logger*, *print* and *logging*, etc. However, to avoid irrelevant results, we excluded phrases not associated with logging practice, typically including *log in*, *log out*, *blue print* or *print command*, etc. We targeted four issue types – *Bug*, *Dependency*, *Dependency upgrade*, and *Improvement* – to extract meaningful log patterns. Moreover, as only resolved issues were considered to ensure data quality, we used the `resolved date` field as a filter. The time range for our

[4] https://www.apache.org/foundation/docs/FY2023AnnualReport.pdf
[5] https://issues.apache.org/

data extraction was from January 1, 2002, to December 31, 2024, covering almost all the log-related issues from the Apache projects.

After collecting the issues, we further processed and analyzed the raw data to establish associations with GitHub code repositories. Specifically, we searched the issue descriptions and comments for links to GitHub or GitBox, which include commits, code snippet references, GitHub issues, and pull requests. As all Apache repositories on GitBox are available on GitHub, we utilized the GitHub REST API [6] to develop a crawler that retrieved the corresponding commit's code change information (i.e., code diff) and the modified code files. This allowed us to restore the committed code context using the diff and the changed code. For code snippet references, we obtained the complete content of the corresponding file after the commit. For related GitHub issues and pull requests, we identified the last valid commit in the pull request timeline. Through this process, we established the relationship between log-related issues and their associated GitHub commit information.

4.2 Step 2: Extracting the Modified Log Statements and the Relevant Code Snippets

Next, we extracted the modified log statements and related code snippets from the gathered information to serve as the distilled data for our dataset. We utilized tree-sitter [7] to perform static analysis on the code files obtained from GitHub and constructed Abstract Syntax Trees (ASTs) to extract the methods within the code files. Additionally, we used a combination of ASTs and regular expressions to extract all log statements present within the methods. We then analyzed the code diff to generate changed log statements before and after the modification. We automated the analysis of code diffs to extract all log statement changes associated with each issue, identifying log statements flagged by issues and subsequently modified along with their related context. Post-change log statements in the discussions on issues are treated as examples of better logging practices, i.e., positive examples. Conversely, the original, pre-change log statements – identified as problematic – are considered poor logging practices, serving as negative examples. It is noted that to ensure the quality of the constructed dataset, we extracted the complete method code containing the target log statement as the code context, including only the Java methods. In particular, the entire conditional structure was extracted if a conditional structure contained only the log statement.

4.3 Step 3: Structuring Data and Constructing the Dataset

Finally, we structured these log statements and their corresponding code contexts to build a high-quality distilled, structured dataset *AucadLog* for the post-training of LLMs.

The dataset contains the most recent versions (i.e., the committed versions) of log statements, reflecting better logging practices accepted by the open-source community following issue discussions and revisions. Additionally, it includes the original, pre-change versions of these log statements, serving as examples of poor logging practices for comparison and alignment.

[6] https://docs.github.com/en/rest
[7] https://tree-sitter.github.io/tree-sitter/

Additionally, to ensure the dataset quality, we applied automated filtering approaches based on rules and code context AST analysis to *AucadLog* Dataset for the further post-training LLMs. This process removed cases with trivial inconsistencies between expected positive and negative examples (e.g., where positive and negative examples differed only in code indentation), and mitigated potential data leakage in the testing set. In addition, considering that logging changes may occur concurrently with modifications to the functional code, we filtered out items that have code contexts closely related to the functional code.

5 Dataset Evaluation

To validate the effectiveness of the *AucadLog* dataset constructed via *AUCAD*, we conduct multi-dimensional evaluations from both human and experimental perspectives.

5.1 Research Questions

To evaluate the quality and comprehensibility of the dataset constructed using *AUCAD*, along with its potential implications for log statement recommendation, we propose the following two research questions:

- **RQ1**: To what extent can the *AUCAD* method correctly identify positive and negative log statements?
- **RQ2**: How can the *AucadLog* enhance the capabilities of LLMs in log statement recommendation through post-training?

RQ1 seeks to comprehensively validate the dataset developed through the *AUCAD* method, ensuring it correctly captures both good log statements (positve examples) and bad log statments (negative examples) as expected. Following this, **RQ2** assesses the practical utility of this dataset in the post-training of LLMs, utilizing various base models and alignment algorithms.

5.2 Human Evaluation

Based on the automatically filtered *AucadLog* dataset, two independent researchers operated separately to carefully identify if the related code diff pairs are relevant to their corresponding issues.

For the randomly shuffled 826 entries, each researcher was assigned 500 of them to ensure that every diff was examined, with 174 entries selected to be labeled twice for measuring inter-rater reliability by computing Cohen's kappa coefficient[11] to check if the two researchers mark the diffs consistently.

5.2.1 Inclusion Criteria. The criteria for determining the relevance of *AucadLog* entries to their associated issues are as follows:

(1) **Relevant entries**: Diffs are deemed as relevant entries if they align clearly with the issue's stated objectives, such as directives for log level adjustments, addition or removal of logged variables, adaptation to a new logging framework, or the adoption of parameter-based logging styles. Obviously, this relevance is a semantic level relevance that requires the evaluator to read and understand both the diff and the corresponding discussion around the issue.
(2) **Non-relevant entries**: Others.

5.2.2 Settings. To manually validate the quality of the *AucadLog* dataset, two separate researchers are participated. Each researcher

has development experience in the industries whose main development language is Java. During the evaluation, the following information is provided:

(1) **Related Issue Title**: the title of the related issue, in the format like *[FLINK-1220] Make INFO logging more verbose.*
(2) **Related Issue URL**: URL of the issue in Apache Issue Tracker.
(3) **Related Issue Description**: original issue description.
(4) **Related Issue Comments**: comments of the related issue, including the commentor's name and the content of the comment.
(5) **Origin Code**: code before modification of the referenced log issue.
(6) **Accepted fixes**: code after modification of referenced log issue.

The evaluators are supposed to read the source code both before (i.e., **Origin Code**) and after (i.e., **Accepted fixes**) modification as well as the **Related Issue Comments** to determine relevance using the *Inclusion Criteria*. They are encouraged to use other information listed above to confirm the judgment. To guarantee the consistency, the two researchers worked independently on their assigned 500 code diffs, among which 174 diffs were shared across the two researchers, which is used to calculate the Cohen's kappa coefficient.

5.3 Experimental Evaluation

In the post-training phase, which includes fine-tuning or alignment, these models are further refined to enhance their capacity to apply the acquired knowledge [68]. In this paper, we utilize the dataset constructed from discussions and modifications of log-related issues during the post-training phase of LLMs, particularly in alignment.

5.3.1 Selection of base LLMs & baselines. Given that most commercial LLMs are either closed-source or provide access exclusively through API calls, concerns over data security (particularly in enterprises) and cost considerations made open-source LLMs the preferred choice for fine-tuning and alignment. Specifically, we utilized Llama 3.1 [3] and Magicoder (based on CodeLlama) [57] as our base models. These models have undergone extensive pre-training on both natural language and code datasets. The Llama series is one of the most prominent LLMs in the open-source community. Meanwhile, Magicoder (based on CodeLlama) [57], as a specialized open-source LLM for code-related tasks. Magicoder has been extensively pre-trained on code datasets, making it particularly effective for code understanding and generation. Furthermore, to accommodate GPU resource, service availability as well as the specific requirements of recommendation tasks, we selected instruction-tuned versions of these models with relatively smaller parameter sizes. Specifically, we used *Meta-Llama-3.1-8B-Instruct* and *Magicoder-S-CL-7B* as the base models for post-training.

Since we intend to compare the alignment effects of our dataset at a similar scale, we set the following criteria to select benchmark apporoaches. (1) The selected approaches must rely on LLM technologies; (2) The selected approaches must be able to use the dataset in our study.

As the result, we identified the following approaches:

(1) LANCE [42] method utilizes a pre-trained and fine-tuned T5 model [49] for recommending log statements in Java code.
(2) FastLog [59] is an end-to-end approach for log statement generation and insertion based on the fine-tuned PLBART model [1].
(3) UniLog [61], which is based on prompt engineering using the LLM of the same parameter scale as the backbone.

It is noteworthy that although one approach SCLogger [33] also represents a recent prompt engineering-based approach, the data different structure applied by SCLogger (methods in project) and our approach (independent methods) make the comparison very limited sense.

Table 1: Datasets used by experiments.

Dataset	Purpose	Size	Data Composition
LANCE	Fine-tuning; Evaluation	131,662	Java methods
AucadLog	Alignment	808	Positive-negative example pairs from log-related issues
LogBench-T	Evaluation	6,849	Transformed unseen code

5.3.2 Datasets. The experiments mainly use three datasets (as shown in Table 1): LANCE dataset [42], which supports initial fine-tuning of the model, our *AucadLog* dataset, which provides unique pairs of positive examples of improved log statements and negative examples of poor logging practices, and LogBench-T dataset [32], which includes unseen code for most LLMs to evaluate the model. While existing empirical studies have constructed log-related datasets [6, 24], most merely catalog predefined static features of log statements (e.g., verbosity levels, message changes) without preserving their contextual code dependencies. This critical limitation renders such datasets incompatible with log statement generation tasks, thereby justifying their exclusion from our experimental framework. In the post-training phase, the goal is to enhance the capability of the LLMs to accomplish the log statement generation task. Note that since our dataset consists of only 808 entries, which, according to common practices (considering that LANCE dataset that used by [59] contains 131,662 entries), is insufficient to support effective fine-tuning on its own. Therefore, we employed the LANCE dataset for initial fine-tuning, followed by alignment using our *AucadLog* dataset. Presenting both the fine-tuning results and the outcomes after alignment allows for a comparative analysis that aids in understanding the significance of our data.

Notably, our *AucadLog* dataset **uniquely** offers both positive examples of improved log statements (after the changes) and negative examples of poor logging practices (before the changes), thereby creating natural pairs of data – *winning and losing responses* in *the same context*. In contrast to conventional log-related datasets [6, 24, 42], *AucadLog* establishes contextual associations between code structures and contrasting log statement pairs. This distinct characteristic allows us to effectively perform alignment. In this work, we leverage such data to further enhance the fine-tuned model using alignment techniques in order to train the LLM to generate higher-quality log statements.

In the evaluation, we selected the testing set from the LANCE dataset and the LogBench-T dataset [32] to assess the model's performance. In particular, the LogBench-T dataset proposed by Li et

al. [32] includes unseen code for most LLMs, addressing potential data leakage problems during the pre-training phase of open-source LLMs that could affect the final evaluation results.

5.3.3 Post-training. Fine-tuning. Fine-tuning an LLM can significantly enhance its capabilities under zero-shot conditions [56], forming the foundation for equipping the model with domain-specific expertise. During the fine-tuning phase, they are refined and guided by a smaller, task-specific dataset to enhance their performance on downstream tasks [62].

However, an LLM typically has a large number of parameters, and as such, it is resource-intensive to fully fine-tune it. To save computational and storage costs, we adopt low-parameter fine-tuning (i.e., LoRA [25]). We adopt a rank of $r = 8$ as the parameter for fine-tuning, which enables us to complete the fine-tuning procedure with merely 0.05% of the model's parameter ratio. To further reduce the training cost and improve the convergence speed, we adopt the more efficient optimizer Lion (Evo**L**ved S**i**gn M**o**me**n**tum) [8], thereby achieving model fitting in fewer training iterations while also reducing GPU memory requirements.

Alignment. Unlike fine-tuning, alignment includes both positive examples of desirable responses and negative examples of undesirable responses. This dual approach helps prevent the model from generating harmful or inappropriate outputs.

An effective approach to aligning LLMs is Reinforcement Learning from Human Feedback (RLHF) [9]. While classical RLHF has demonstrated strong performance, its multi-stage process introduces several optimization challenges, such as the need to train a reward model followed by the optimization of a policy model to maximize that reward. Recently, researchers have investigated simpler offline algorithms [19, 43, 44, 48, 64, 69]. In our study, we selected three alignment algorithms – *DPO* (Direct Preference Optimization) [48], *cDPO* (conservative *DPO*) [44], and *SimPO* (Simple Preference Optimization) [43] – to align the model, leveraging the advantages of paired data derived from code changes linked to issues in open-source community logs. Unless mentioned in the paper, all hyper-parameters adopt the default settings in the original papers[43, 44, 48].

Prompting strategy. In this work, the fine-tuning experiments use the prompting strategy of a regular *code completion scheme*, as outlined below. In this approach, the task of generating log statements is framed as a code completion task. This is achieved by constructing zero-shot prompts that instruct the model with complete code snippets.

> *Recommend the optimal log statements in the following given codes. You need to output the full code with optimal log statement inserted, and do not explain the reason.*
> *Code:*
> ```{code}```

Specifically, for LANCE and FastLog, which do not support the prompt engineering approach, we utilized the model-specific input format for evaluation (i.e., only the code part above as the model input without the rest accompanying instructional text).

5.3.4 Evaluation metrics. In our evaluation, we selected the typical metrics adopted by the previous studies [34, 42, 61] to evaluate the position, verbosity level, and message in generated log statements. Details of the metrics are elaborated below.

Level in ground truth \ Level in response	FARAL	ERROR	WARN	INFO	DEBUG	TRACE
FARAL		■	■	■	■	■
ERROR	■			■	■	■
WARN	■			■	■	■
INFO	■	■	■			■
DEBUG	■	■	■			
TRACE	■	■	■	■		

■ Level must be adjusted

Figure 2: Level must be adjusted between the ground truth and responses when calculating Adjusted Level Accuracy.

Position Accuracy (PA) is calculated as follows. If the position of the generated log statement matches the position of the actual log statement of ground truth in the source code for testing, *PA* = 1 (indicating a successful recommendation); otherwise, *PA* = 0 (indicating an unsuccessful recommendation).

Level Accuracy (LA) is calculated as the percentage of correctly predicted log levels, which involves comparing the predicted log levels against the actual log levels and determining how many predictions are correct. However, considering that there is no strict standard for log levels, and that different logging practitioners might have varying habits for assigning levels (e.g., the INFO and DEBUG level logs may not make a significant difference in real logging practice), it seems unfair to directly compare the log levels recommended by the model with the ground truth. Therefore, we propose *Adjusted Level Accuracy* (*Adj. LA*), which only examines specific log levels that must be adjusted when the model-recommended log level does not match the ground truth. The rationale behind *Adj. LA* is very straightforward, Fig. 2 shows all the levels that need to be adjusted, which constructed from large-scale log-related issues [24]. Firstly, the log verbosity levels are sequentially encoded from the lowest level TRACE (0) to the highest level FATAL (5). If recommended log statement falls into any of the levels in the corresponding right-hand column for a certain verbosity level (the left column), we consider it to be a wrong recommendation (i.e., the verbosity level must be adjusted), otherwise, we consider it to be a correct recommendation. For example, if the ground truth is the INFO level and the recommendation is DEBUG, we still consider it to be a correct Level recommendation. Then *Adj. LA* is calculated as the percentage of correct level recommendation to all the recommended results in this sense.

Message Accuracy (MA) calculates the proportion of words in the response that are also present in the real text in ground truth as $MA = |W_p \cap W_t|/|W_p|$, where W_p and W_t represent the constituent words split in model response and the real text in ground truth, respectively.

Text Accuracy is calculated using *BLEU* [46] in NLP to evaluate the performance of the message generated by different models.

BLEU method evaluates the similarity of the text by comparing the candidate text with one or more reference texts, which is broadly used in the machine translation task. In particular, we adopt the *BLEU-DM* variant [51], the sentence-level *BLEU* with no smoothing method included, to obtain a more precise evaluation of the log messages in a more concise way, maintaining the integrity and fidelity of the information in the log statements. To be specific, the *BLEU-DM* score is calculated based on n-gram precision and brevity penalty, i.e., $BLEU\text{-}DM = \text{BP} \cdot \exp\left(\sum w_n \log p_n\right)$, where p_n is the n-gram precision (i.e., the proportion of n-grams in which the candidate text matches the reference text), w_n is the n-gram weight, and $\text{BP} = \max\left(\exp\left(1 - r/c\right), 1\right)$ is the brevity penalty for penalizing too-short candidate texts, where c is the length of the candidate text, and r is the length of the reference text.

Variable Accuracy evaluate the accuracy of variables recorded in the log statements, we denote the variables in each generated log statement as V_p and the variables in each actual log statement of ground truth as V_t. Therefore, we calculated the *Variable Precision* $VP = |V_p \cap V_t|/|V_p|$, *Variable Recall* $VR = |V_p \cap V_t|/|V_t|$, and *Variable F1 score* $VF1 = 2 \times VP \times VR/(VP + VR)$ as metrics. Note that if the same variable is used differently in log statement, they will be treated as different variables.

Table 2: The experiment settings.

Method	Training & Validation Set	Testing Set
Original Model	N/A	LANCE testing set, LogBench-T
UniLog-w/ warmup	LANCE training set (as examples) and validation set (for warmup)	
FT	**Fine-tuning** (FT for short): LANCE training & validation set **Alignment**: *AucadLog*	
FT+*DPO* [*ours*]		
FT+*cDPO* [*ours*]		
FT+*SimPO* [*ours*]		

5.3.5 Experimental Settings. To comprehensively assess the performance of our approach and the impact of the resulting dataset, we established a series of experimental evaluation settings according to Section 5.3, as shown in Table 2.

Model fine-tuning and alignment were conducted using dual NVIDIA A100 80GB GPUs with BF16 precision. For fine-tuning, we used a micro-batch size of 2 with 16 gradient accumulation steps (global batch size 32) over 10 epochs. The alignment process employed LoRA under GPU constraints, using a micro-batch size of 1 with 64 accumulation steps (global batch size 64) over 5 epochs. Shared hyperparameters included: learning rate of 1e−5, max sequence length of 4096 tokens, and LoRA configuration ($r = 8$, $\alpha = 16$, dropout=0.05).

6 Result Analysis

In this section, we analyze the results obtained.

6.1 Validation with Logging Practices (RQ1)

Cohen's kappa coefficient calculated based on the overlapped 174 samples is 0.886, which shows that the two researchers are fairly consistent in evaluating the results. Yet after joint discussions, we finally only filtered 18 non-relevant diffs, which means that the overwhelming majority of our diffs are relevant to the related issues (808/826 = 97.8%), and their quality can be guaranteed by the quality of well-discussed issues in the Apache community. Evidently, minor modifications like fixing typos and merely paraphrasing static text are considered trivial and lack instructional value for understanding effective logging practices.

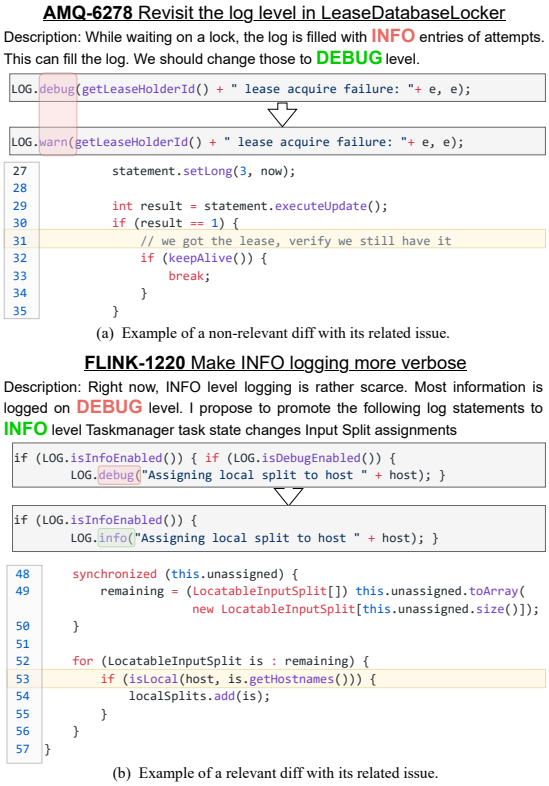

Figure 3: Example of non-relevant and relevant diffs with their related issues.

Nevertheless, the filtered diffs are only not that relevant to the issue rather than necessarily totally irrelevant to the issue. For example, as Fig. 3 (a) shows, in the call of adjusting the redundant INFO level log statement to DEBUG, the diff reversely raises the level of a log statement from DEBUG to WARN. In the context of the code, this modification may possibly be reasonable as the importance of the *acquire failure* message is worth the level of WARN, but anyway, it is different from the appeal of the issue. Consequently, the diff is excluded in our *AucadLog* dataset.

In contrast, in the upcoming example shown in Fig. 3 (b), the diff is congruent with the claim of the issue that more information should be logged in INFO level rather than the DEBUG level, as a result, we consider it relevant to the issue it related to.

6.2 Model Enhancement via Alignment (RQ2)

To address **RQ2**, we investigate how the *AucadLog* dataset can improve the capabilities of LLMs for the task of log statement recommendation after post-training.

The results with the optimal alignment hyperparameters determined after multiple attempts presented in Table 3 illustrate that

Table 3: Experimental evaluation results of the LANCE testing set and LogBench-T dataset.

Method (Backbone)	PA	LA	Adj. LA	MA	BLEU-DM	VP	VR	VF1
Results of the LANCE testing set:								
LANCE [42]	0.546	0.378	0.546	0.100	0.110	0.271	0.253	0.254
FastLog [59]	0.527	0.410	0.505	0.156	0.174	0.228	0.234	0.224
UniLog [61] (Llama 3.1)	0.523	0.522	0.674	0.239	0.255	0.356	0.348	0.345
UniLog [61] (Magicoder)	0.116	0.213	0.315	0.077	0.081	0.121	0.119	0.119
Llama 3.1	0.380	0.473	0.657	0.188	0.226	0.364	0.357	0.350
FT (Llama 3.1)	0.571 (+ 50.5%)	0.570 (+20.5%)	0.718 (+ 9.2%)	0.264 (+40.2%)	0.281 (+24.4%)	0.474 (+30.1%)	0.449 (+25.8%)	0.451 (+28.9%)
FT+*DPO* (Llama 3.1) [*ours*]	0.634 (+67.2%)	0.581 (+22.8%)	0.743 (+13.0%)	0.273 (+45.0%)	0.292 (+29.4%)	0.484 (+33.1%)	0.467 (+30.7%)	0.465 (+32.9%)
FT+*cDPO* (Llama 3.1) [*ours*]	**0.635** (+67.2%)	0.586 (+23.8%)	0.738 (+12.4%)	0.274 (+45.6%)	0.291 (+29.1%)	0.486 (+33.7%)	0.466 (+30.6%)	0.466 (+33.2%)
FT+*SimPO* (Llama 3.1) [*ours*]	**0.635** (+67.4%)	**0.591** (+25.0%)	**0.745** (+13.3%)	**0.275** (+46.1%)	**0.293** (+29.9%)	**0.488** (+34.2%)	**0.468** (+31.1%)	**0.468** (+33.8%)
Magicoder	0.303	0.432	0.559	0.148	0.180	0.285	0.275	0.272
FT (Magicoder)	0.493 (+62.4%)	0.429 (- 0.5%)	0.534 (- 4.4%)	0.182 (+23.3%)	0.202 (+11.8%)	0.343 (+20.0%)	0.326 (+18.3%)	0.325 (+19.6%)
FT+*DPO* (Magicoder) [*ours*]	0.507 (+67.3%)	**0.444** (+ 2.8%)	0.552 (- 1.2%)	0.189 (+27.7%)	0.210 (+16.5%)	0.351 (+23.0%)	0.334 (+21.5%)	0.334 (+22.7%)
FT+*cDPO* (Magicoder) [*ours*]	0.510 (+68.0%)	0.443 (+ 2.6%)	0.554 (- 0.9%)	**0.190** (+28.8%)	**0.211** (+17.1%)	**0.355** (+24.5%)	**0.339** (+23.1%)	**0.338** (+24.3%)
FT+*SimPO* (Magicoder) [*ours*]	**0.511** (+68.3%)	**0.444** (+ 2.9%)	0.554 (- 0.9%)	0.188 (+27.6%)	**0.211** (+16.8%)	0.354 (+23.9%)	0.337 (+22.5%)	0.336 (+23.7%)
Results of the LogBench-T dataset:								
LANCE [42]	0.506	0.356	0.539	0.059	0.071	0.341	0.333	0.332
FastLog [59]	0.493	0.429	0.571	0.095	0.118	0.330	0.339	0.329
UniLog [61] (Llama 3.1)	0.347	0.442	0.671	0.106	0.132	0.334	0.333	0.330
UniLog [61] (Magicoder)	0.158	0.210	0.345	0.041	0.049	0.141	0.140	0.139
Llama 3.1	0.411	0.503	0.714	0.100	0.144	0.396	0.393	0.388
FT (Llama 3.1)	0.592 (+44.1%)	0.529 (+ 5.3%)	0.708 (- 0.9%)	0.129 (+28.9%)	0.154 (+ 7.2%)	0.441 (+11.4%)	0.426 (+ 8.4%)	0.427 (+10.0%)
FT+*DPO* (Llama 3.1) [*ours*]	0.639 (+55.4%)	0.528 (+ 5.0%)	0.763 (+ 6.9%)	0.138 (+37.6%)	0.167 (+16.4%)	0.450 (+13.7%)	0.438 (+11.3%)	0.437 (+12.6%)
FT+*cDPO* (Llama 3.1) [*ours*]	0.640 (+55.7%)	0.525 (+ 4.3%)	0.767 (+ 7.4%)	**0.140** (+39.9%)	**0.168** (+16.8%)	0.459 (+15.8%)	0.445 (+13.3%)	0.445 (+14.6%)
FT+*SimPO* (Llama 3.1) [*ours*]	**0.642** (+56.1%)	**0.536** (+ 6.5%)	**0.769** (+ 7.7%)	**0.140** (+39.8%)	**0.168** (+16.6%)	**0.464** (+17.1%)	**0.451** (+14.7%)	**0.450** (+16.0%)
Magicoder	0.250	0.409	0.492	0.072	0.107	0.282	0.276	0.274
FT (Magicoder)	0.475 (+90.0%)	0.432 (+ 5.8%)	0.573 (+16.5%)	0.098 (+35.4%)	0.124 (+16.2%)	0.338 (+20.1%)	0.328 (+19.1%)	0.327 (+19.5%)
FT+*DPO* (Magicoder) [*ours*]	**0.495** (+97.8%)	0.448 (+ 9.6%)	**0.595** (+20.8%)	**0.108** (+49.5%)	0.131 (+22.8%)	0.347 (+23.1%)	0.337 (+22.1%)	0.335 (+22.4%)
FT+*cDPO* (Magicoder) [*ours*]	0.493 (+97.0%)	0.442 (+ 8.3%)	0.593 (+20.5%)	0.106 (+47.0%)	**0.132** (+23.3%)	0.350 (+24.2%)	**0.340** (+23.2%)	**0.338** (+23.4%)
FT+*SimPO* (Magicoder) [*ours*]	0.493 (+97.2%)	**0.449** (+ 9.9%)	**0.595** (+20.8%)	0.105 (+45.2%)	0.130 (+21.5%)	**0.351** (+24.4%)	**0.340** (+23.3%)	**0.338** (+23.6%)

† Llama 3.1 model adopts $\beta = 0.5$ for *DPO*, $\beta = 0.1$ for *cDPO*, $\beta = 2.5$ for *SimPO*; Magicoder model adopts $\beta = 0.5$ for *DPO*, $\beta = 1.0$ for both *cDPO* and *SimPO*.

the *AucadLog* dataset plays a crucial role in enhancing model performance through effective alignment strategies. Both the Llama 3.1 and Magicoder models exhibit significant improvements when aligned with the *cDPO* or *SimPO* algorithms, compared with only fine-tuned models, demonstrating that the quality and importance of training data are instrumental in optimizing model capabilities.

Position. In the Llama 3.1 model, the SimPO algorithm achieves a remarkable 67.4% increase in *Position Accuracy* (*PA*) compared with the original Llama 3.1, which indicates a substantial enhancement in the model's ability to accurately recommend log statement positions. This improvement is critical, as accurate positioning of log statements directly impacts the clarity and utility of logs for developers. In contrast, the fine-tuned model only achieves an increase of 50.5% and also shows improvements, but falls short of the performance gains achieved by alignment. Correspondingly, there is a similar improvement for Magicoder, although not as significant as Llama 3.1, which illustrates the benefit of using *AucadLog* dataset alignment. Even on unseen code in the LogBench-T dataset, alignment has a positive impact, with the performance of Llama 3.1 increasing by 12% compared to fine-tuning alone.

Level. The effectiveness of the *AucadLog* dataset is further evidenced in verbose-level recommendations. The Llama 3.1 model demonstrates increases in both *Log Level* (*LA*) and *Adjusted Log Level* (*Adj. LA*), with improvements of 25% and 13.3%, respectively. In contrast, the fine-tuned model also achieves 10.3% and 2.5% increase but it is less than the performance gains brought by alignment. On both Magicoder and LogBench-T datasets, evaluations of alignment models using the *AucadLog* dataset showed similar performance (except *Adj. LA* on Magicoder). This highlights the ability of the model to provide more contextually appropriate verbosity in log statements, which can enhance readability and understanding for users.

Content. For the textual content in the log statements, we adopt *MA* and *BLEU-DM* for the accuracy of message text in the log statements, together with *Variable Precision* (*VP*), *Variable Recall* (*VR*), and *Variable F1* (*VF1*) for the accuracy of variables in log message. The results demonstrate that alignment techniques, especially *SimPO*, bring significant improvements across these metrics. For *MA* and *BLEU-DM*, *SimPO* consistently enhances the accuracy of message text, with *BLEU-DM* scores indicating closer alignment between generated and reference text. This boost in *BLEU-DM* scores suggests that the aligned models produce log messages with content that is both semantically and structurally more accurate. In terms of variable accuracy, the *VP*, *VR*, and *VF1* metrics show notable gains with alignment algorithms, with *SimPO* providing the most substantial improvements. For example, *SimPO* raises *VP* by up to 34.2% in Llama 3.1 on the LANCE testing set, indicating enhanced precision in selecting correct variables. Similarly, *VR* and *VF1* improvements reflect better recall and balanced precision-recall performance, pointing to greater model flexibility and consistency in handling variable content. Overall, these results highlight that *SimPO* and related alignment algorithms improve both the textual accuracy and variable handling in log statements, thus enabling more effective and reliable log recommendations.

In summary, the findings strongly indicate that the alignment facilitated by our *AucadLog* dataset is pivotal in improving the performance of LLMs. Notably, our implementation of UniLog shows a certain degree of performance improvement compared to its base

model, Llama 3.1 (8B), aligning with the improvement reported in its original paper. This consistency suggests that our implementation is likely correct. However, in terms of impact, prompt engineering methods represented by UniLog do not demonstrate significant advantages at the 8B parameter scale of LLMs. In fact, post-training methods, especially alignment algorithms such as *SimPO*, *DPO*, and *cDPO*, show more substantial performance gains, particularly excelling in key metrics of accuracy and consistency in generated content. This indicates that for models with smaller parameter scales, post-training alignment optimization may be more effective than prompt engineering.

7 Discussion

In this section, we discuss some implications of our study.

7.1 High-quality Dataset and its Automatic Construction

High-quality datasets are crucial in the post-training of LLMs, especially in the alignment. Our work involves extracting the code before and after the resolution of log-related issues in open-source software systems. The extracted code includes both log statements and surrounding contextual code, serving as training corpora for generating high-quality log statements. The results have been exceptionally promising. Since these log-related issues have already been resolved, they represent optimal examples of where and what to log, two of the most critical aspects regarding log statements [7, 22]. Otherwise, the issues would not have been raised and addressed. Moreover, the value of these datasets extends beyond positive examples. They also contain instances of poor or sub-optimal log statements – commonly referred to as negative examples. Applying alignment techniques, like *DPO* [48], *cDPO* [44], and *SimPO* [43], our approach and dataset have yielded outstanding results, with enhancing open-source LLMs outperforming the current state-of-the-art models in generating log statements. Another noteworthy point is that this method of dataset construction has to be automated given the huge labor costs. In this sense, our approach shows full potential, as the size of our dataset can be increased by incorporating more open source projects and log-related issues. To better implement the value of our work, we plan to make this dataset publicly available (see Section **Data Availability**) to allow interested researchers to replicate our study or explore further.

7.2 The Significance of Automated Log Statement Generation

Since large language models (LLMs) have been integrated into code completion and generation tools, their performance has gained substantial recognition, as demonstrated by the widespread adoption of tools like Copilot. These tools have attracted a considerable user base, and multiple reports suggest that they significantly enhance production efficiency [45, 58]. We contend that automated log statement generation is particularly critical within the realm of LLM-enabled code activities. Log statements are essential components of modern software systems, and if their creation does not keep pace with other business code, they could become a bottleneck that impedes overall software development productivity.

Moreover, the quality of the data significantly impacts the fine-tuning of models. Our constructed dataset effectively supports the fine-tuning of LLMs, thereby enhancing the generation of log statements. By fine-tuning open-source LLMs and subsequently utilizing the *AucadLog* dataset to align models such as Llama 3.1 and Magicoder, we have demonstrated significant improvements in log statement generation. These fine-tuned models notably outperform UniLog and state-of-the-art fine-tuning approaches across multiple evaluation metrics.

Furthermore, our observations also indicate that our dataset can counteract the performance decline commonly associated with fine-tuning. Additionally, the degree of improvement achieved through alignment with our dataset surpasses that obtained with the fine-tuned models, underscoring the efficacy of high-quality data.

8 Threats to Validity

In this section, we discuss some factors that may potentially bring risks to the findings of the study.

Limited LLMs. Due to our constraint on GPU capabilities, we conducted fine-tuning experiments on Llama 3.1 and Magicoder, which inevitably introduced some limitations in result generalization. However, *AUCAD* is designed as a model-agnostic method to work with multiple LLMs, and we have also made our dataset and original response from LLMs publicly available (see Section **Data Availability**). We encourage interested researchers to replicate our work on a broader range of LLMs and thoroughly validate the effectiveness of the proposed method.

Only involving log-related issues. Limiting our dataset to log-related issues imposes constraints on its diversity and the coverage of real-world logging scenarios. In practice, certain log statements and their associated code snippets are inherently sensible and may not be flagged as issues. Unfortunately, these valuable data points are excluded from our dataset. On the one hand, distilling high-quality datasets from issues is an effective way to fine-tune LLMs optimized for the log statement generation task. On the other hand, there is currently no automated method for extracting high-quality log statements and context code directly from open-source projects. Manual distillation, while theoretically possible, is too costly and time-consuming. Nevertheless, it is worth noting that the *AucadLog* dataset has already covered all log-related issues in Apache (as discussed in Section 4). As more data added to the dataset in the future, this problem should gradually be alleviated.

Replication bias. Since the dataset, source code as well as the base LLM model are not accessible, we have to replicate the UniLog method, which may lead to replication bias. However, the experimental results from our replicated UniLog show similar results to the metrics of the original paper [61] in terms of the increasing trends of all the evaluation metrics, which to a large degree lowered the risk pertinent to replication bias.

About the level evaluation. To more accurately assess the log levels recommended by our model, we have introduced the metric *Adjust Level Accuracy* (*Adj. LA*), which focuses only on the log levels that need adjustment. However, it should be noted that this is not a perfect solution but rather a more reasonable compromise towards a pragmatic logging practice. We believe that further exploration is needed to develop more reasonable evaluation mechanisms.

About the content evaluation. Borrowing from previous studies [42, 61], we also employed text similarity metrics, such as *BLEU* [46] and its variants (e.g., *BLEU-DM* [51]), to measure the quality of the generated log statements. However, the inherent content generation nature of LLMs poses challenges to objective and accurate evaluation using metrics like *BLEU* and its variants, which also has been acknowledged in prior research [13]. To mitigate this threat, we extracted variables when evaluating log statement content. We employed metrics of variables for evaluations, e.g., *Variable Precision* (*VP*), *Variable Recall* (*VR*), and *Variable F1 score* (*VF1*), thereby providing multiple perspectives for evaluating the content of the log statements generated by LLM. We acknowledge the validity risk associated with using this metric and will keep an open mind to tracking research advances to find better ways to evaluate the quality of log statement content.

9 Conclusion

Log data plays a pivotal role in the operation and maintenance of modern software systems. Logs are generated through log statements strategically placed throughout the software codebase. With the rapid advancements in Large Language Models (LLMs), there is potential for significantly improving code completion tasks, which includes the automated generation of log statements. As a result, the automated generation of log statements has gained prominence in contemporary software development. Despite the progress in LLMs, their current capabilities in generating log statements still require enhancement to truly benefit software development processes [32]. Post-training with high-quality dataset is a widely accepted solution, however the huge cost of manual annotation makes this solution not very feasible in practice. We address this gap by proposing *AUCAD*, an automated method for constructing a high-quality post-training dataset, and by contributing *AucadLog*, a custom-curated dataset comprising pre- and post-changed log statements and their corresponding code segments from real-world log-related issues in the open-source community. By fine-tuning open-source LLMs then using *AucadLog* to align them, Llama 3.1 and Magicoder, to devise exemplar models with the *AucadLog* dataset, the fine-tuned models achieve significant improvements in log statement generation, notably outperforming UniLog and the state-of-the-art finetuning methods in terms of multiple evaluation metrics.

Future research should consider extending the *AUCAD* method to construct preference alignment datasets across a wider array of scenarios, incorporating various LLMs, diverse parameter scales, and multiple programming languages. Additionally, exploring more nuanced code context ranges may offer further improvements in model performance.

Data Availability

Our *AucadLog* dataset and the original response of LLMs are publicly available at https://github.com/aiopsplus/AucadLog.

References

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (Online). Association for Computational Linguistics, 2655–2668. doi:10.18653/V1/2021.NAACL-MAIN.211

[2] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T. Barr. 2024. Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization). arXiv:2304.06815 [cs.SE] https://arxiv.org/abs/2304.06815

[3] AI@Meta. 2024. Llama 3.1 Model Card. https://github.com/meta-llama/llama-models/blob/main/models/llama3_1/MODEL_CARD.md

[4] Kaiyan Chang, Kun Wang, Nan Yang, Ying Wang, Dantong Jin, Wenlong Zhu, Zhirong Chen, Cangyuan Li, Hao Yan, Yunhao Zhou, Zhuoliang Zhao, Yuan Cheng, Yudong Pan, Yiqi Liu, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. 2024. Data is all you need: Finetuning LLMs for Chip Design via an Automated design-data augmentation framework. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (San Francisco, CA, USA) *(DAC '24)*. Association for Computing Machinery, New York, NY, USA, Article 60, 6 pages. doi:10.1145/3649329.3657356

[5] Boyuan Chen. 2019. Improving the software logging practices in DevOps. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE / ACM, 194–197. doi:10.1109/ICSE-COMPANION.2019.00080

[6] 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* 24, 4 (2019), 2285–2322. doi:10.1007/S10664-019-09690-0

[7] Boyuan Chen and Zhen Ming (Jack) Jiang. 2021. A Survey of Software Log Instrumentation. *ACM Comput. Surv.* 54, 4, Article 90 (may 2021), 34 pages. doi:10.1145/3448976

[8] Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Yao Liu, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, and Quoc V. Le. 2023. Symbolic Discovery of Optimization Algorithms. arXiv:2302.06675 [cs.LG] https://arxiv.org/abs/2302.06675

[9] Paul F. Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 4302–4310.

[10] Maëlick Claes and Mika V. Mäntylä. 2020. 20-MAD: 20 Years of Issues and Commits of Mozilla and Apache Development. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 503–507. doi:10.1145/3379597.3387487

[11] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[12] 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, 349–360. doi:10.1109/SANER53432.2022.00051

[13] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the BLEU: How should we assess quality of the Code Generation models? *J. Syst. Softw.* 203, C (sep 2023), 17 pages. doi:10.1016/j.jss.2023.111741

[14] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53. doi:10.1109/ICSE-FoSE59343.2023.00008

[15] 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* (Hyderabad, India) *(ICSE Companion 2014)*. Association for Computing Machinery, New York, NY, USA, 24–33. doi:10.1145/2591062.2591175

[16] Ying Fu, Meng Yan, Pinjia He, Chao Liu, Xiaohong Zhang, and Dan Yang. 2024. End-to-end log statement generation at block-level. *Journal of Systems and Software* 216 (2024), 112146. doi:10.1016/j.jss.2024.112146

[17] Radoslav Gatev. 2021. *Observability: Logs, Metrics, and Traces*. Apress, Berkeley, CA. 233–252 pages. doi:10.1007/978-1-4842-6998-5_12

[18] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 39, 13 pages. doi:10.1145/3597503.3608134

[19] Mohammad Gheshlaghi Azar, Zhaohan Daniel Guo, Bilal Piot, Remi Munos, Mark Rowland, Michal Valko, and Daniele Calandriello. 2024. A General Theoretical Paradigm to Understand Learning from Human Preferences. In *Proceedings of The 27th International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 238)*. PMLR, 4447–4455. https://proceedings.mlr.press/v238/gheshlaghi-azar24a.html

[20] Sina Gholamian. 2021. Leveraging Code Clones and Natural Language Processing for Log Statement Prediction. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1043–1047. doi:10.1109/ASE51524.

2021.9678596

[21] Shenghui Gu, Guoping Rong, Tian Ren, He Zhang, Haifeng Shen, Yongda Yu, Xian Li, Jian Ouyang, and Chunan Chen. 2023. TrinityRCL: Multi-Granular and Code-Level Root Cause Localization Using Multiple Types of Telemetry Data in Microservice Systems. *IEEE Transactions on Software Engineering* 49, 5 (2023), 3071–3088. doi:10.1109/TSE.2023.3241299

[22] Shenghui Gu, Guoping Rong, He Zhang, and Haifeng Shen. 2023. Logging Practices in Software Engineering: A Systematic Mapping Study. *IEEE Transactions on Software Engineering* 49, 2 (2023), 902–923. doi:10.1109/TSE.2022.3166924

[23] Hongcheng Guo, Jian Yang, Jiaheng Liu, Jiaqi Bai, Boyang Wang, Zhoujun Li, Tieqiao Zheng, Bo Zhang, Junran Peng, and Qi Tian. 2024. LogFormer: A Pre-train and Tuning Pipeline for Log Anomaly Detection. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024* (Vancouver, Canada), Vol. 38. AAAI Press, 135–143. doi:10.1609/AAAI.V38I1.27764

[24] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. 2018. Studying and detecting log-related issues. *Empirical Softw. Engg.* 23, 6 (dec 2018), 3248–3280. doi:10.1007/s10664-018-9603-z

[25] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685 [cs.CL] https://arxiv.org/abs/2106.09685

[26] Yuan Huang, Yinan Chen, Xiangping Chen, Junqi Chen, Rui Peng, Zhicao Tang, Jinbo Huang, Furen Xu, and Zibin Zheng. 2024. Generative Software Engineering. arXiv:2403.02583 [cs.SE] https://arxiv.org/abs/2403.02583

[27] 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)*, Vol. 1. IEEE Computer Society, 326–337. doi:10.1109/SANER.2016.29

[28] Amirmahdi Khosravi Tabrizi, Naser Ezzati-Jivan, and Francois Tetreault. 2024. An Adaptive Logging System (ALS): Enhancing Software Logging with Reinforcement Learning Techniques. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering* (London, United Kingdom) *(ICPE '24)*. Association for Computing Machinery, New York, NY, USA, 37–47. doi:10.1145/3629526.3645033

[29] Blagovesta Kostova, Seda Gürses, and Carmela Troncoso. 2020. Privacy Engineering Meets Software Engineering. On the Challenges of Engineering Privacy ByDesign. arXiv:2007.08613 [cs.SE] https://arxiv.org/abs/2007.08613

[30] Van-Hoang Le and Hongyu Zhang. 2021. Log-based Anomaly Detection Without Log Parsing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 492–504. doi:10.1109/ASE51524.2021.9678773

[31] Van-Hoang Le and Hongyu Zhang. 2022. Log-based anomaly detection with deep learning: how far are we?. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1356–1367. doi:10.1145/3510003.3510155

[32] Yichen Li, Yintong Huo, Zhihan Jiang, Renyi Zhong, Pinjia He, Yuxin Su, Lionel C. Briand, and Michael R. Lyu. 2024. Exploring the Effectiveness of LLMs in Automated Logging Statement Generation: An Empirical Study. *IEEE Transactions on Software Engineering* 50, 12 (2024), 3188–3207. doi:10.1109/TSE.2024.3475375

[33] 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. *Proc. ACM Softw. Eng.* 1, FSE, Article 28 (jul 2024), 22 pages. doi:10.1145/3643754

[34] 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, 1461–1472. doi:10.1109/ICSE43902.2021.00131

[35] Dayi Lin, Filipe Roseiro Cogo, Gopi Krishnan Rajbahadur, and Ahmed E. Hassan. 2024. Technical Brief on Software Engineering for FMware. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings* (Lisbon, Portugal) *(ICSE-Companion '24)*. Association for Computing Machinery, New York, NY, USA, 431–433. doi:10.1145/3639478.3643062

[36] Xinyu Lin, Wenjie Wang, Yongqi Li, Shuo Yang, Fuli Feng, Yinwei Wei, and Tat-Seng Chua. 2024. Data-efficient Fine-tuning for LLM-based Recommendation. arXiv:2401.17197 [cs.IR] https://arxiv.org/abs/2401.17197

[37] 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 2022)*. Association for Computing Machinery, New York, NY, USA, 27–38. doi:10.1145/3533767.3534379

[38] Yilun Liu, Shimin Tao, Weibin Meng, Feiyu Yao, Xiaofeng Zhao, and Hao Yang. 2024. LogPrompt: Prompt Engineering Towards Zero-Shot and Interpretable Log Analysis. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings* (Lisbon, Portugal) *(ICSE-Companion '24)*. Association for Computing Machinery, New York, NY, USA, 364–365. doi:10.1145/3639478.3643108

[39] Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2021. Which Variables Should I Log? *IEEE Transactions on Software Engineering* 47, 9 (2021), 2012–2031. doi:10.1109/TSE.2019.2941943

[40] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 647–658. doi:10.1109/ISSRE59848.2023.00026

[41] Antonio Mastropaolo, Valentina Ferrari, Luca Pascarella, and Gabriele Bavota. 2024. Log statements generation via deep learning: Widening the support provided to developers. *Journal of Systems and Software* 210 (2024), 111947. doi:10.1016/j.jss.2023.111947

[42] 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* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2279–2290. doi:10.1145/3510003.3511561

[43] Yu Meng, Mengzhou Xia, and Danqi Chen. 2024. SimPO: Simple Preference Optimization with a Reference-Free Reward. In *Advances in Neural Information Processing Systems*, Vol. 37. Curran Associates, Inc., 124198–124235. https://proceedings.neurips.cc/paper_files/paper/2024/file/e099c1c9699814af0be873a175361713-Paper-Conference.pdf

[44] Eric Mitchell. 2023. A note on DPO with noisy preferences & relationship to IPO. https://ericmitchell.ai/cdpo.pdf

[45] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. 2023. GitHub Copilot AI pair programmer: Asset or Liability? *J. Syst. Softw.* 203, C (sep 2023), 23 pages. doi:10.1016/j.jss.2023.111734

[46] 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 on Association for Computational Linguistics* (Philadelphia, Pennsylvania) *(ACL '02)*. Association for Computational Linguistics, USA, 311–318. doi:10.3115/1073083.1073135

[47] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. 2015. Industry Practices and Event Logging: Assessment of a Critical Software Development Process. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE Computer Society, 169–178. doi:10.1109/ICSE.2015.145

[48] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. In *Advances in Neural Information Processing Systems*, Vol. 36. Curran Associates, Inc., 53728–53741. https://proceedings.neurips.cc/paper_files/paper/2023/file/a85b405ed65c6477a4fe8302b5e06ce7-Paper-Conference.pdf

[49] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

[50] Guoping Rong, Yangchen Xu, Shenghui Gu, He Zhang, and Dong Shao. 2020. Can You Capture Information As You Intend To? A Case Study on Logging Practice in Industry. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 12–22. doi:10.1109/ICSME46990.2020.00012

[51] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the evaluation of neural code summarization. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1597–1608. doi:10.1145/3510003.3510060

[52] Jiahao Shi, Sihang Jiang, Bo Xu, and Yanghua Xiao. 2023. ServerRCA: Root Cause Analysis for Server Failure using Operating System Logs. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 486–496. doi:10.1109/ISSRE59848.2023.00083

[53] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, Hanwei Qian, Yang Liu, and Zhenyu Chen. 2023. Automatic Code Summarization via ChatGPT: How Far Are We? arXiv:2305.12865 [cs.SE] https://arxiv.org/abs/2305.12865

[54] Fahim Tajwar, Anikait Singh, Archit Sharma, Rafael Rafailov, Jeff Schneider, Tengyang Xie, Stefano Ermon, Chelsea Finn, and Aviral Kumar. 2024. Preference Fine-Tuning of LLMs Should Leverage Suboptimal, On-Policy Data. arXiv:2404.14367 [cs.LG] https://arxiv.org/abs/2404.14367

[55] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936. doi:10.1109/TSE.2024.3368208

[56] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2022. Finetuned Language Models Are Zero-Shot Learners. arXiv:2109.01652 [cs.CL] https://arxiv.org/abs/2109.01652

[57] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering Code Generation with OSS-Instruct. arXiv:2312.02120 [cs.CL] https://arxiv.org/abs/2312.02120

[58] Michel Wermelinger. 2023. Using GitHub Copilot to Solve Simple Programming Problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 172–178. doi:10.1145/3545945.3569830

[59] Xiaoyuan Xie, Zhipeng Cai, Songqiang Chen, and Jifeng Xuan. 2024. FastLog: An End-to-End Method to Efficiently Generate and Insert Logging Statements. arXiv:2311.02862 [cs.SE] https://arxiv.org/abs/2311.02862

[60] Zhenchang Xing, Qing Huang, Yu Cheng, Liming Zhu, Qinghua Lu, and Xiwei Xu. 2023. Prompt Sapper: LLM-Empowered Software Engineering Infrastructure for AI-Native Services. arXiv:2306.02230 [cs.SE] https://arxiv.org/abs/2306.02230

[61] Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liqun Li, Yu Kang, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. 2024. UniLog: Automatic Logging via LLM and In-Context Learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 14, 12 pages. doi:10.1145/3597503.3623326

[62] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Shaochen Zhong, Bing Yin, and Xia Hu. 2024. Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond. *ACM Trans. Knowl. Discov. Data* 18, 6, Article 160 (apr 2024), 32 pages. doi:10.1145/3649506

[63] Mengyuan Yang, Mengying Zhu, Yan Wang, Linxun Chen, Yilei Zhao, Xiuyuan Wang, Bing Han, Xiaolin Zheng, and Jianwei Yin. 2024. Fine-Tuning Large Language Model Based Explainable Recommendation with Explainable Quality Reward. In *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI Press, 9250–9259. doi:10.1609/aaai.v38i8.28777

[64] Yongcheng Zeng, Guoqing Liu, Weiyu Ma, Ning Yang, Haifeng Zhang, and Jun Wang. 2024. Token-level Direct Preference Optimization. arXiv:2404.11999 [cs.CL] https://arxiv.org/abs/2404.11999

[65] Haonan Zhang, Yiming Tang, Maxime Lamothe, Heng Li, and Weiyi Shang. 2022. Studying logging practice in test code. *Empirical Softw. Engg.* 27, 4 (jul 2022), 45 pages. doi:10.1007/s10664-022-10139-0

[66] Yang Zhang, Xiaosong Chang, Lining Fang, and Yifan Lu. 2023. DeepLog: Deep-Learning-Based Log Recommendation. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 88–92. doi:10.1109/ICSE-Companion58688.2023.00031

[67] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2024. Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code. arXiv:2311.07989 [cs.CL] https://arxiv.org/abs/2311.07989

[68] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2025. A Survey of Large Language Models. arXiv:2303.18223 [cs.CL] https://arxiv.org/abs/2303.18223

[69] Yao Zhao, Rishabh Joshi, Tianqi Liu, Misha Khalman, Mohammad Saleh, and Peter J. Liu. 2023. SLiC-HF: Sequence Likelihood Calibration with Human Feedback. arXiv:2305.10425 [cs.CL] https://arxiv.org/abs/2305.10425

[70] 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*, Vol. 1. IEEE Computer Society, 415–425. doi:10.1109/ICSE.2015.60

[71] Jing Zhu, Guoping Rong, Guocheng Huang, Shenghui Gu, He Zhang, and Dong Shao. 2019. JLLAR: A Logging Recommendation Plug-in Tool for Java. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware* (Fukuoka, Japan) *(Internetware '19)*. Association for Computing Machinery, New York, NY, USA, Article 16, 6 pages. doi:10.1145/3361242.3361261