Benchmarking Open-Source Large Language Models for Log Level Suggestion

Yi Wen Heng[1], Zeyang Ma[1], Zhenhao Li[2], Dong Jae Kim [3], Tse-Hsun (Peter) Chen[1]
[1]*Software PErformance, Analysis, and Reliability (SPEAR) lab, Concordia University, Montreal, Canada*
[2]*York University, Toronto, Canada*
[3]*DePaul University, Chicago, Illinois, USA*
he_yiwen@encs.concordia.ca, m_zeyang@encs.concordia.ca, lzhenhao@yorku.ca
djaekim086@gmail.com, peterc@encs.concordia.ca

***Abstract*—Large Language Models (LLMs) have become a focal point of research across various domains, including software engineering, where their capabilities are increasingly leveraged. Recent studies have explored the integration of LLMs into software development tools and frameworks, revealing their potential to enhance performance in text and code-related tasks. Log level is a key part of a logging statement that allows software developers control the information recorded during system runtime. Given that log messages often mix natural language with code-like variables, LLMs' language translation abilities could be applied to determine the suitable verbosity level for logging statements. In this paper, we undertake a detailed empirical analysis to investigate the impact of characteristics and learning paradigms on the performance of 12 open-source LLMs in log level suggestion. We opted for open-source models because they enable us to utilize in-house code while effectively protecting sensitive information and maintaining data security. We examine several prompting strategies, including Zero-shot, Few-shot, and fine-tuning techniques, across different LLMs to identify the most effective combinations for accurate log level suggestions. Our research is supported by experiments conducted on 9 large-scale Java systems. The results indicate that although smaller LLMs can perform effectively with appropriate instruction and suitable techniques, there is still considerable potential for improvement in their ability to suggest log levels.**

I. Introduction

Logs are invaluable for tracking system runtime behavior. Logs contribute to software quality assurance by monitoring system performance and reliability [1], [2], debugging [3], [4], failure diagnosis [5], [6], program comprehension [7], [8], [9], [10], and anomaly detection [11], [12]. As an example, the logging statement `LOG.debug("Task FINISHED, but concurrently went to state " + state);` is at the "*debug*" level and contains the log message "*Task FINISHED, but concurrently went to state*", and records the value for the `state` variable. Such logging statements capture important runtime information for later analysis.

Despite their importance, the massive volume of logs generated by modern software systems, often reaching tens of gigabytes or even terabytes daily [13], [14], [15], poses significant challenges in log management and analysis, potentially bottlenecking quality assurance processes. To facilitate log management, verbosity levels (e.g., *trace*, *debug*, *info*, *warn*, and *error*) are employed to indicate the urgency and prioritization of log analysis. Logs can also be wrapped in log guards, such as `isDebugEnabled` or `isErrorEnabled`, which optimize performance by verifying whether specific logging levels are enabled before generating log messages.

However, selecting appropriate verbosity levels in logging statements can be challenging due to a limited understanding of system runtime behaviors [16], [17]. This often leads to incorrect log-level assignments, influenced by subjective interpretations or human error [16]. Such misclassifications can result in critical messages being overlooked or trivial events being misrepresented, complicating log management and analysis efforts [18], [16] and imposing additional overhead on these processes [16], [19]. Tools like GitHub Copilot, which are powered by large language models (LLMs), offer a potential solution to mitigate these challenges by providing automated code improvement and competition, including log level suggestions [20]. Yet, despite the usefulness of these tools, companies such as Apple, Samsung, and Amazon have banned the use of AI tools powered by Large Language Models (LLMs) due to concerns about compromising proprietary code or sensitive information [21], [22].

In this paper, we conducted a comprehensive empirical evaluation of various open-source LLMs. Our focus on open-source models stems from the desire to enable users to leverage LLMs without concerns about data privacy. Given that log levels are essential for code improvement, we use log level suggestions as a key task to assess the performance of these models. We evaluated models of different sizes and architectures, including general-purpose language models like BERT and RoBERTa, as well as code-specific models such as CodeBERT and GraphCodeBERT.

For the experiment, we used a benchmark dataset of logging statements from nine large-scale, open-source Java systems. These systems span various domains and represent real-world projects, making them suitable for examining logging practices and log level predictions. We evaluated each LLM under zero-shot, few-shot, and fine-tuning paradigms to assess their effectiveness in suggesting appropriate verbosity levels (e.g., *debug*, *info*, *warn*, *error*) for log statements.

We first preprocessed the dataset to extract logging statements alongside their corresponding source code, isolating the relevant code segments and log messages for each event. Then, we created prompts for the LLMs that included both

the log message and the surrounding code context, aiming to simulate real-world logging scenarios faced by developers. Following prior studies [23], [24], we use Accuracy, Area Under the Curve (AUC), and Average Ordinal Distance Score (AOD), to assess the performance of LLMs in suggesting log levels. We also explored how incorporating additional context, such as calling methods, influenced the model's accuracy in log level predictions. This experimental setup enabled us to benchmark both small and large models, providing insights into how model size, fine-tuning, and contextual information affect performance in practical software logging tasks. In summary, the contributions of our paper are as follows[1]:

- We discover that Fill-Mask models such as GraphCodeBERT, when fine-tuned, can outperform larger models in log level suggestions. Fine-tuning using task-specific data leads to substantial improvements in accuracy and makes these models competitive with state-of-the-art methods.
- We identify the impact of additional context, such as including the source code of calling methods, on LLM performance, noting that it can decrease accuracy and increase invalid outputs. The finding shows that adding undistilled information may negatively influence model output.
- We highlight the critical role of task-specific data in optimizing LLMs and show that Text Generation LLMs are more effective when such data is unavailable. This lays the groundwork for future research focused on enhancing LLMs for code-related tasks.

Paper organization. Section II introduces the background and related works to our study. Section III describes the overall design of our study. Section IV presents the study results. Section V offers actionable insights and suggestions for future directions. Section VI discusses the threats to validity. Section VII concludes the paper.

II. Background and Related Work

In this section, we discuss the background and related work of our study.

A. Logging

Log Level Suggestion. Prior works have investigated log-level suggestions using various machine learning and deep learning techniques [26], [23], [27], [24]. Li et al. [26] utilized the ordinal regression machine learning model to recommend appropriate log levels for logging statements. Ouatiti et al. [27] uses ordinal regression model and focuses on the performance of the log level suggestion that is trained separately multi-component software systems and their components. Li et al. [23] uses Ordinal Based Neural Networks to analyze syntactic context and message features of logging statements in order to suggest log levels. Liu et al. [24] applied graph neural networks to encode intra-block and inter-block features into code block representations, guiding log level suggestions.

[1]Our replication package is available at [25].

In contrast, our study uses open-source LLMs for log level suggestion to examine how different attributes and learning paradigms influence model performance. This approach makes use of readily available technology and offers insights into how various LLM characteristics and learning strategies can be effectively applied to log level suggestion.

Logging Statement Generation with LLMs. Some studies have explored logging statement generation using LLMs. For instance, Li et al. [28] proposed incorporating static context into code prompts using a self-refinement approach based on chain-of-thought (COT) prompts derived from static analysis. Xu et al. [29] utilized Codex, a fine-tuned GPT language model [30], to generate logging statements, examining both In-Context Learning and fine-tuning approaches. Unlike our approach, which utilizes open-source language models, Xu et al. [29] employed several models from the GPT-3 series for comparison purposes. Another close work to our study is by Li et al. [31], where they conducted an empirical study on the ability of LLM in logging statement generation by creating a dataset of unseen code, selected 11 general-purpose, logging-specific, and code-based LLMs, and used prompt instruction on the LLMs to generate result. While their study covered the In-Context Learning paradigm, it did not include fine-tuning.

While prior research has focused on automatically generating logging statements and integrating LLM-based tools into software development, our study benchmarks LLMs for log level suggestions. We examine how model size, task types, and learning paradigms influence outcomes. Our findings shed light on improving log level suggestions and lay the groundwork for future studies to enhance code quality through LLMs.

B. Large Language Models

In recent years, advancements in natural language processing (NLP) have been transforming various fields, including software engineering. This transformation highlights the importance of LLMs in code-related work and their significance in coding tasks [32], [33], [34], [35], which are essential for enhancing artificial intelligence in software engineering (AI4SE). Recent research has extensively explored the integration of LLMs into software engineering tools and processes to enhance development practices and advance both academic and industry applications. Studies have examined how LLMs improve tasks such as logging statement generation [28], [29], [30] and log parsing [36], [37]. Despite these advancements, several areas remain unclear and warrant further investigation:

*1) **Characteristics of LLMs**:* The effectiveness of various LLMs in performing specific tasks remains an area of active research. The following characteristics of LLMs contribute to their performance and influence how well they can handle different applications:

Task Types. Text Generation models, like those trained with autoregressive techniques (e.g., LLaMA 2), excel in generating coherent and contextually relevant text by modeling long-range dependencies and ensuring sequential consistency from given

prompts [38]. In contrast, Fill-Mask models like BERT and RoBERTa are trained by masking random tokens in a sentence and predicting them based on both left and right context. This bidirectional approach helps the model leverage context from both directions, making them particularly effective for tasks requiring deep contextual understanding [39], [40].

Parameter Sizes. The size of an LLM, indicated by the number of parameters, affects its capacity to learn and generalize. Larger models often have more expressive power but require more computational resources [41].

Pre-training Objectives. NLP-based LLMs are trained on natural language text and are designed to perform a variety of language tasks, including question-answering, translation, summarization, and text generation. Code-based LLMs are specifically trained on programming languages and are tailored to understand and generate code [42], [43]. However, some code LLMs, such as CodeLlama, are fine-tuned from LLaMA2, a NLP-based language model, rather than being trained exclusively on code. This fine-tuning process adapts the base model to better handle code-specific tasks [44].

*2) **Learning Paradigms of LLMs**:* LLMs are highly adaptable due to their ability to leverage transfer learning, which enables them to apply pre-trained knowledge across various tasks. However, they may still face difficulties with certain tasks because of limited domain knowledge. To further tailor LLMs for more context-specific tasks, two key learning paradigms are commonly employed:

In-Context Learning. In-Context Learning (ICL) is a method where an LLM is provided with a prompt that includes detailed instructions, task-specific demonstrations, or both. This enables the model to adjust its responses to the new task based on the provided context, without altering its underlying parameters. By leveraging its pre-trained knowledge, the model generates responses based on the provided context [45], [46]. However, ICL has limitations. The model's context size limits the number of examples that can be included, which may reduce its effectiveness. Additionally, processing multiple examples can increase computational and financial costs due to increased input tokens [41], leading to longer inference times.

Fine-tuning. In fine-tuning, a LLM undergoes a secondary training phase on a more specific dataset related to a particular task or domain [47], [48]. This process involves adjusting the model's parameters based on the new data, which enhances the model's performance on tasks relevant to the fine-tuning dataset. However, fine-tuning LLMs can be time-consuming because it involves updating full parameters. To address this, recent techniques have been introduced that only adjust additional parameters, thereby accelerating the fine-tuning process. Techniques like LoRA (Low-Rank Adaptation) can be used during fine-tuning to efficiently adapt pre-trained models by introducing low-rank matrices into the network, which helps manage the computational cost while preserving model performance [49]. Unlike in-context learning, which temporarily adjusts the model based on the input, fine-tuning changes the model's behavior and knowledge.

TABLE I: An overview of the log level distribution across nine large scale systems.

System	Version	LOC	NOL	Trace	Debug	Info	Warn	Error	Fatal
Cassandra	3.11.4	432K	1.3K	16.7%	10.9%	15.8%	16.8%	39.8%	0.0%
ElasticSearch	7.4.0	1.50M	2.5K	28.5%	32.4%	10.0%	19.2%	9.9%	0.0%
Flink	1.8.2	177K	2.5K	1.0%	30.8%	26.6%	23.7%	17.9%	0.0%
HBase	2.2.1	1.26M	5.5K	7.4%	17.3%	17.1%	24.4%	33.8%	0.0%
JMeter	5.3.0	143K	1.9K	0.7%	29.9%	16.9%	26.5%	26.0%	0.0%
Kafka	2.3.0	267K	1.5K	12.9%	28.5%	20.4%	15.3%	22.9%	0.0%
Karaf	4.2.9	133K	0.8K	0.9%	21.9%	23.1%	30.0%	23.6%	0.5%
Wicket	8.6.1	216K	0.4K	2.2%	39.3%	7.6%	28.5%	22.4%	0.0%
Zookeeper	3.5.6	97K	1.2K	2.2%	18.3%	19.3%	35.3%	24.9%	0.0%
Average	——	469K	2.0K	8.0%	25.5%	17.5%	24.5%	24.4%	0.1%

*3) **Privacy Concerns**:* LLM with sensitive data, such as code, raises serious privacy concerns, especially with commercial models like ChatGPT. For instance, Samsung recently banned the use of ChatGPT among employees due to a sensitive code leak, which highlighted concerns over data security and interactions with proprietary information [22]. To prevent these privacy concerns, our study utilizes open-sourced LLMs. By choosing open-source models for local deployment, we ensure data privacy and adherence to strict protection standards, allowing us to explore the effective integration of LLMs with in-house code while safeguarding sensitive information and maintaining data security.

By focusing on log level suggestion, our study explores under-examined areas in the application of LLMs, including challenges related to model fine-tuning, comparisons between traditional and LLM-based approaches, and privacy concerns.

III. STUDY DESIGN

A. Studied Dataset

Overview. We conduct the study on nine large-scale open-source Java systems (Table I). We chose these systems because they are actively maintained by the Apache Software Foundation, well-documented, and cover a variety of domains from database systems to search engines. They vary in size, with Lines of Code (LOC) ranging from 97K to 1.5M and Number of Logging Statements (NOL) ranging from 0.4K to 5.5K. These systems have also been widely used in prior research on logging [23], [24], [50], [51].

We analyzed the logging statements across all nine systems. Table I shows the log level distribution across these different systems. We observed that the primary use of logging statements is to highlight potential issues during system execution, with a notable distribution among the *debug*, *info*, *warn*, and *error* levels. For example, 25.5% of the logging statements belong to the *debug* level, 24.5% to the *warn* level, 24.4% to the *error* level, and 17.5% to the *info* level. However, we observed that *fatal* log levels are present in only one of the nine systems studied (i.e., Karaf), accounting for less than 1% of the total logging statements. As the *fatal* level is considered obsolete in modern logging frameworks due to its similarity to the *error* level, as stated in the SLF4J official documentation [52], we excluded it from our study. Moreover, the *trace* level has a lower occurrence: an average of 8.0% of logging statements. Notably, projects such as Flink, JMeter, Karaf, Wicket, and Zookeeper exhibit a negligible number of

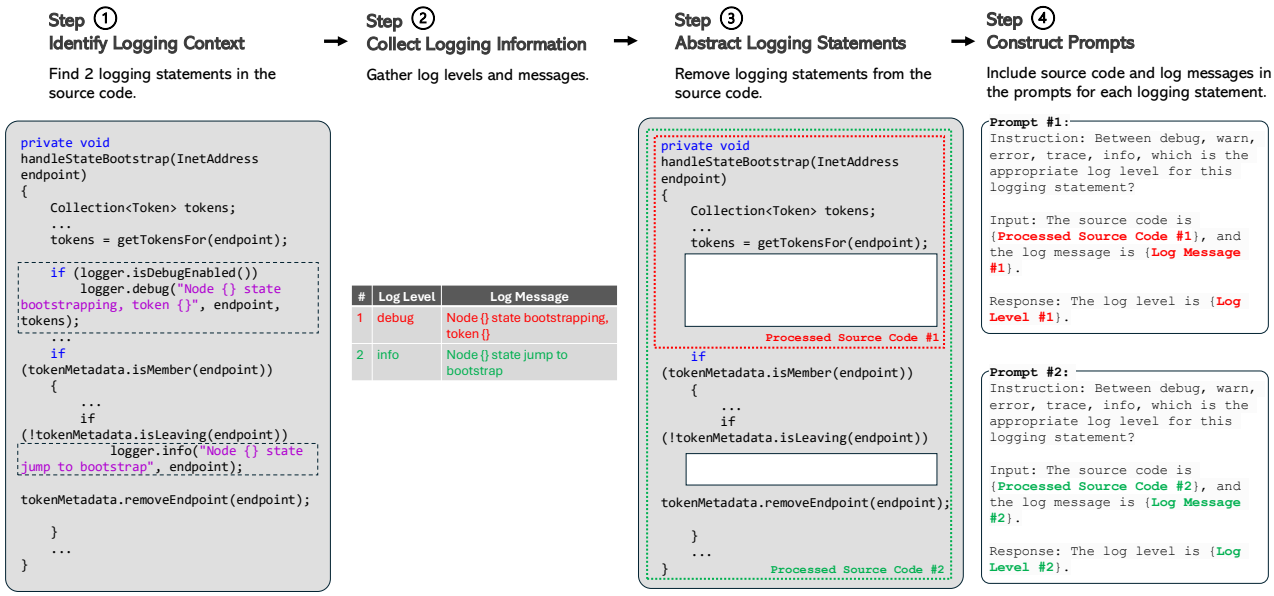

Fig. 1: Outline of the data processing stage: ① **Identify Logging Context:** Locate methods with logging statements, ② **Collect Logging Information:** Determine the log levels and messages, ③ **Abstract Logging Statements:** Remove logging statements and log guards from the source code, ④ **Construct Prompts:** Create prompts for the LLM using the gathered information.

trace-level logging statements, ranging from 0.7% to 2.2%. According to the SLF4J official documentation, the *trace* level shares a similar semantic meaning with the *debug* level, which may explain the lower occurrence of logging statements at the *trace* level. In summary, *debug*, *warn*, and *error* collectively constitute 75% of the logging statements.

The uneven distribution of log levels presents a significant challenge in achieving accurate log level suggestion. The disparity in frequency among different log levels means that some levels are more common than others, which implies that suggesting these more frequent levels might be simpler. However, this approach fails to account for the challenge of accurately identifying less frequent but still crucial log levels. Therefore, while it may appear that focusing on the most common log levels could improve suggestion accuracy, addressing the imbalance and ensuring accurate suggestions across all levels remains a complex and crucial task for effective log level suggestion.

Data Collection. Figure 1 illustrates the structure of our data preparation process for the log level suggestion. It consists of four key components: ① Identify Logging Context, ② Collect Logging Information, ③ Abstract Logging Statement, and ④ Construct Prompts.

① **Identify Logging Context.** We used static analysis to locate logging statements by traversing the abstract syntax tree to find method invocations using widely used logging libraries such as Log4j and SLF4J. This approach enables us to identify the context in which logging occurs by retrieving the calling method for each method containing a logging statement. Understanding the calling method is crucial because it helps us trace the source of the logging events, providing insight into the broader execution context and potentially identifying patterns or issues related to logging practices.

② **Collect Logging Information.** After identifying methods with logging statements, we extract and parse these statements to gather relevant information. Specifically, we collect (1) the log message and (2) the verbosity level. In our example, there are two logging statements with different log level identified in the method `handleStateBootstrap`. In total, we retrieved 17.6K logging statements from the nine systems.

③ **Abstract Logging Statements.** Our goal is to abstract logging statements by not only removing the logging statements but also excluding any code from basic blocks that follow the one containing each logging statement. For each logging statement, we extract the source code from the start of the method up to the end of the basic block where the logging statement is found. This approach ensures we consider only the relevant part of the code, as basic blocks provide a continuous sequence of statements without branching. As shown in Step 3 of Figure 1, Log Statement #2 is associated with a longer segment of code because it is located further down in the method, whereas Log Statement #1 is in a shorter block.

In addition to removing logging statements, we also eliminate log guards, conditions that determine whether a log message should be processed based on current logging settings. We exclude all AST nodes related to log guards before sending the code context to LLMs, a practice that prevents data leakage and aligns with previous work [23], [24].

④ **Construct Prompts.** Prior studies [23], [24] have investigated various feature combinations for suggesting log levels. For instance, DeepLV [23] combines the syntactic context of

TABLE II: Language models used in our study.

Task Type	Pre-training Objective	Model Name	Parameter Size
Fill-Mask	NLP-based	BERT base	110M
		BERT large	336M
		RoBERTa base	125M
		RoBERTa large	355M
	Code-based	CodeBERTa	84M
		CodeBERT	125M
		codebert-java	125M
		GraphCodeBERT	125M
Text Generation	NLP-based	LLaMA 2 7B	7B
		LLaMA 2 13B	13B
	Code-based	CodeLlama 7B	7B
		CodeLlama 13B	13B

the logging statement with the log message content, while TeLL [24] integrates multi-level block information with log messages. Inspired by these methods, our study leverages the strengths of combining diverse information sources. As depicted in Step 4 of Figure 1, we have developed a prompt template that incorporates two key features for input into the LLMs: (1) Processed Source Code and (2) Log Message.

B. Selecting Large Language Models (LLMs)

We selected 12 open-source Large Language Models, as enumerated in Table II, for our experimental purposes. These models encompassing both NLP-based and Code-based models, vary in size, ranging from 84 million to 13 billion parameters, and will all be executed using identical datasets in III-A to examine the influence of both model size and type on performance, eliminating the potential confounding factor of using diverse training or testing data.

Fill-Mask Models. NLP-based Language Models like BERT base, BERT large, RoBERTa base, and RoBERTa large are chosen for examination to assess whether programming statements can effectively extract semantic information. Specifically, these Language Models undergo pretraining using the Masked Language Modeling (MLM) objective. This involves randomly masking 15% of the words in a sentence, subsequently processing the entire masked sentence through the model, and predicting the masked word [39]. For code-based models, we opted for CodeBERT and CodeBERTa as they are pre-trained on data from various programming languages. We also consider, codebert-java, a specialized CodeBERTa model trained exclusively on Java code, shares the identical model architecture and size (125M parameters) due to their common foundation in the masked-language-modeling task [53].

Text Generation Models. We selected LLaMA 2 models as they provide a benchmark for evaluating large-scale, general-purpose language understanding and generation [38]. While the CodeLlama models, specifically fine-tuned from LLaMA 2 models for coding tasks, offer insights into how specialized training affects performance in code-related applications. For both LLaMA 2 and CodeLlama, we selected 7B and 13B versions from both LLMs to compare and assess the impact of parameter size on accuracy in log level suggestion and investigate how these models perform under In-Context Learning and fine-tuning scenarios.

C. Sampling Few Shot Data

Prior studies [54], [36] have shown that using diverse training data improves the performance and generalization of deep learning models. Based on this, we select few shot samples from the target system to ensure we cover each available log levels. For example, for a 5-shot sample, we randomly select one logging sample from each level: *debug*, *warn*, *error*, *trace*, and *info*. For a 30-shot sample, we choose six logging samples from each log level. Random selection helps to avoid bias and ensures that our samples represent a wide range of scenarios, ensuring the diversity of the samples.

To explore how the number of shots affects LLM performance, we selected samples with 5, 10, 20, and 30 shots. These samples were then utilized in two different learning paradigms:

In-Context Learning. Given the token limitations of the prompt, we concentrate on using 5-shot samples for In-Context Learning (ICL). In our experiment, each prompt includes five samples, with each sample containing the processed source code and log message as input, and the corresponding log level of a logging statement as the output. The goal is to assess if the LLM can accurately match log levels with functions and logging statements. We also evaluate ICL's effectiveness using a 0-shot prompt to determine how well the model can understand instructions without any prior examples.

Fine-tuning. To compare the performance of In-Context Learning and fine-tuning, we use the 5-shot samples in ICL prompts for fine-tuning LLMs. This comparison evaluates differences in model performance between these approaches: adapting to tasks with in-context examples versus updating the model's parameters through fine-tuning. Our analysis aims to determine which method provides superior accuracy for log level suggestion. Additionally, to assess the impact of the number of shots on LLM performance, we fine-tuned the models using 10, 20, and 30 logging samples, following the prompt template illustrated in Step 4 of Figure 1. To ensure a fair comparison with prior works [23], [24], we follow these studies by also fine-tuning the LLMs using 60% of the data as a training dataset. We apply stratified random sampling [55] to divide the dataset into 60% of the input data for training, 20% for validation, and 20% for testing. This approach maintains the same amount of data used in the training, as well as ensures distribution of log levels across all sampled datasets as in the original data.

D. Evaluation Metrics

A "verbalizer" is a method for converting abstract labels or tokens into specific, interpretable terms. For Fill-Mask LLMs, log levels are used as tokens for verbalizers to provide clear mappings for suggestion tasks, whereas Text Generation LLMs do not require explicit verbalizers, as these models can inherently identify and apply suitable verbalizations for log levels. Unlike Fill-Mask LLMs, which focus on completing or predicting specific words within a given context, Text

Generation LLMs are designed to produce coherent and contextually appropriate text sequences that extend beyond single words. To handle this, we use post-processing techniques to extract the relevant log level from their outputs, which aligning with methods used in related studies [56]. Following prior studies [23], [24], we use Accuracy, Area Under the Curve (AUC), and Average Ordinal Distance Score (AOD), to assess the performance of LLMs in suggesting log levels.

Accuracy. The accuracy metric, widely used in previous multi-class classification studies [57], [23], [29], [56], [24], measures the proportion of correctly suggested log levels relative to the total. Accuracy is defined as the percentage of correct suggestions from the suggestion process. Higher accuracy reflects a model's proficiency in recommending log levels for more logging statements.

Area Under the Curve (AUC). The AUC (Area Under the Curve) measures a model's ability to discriminate between classes using the ROC (Receiver Operating Characteristic) curve, which plots the true positive rate against the false positive rate. AUC values range from 0 to 1, with higher values indicating better discrimination. An AUC below 0.5 suggests random performance. This study adopts the multiclass AUC definition by Hand et al.[58], following prior work[26], [23], [24]. In both previous and current research, this metric demonstrates a model's ability to distinguish log levels.

Average Ordinal Distance Score (AOD). It assesses the proximity between the actual log level and the suggested log level for each logging statement [23]. Each log level is assigned a numerical value, and the AOD is computed using the following formula:

$$AOD = \frac{\sum_{i=1}^{N} \left(1 - \frac{Dis(a_i, s_i)}{MaxDis(a_i)}\right)}{N} \quad (1)$$

where N represents the total number of suggestions. For each logging statement and its suggested log level, *Dis(a, s)* is the distance between the *actual* log level a_i and the *suggested* log level s_i (e.g., the distance between *error* and *info* is 2). The maximum possible distance of the actual log level ai is denoted by *MaxDis(a)*. The resulting AOD value ranges from 0 to 1, with a higher value indicating that the suggested log level is closer to the actual log level.

Environment and Implementation. Our experiments were conducted on a server with an NVIDIA Tesla V100 GPU using CUDA 12.2.2. We use the OpenPrompt framework [59] to fine-tune Fill-Mask LLMs by masking words in sentences and suggesting appropriate log levels. For Text Generation models, we fine-tune using LoRA [49], applying a maximum learning rate of 5e-4, the AdamW [60] optimizer, and a linear learning rate decay schedule. LLMs exhibit inherent randomness during the inference process. To ensure consistent output for the same input, we set the temperature parameter to 0.

IV. Results

In this section, we present our study results by answering the research questions (RQs).

A. RQ1: What is the accuracy of LLMs in suggesting log level?

Motivation. While LLMs have been used in prior studies to generate log statements [29], [31], [56], there is limited research on comprehensively suggesting log levels across diverse LLMs. Hence, we investigate (1) whether LLMs trained with different objectives (Fill-Mask and Text Generation Task) and different datasets (code snippets vs. natural language) vary in their effectiveness at suggesting log levels and (2) whether in-context learning or fine-tuning performs better in log-level suggestions. In particular, we answer the following three RQs to benchmark the capability of LLMs in log level suggestion:

RQ1-A: What is the effectiveness of LLMs in log level suggestion?

RQ1-B: How does fine-tuning or in-context learning impact LLMs?

RQ1-C: How does the performance of LLMs in log level suggestion compare to existing state-of-the-art?

1) RQ1-A: What is the effectiveness of LLMs in log level suggestion:

Results. ***For log level suggestion tasks, while larger LLMs generally show better accuracy, smaller code-specific models like CodeLlama 7B can achieve comparable performance to their larger counterparts with lower resource consumption, making them a more efficient option.*** Table III shows the performance of log level suggestion across diverse fill-mask and Text Generation LLMs. We consistently observe that larger variant of LLMs (RoBERTa large, LLaMA 2 13B) which are trained with more parameters outperform their base counterparts. For instance, the accuracy of roberta-large surpasses that of roberta-base by 6.95%. Similarly, both 13B versions of LLaMA 2 and CodeLlama exhibit slight but discernible improvements over their 7B counterparts, with 2.06% and 0.88% higher accuracy, respectively. However, the improvement in accuracy achieved by using larger variants of LLMs is not as substantial as the improvement gained from LLMs that have been trained with code. Interestingly, CodeLlama 7B performs comparably to CodeLlama 13B, despite having fewer parameters. Hence, for log level suggestion, the 7B model may be sufficient to achieve high performance with less resource consumption.

Choosing an LLM trained on relevant data is more crucial than selecting one trained with a larger number of parameters. Code-based LLM achieves superior performance (20% to 40% higher) compared to NLP-based LLMs, despite having smaller parameters. As expected, for both in-context learning and fine-tuning, LLMs trained on code achieve the highest accuracy, AUC, and AOD in both Fill-Mask and Text Generation tasks. Among fill-mask LLMs, the highest accuracy is achieved by GraphCodeBERT, while among text generation models, CodeLlama 7B leads. For instance, when comparing the accuracies of GraphCodeBERT and CodeLlama 7B, both achieve around 40%, whereas NLP-based LLMs like BERT and Llama2 achieve slightly more than 20% accuracy.

TABLE III: Comparison of accuracy, AUC, and AOD of LLMs across in-context learning and fine-tuning with varying few-shot sizes. The highest value for Fill-Mask LLMs is denoted in **blue**, and the highest value for Text Generation LLMs is in **red**.

LLMs	In-Context Learning						Fine-tuning														
	0 shots			5 shots			5 shots			10 shots			20 shots			30 shots			60% shots		
	Acc.	AUC	AOD	Acc.	AUC	AOD	Acc.	AUC	AOD	Acc.	AUC	AOD	Acc.	AUC	AOD	Acc.	AUC	AOD	Acc.	AUC	AOD
Fill-Mask																					
BERT base	18.67	53.12	57.65	22.58	57.94	57.58	29.30	59.69	54.73	31.43	60.78	54.16	35.64	64.00	61.06	38.38	65.51	63.19	60.77	81.08	79.03
BERT large	18.79	53.37	58.24	21.84	57.97	62.13	31.47	62.00	58.75	36.03	64.30	59.81	39.54	67.03	65.23	38.91	66.84	65.07	61.4	81.48	79.86
RoBERTa base	18.47	53.10	57.83	19.93	54.32	53.55	29.64	59.50	57.81	35.01	64.31	62.32	38.62	65.99	63.9	41.91	68.77	67.36	62.74	82.23	80.18
RoBERTa large	**27.40**	54.10	57.35	21.50	52.58	50.01	30.57	60.41	**61.03**	36.31	64.84	63.67	40.58	65.97	65.48	45.76	70.64	70.26	63.90	82.78	81.31
CodeBERTa	23.12	56.09	59.21	27.93	61.32	63.50	28.93	59.34	56.30	29.98	59.36	55.96	34.53	61.80	59.34	38.05	64.51	62.69	60.73	80.54	78.38
CodeBERT	23.07	**57.03**	68.65	**31.76**	**65.20**	68.83	31.55	62.86	56.63	36.75	66.76	62.20	42.47	71.24	69.33	46.46	72.78	71.02	63.88	82.88	81.08
codebert-java	23.41	56.66	68.98	26.46	58.41	**69.50**	**35.48**	64.52	58.75	38.57	65.98	61.48	45.04	71.22	69.86	47.16	72.49	71.54	63.42	82.16	80.60
GraphCodeBERT	22.56	55.99	68.71	30.01	66.48	68.02	34.07	**65.00**	58.55	**40.41**	**69.51**	**66.53**	**45.60**	**72.50**	**71.36**	**48.61**	**74.02**	**71.76**	**64.81**	**83.20**	**81.28**
Text Generation																					
LLaMA 2 7B	29.60	60.18	66.45	27.73	57.88	64.7	27.77	62.37	58.52	29.39	59.73	57.18	33.5	66.42	68.2	34.51	64.61	68.28	38.14	60.9	66.51
LLaMA 2 13B	31.56	64.44	70.85	24.99	56.45	67.35	27.98	61.16	58.27	33.85	66.54	63.74	37.82	69.44	68.42	41.22	71.55	71.84	55.13	78.66	79.83
CodeLlama 7B	34.22	64.16	72.09	**44.83**	**75.53**	**78.87**	33.69	64.64	66.93	37.68	68.87	65.19	42.37	73.09	72.69	42.75	73.21	70.79	**58.33**	**80.45**	**81.56**
CodeLlama 13B	**42.40**	**74.01**	**76.22**	39.58	69.98	69.98	**41.92**	**73.73**	**74.78**	**41.53**	**72.45**	**70.77**	**44.00**	**75.41**	**74.89**	**43.78**	**75.06**	**75.18**	45.53	75.75	77.31

TABLE IV: Comparison between our LLM-based log level suggestion: FM (Fill-Mask LLM) and TG (Text Generation LLM), and two state-of-the-arts deep learning-based techniques: DeepLV [23] and TeLL [24].

Project	Accuracy				AUC				AOD			
	DeepLV	TeLL	FM	TG	DeepLV	TeLL	FM	TG	DeepLV	TeLL	FM	TG
Cassandra	60.6	63.5	62.2	41.7	84.2	88.4	80.0	66.4	80.5	81.2	78.4	73.1
ElasticSearch	57.7	70.3	55.8	63.9	81.3	90.5	77.1	82.9	80.2	84.1	77.4	82.9
Flink	65.2	72.9	72.0	75.2	85.1	92.5	87.1	88.6	83.8	86.3	85.2	87.3
Hbase	60.3	70.7	64.5	67.3	84.2	92.1	83.5	85.3	81.7	87.3	80.4	82.7
Jmeter	62.3	73.7	69.5	70.2	83.9	92.1	85.1	86.0	80.9	87.2	83.7	86.7
Kafka	51.8	64.2	60.6	53.5	79.5	88.8	81.8	81.8	77.5	81.2	80.9	79.8
Karaf	67.2	75.0	67.9	58.0	85.6	90.8	85.6	86.5	81.6	86.7	83.6	84.6
Wicket	63.8	74.4	63.7	50.0	85.0	89.9	82.2	68.7	79.3	85.6	78.8	78.5
Zookeeper	60.9	74.6	67.0	45.0	84.8	92.4	86.5	77.8	82.0	88.7	83.0	78.4
Average	61.1	71.0	64.8	58.3	83.7	90.8	83.2	80.4	80.8	85.4	81.3	81.6

2) RQ1-B: How does In-Context Learning or fine-tuning impact LLMs:

Results. ***When using the same set of samples, 10 out of 12 LLMs perform better when these samples are used for fine-tuning rather than for prompt construction.*** Out of 8 Fill-Mask LLMs, 7 exhibited improved performance when the 5-shot samples were used for fine-tuning rather than for In-Context Learning. The exception was CodeBERT, which saw a slight accuracy decrease of 0.3%. The other Fill-Mask LLMs demonstrated performance gains, with RoBERTa large showing the most significant improvement, up to 9.7%. In contrast, among the Text Generation LLMs, only CodeLlama 7B experienced a minor decline in performance.

Text Generation LLMs demonstrate superior performance out of the box, but they are difficult to fine-tune effectively. Although all LLMs showed potential for improvement through fine-tuning, the rate of improvement varied between Fill-Mask and Text Generation LLMs when fine-tuning with 10, 20, and 30 shots. Off-the-shelf Text Generation LLMs, with accuracies ranging from 29.60% to 42.40%, outperform Fill-Mask LLMs, which have accuracies between 18.47% and 27.40%. However, increasing the data to 60% did not improve results for Text Generation LLMs. In contrast, Fill-Mask LLMs showed significant gains across all metrics, with accuracies nearly tripling.

3) RQ1-C: How does the performance of LLMs in log level suggestion compare to existing state-of-the-art:

Results. ***Given the log message and source code of the method provided, fine-tuned Fill-Mask LLM achieves competitive performance with state-of-the-art models, with a margin under 7%.*** As noted in RQ1-B, Text Generation LLMs performed poorly even after fine-tuning with 60% of the input data. Specifically, CodeLlama 7B achieved 58.3% accuracy, comparable to DeepLV models trained with log messages (61.1%). On the other hand, Fill-Mask LLMs benefited significantly from fine-tuning. Table IV showed that Fill-Mask LLMs surpass DeepLV in accuracy, AUC, and AOD metrics with GraphCodeBERT. Notably, accuracy improved by 3.7%, though it still lags behind TeLL by approximately 6.2%. This demonstrates that a fine-tuned Fill-Mask LLM achieves competitive performance with state-of-the-art models when provided with both source code and log messages.

Discussions. ***Text Generation LLMs, especially NLP-based LLMs, pose a high risk of producing invalid results, but this risk can be mitigated with the aid of fine-tuning.*** We have noticed that Text Generation LLMs often rephrase questions or repeat instructions, leading to invalid suggestions. This issue, called "hallucination" [61], [62], [63], refers to generating text that is nonsensical or strays from the original content. These hallucinations motivate us to investigate the frequency of unavailable results and to explore strategies for minimizing their occurrence.

Among Text Generation LLMs, the CodeLlama 13B model has the lowest hallucination rate, with only a 0.19% likelihood of producing an invalid output—about 2 out of 1,000 log level suggestions—despite being untrained and lacking example prompts. This shows that CodeLlama 13B performs the strongest in interpreting our prompts and generating appropriate responses.

Among all learning paradigms for CodeLlama 13B, fine-tuning with 5 shots has the highest probability of returning an invalid output at 2.62%, while other paradigms show less than 1%. Similarly, CodeLlama 7B reaches a 5.73% possibility of returning invalid results with fine-tuning and 5 shots. However, CodeLlama 7B has a 20.1% chance of returning invalid input when untrained and without examples in the prompt, opposite to CodeLlama 13B. Despite this, CodeLlama 7B and 13B have accuracies of 34.22% and 42.40%, respectively, meaning that

although 1 in 5 suggested results are invalid, CodeLlama 7B achieves higher accuracy in log level suggestion.

For LLaMA 2 variants, off-the-shelf LLaMA 7B has similar chances of returning invalid data regardless of samples provided in the prompt: ICL 0 shots have a 25.41% chance, and ICL 5 shots have 24.30%. While still higher than Code LLaMA variants, fine-tuned LLaMA 2 models are less likely to return invalid results. However, LLaMA 2 13B is most prone to confusion with long prompts, returning invalid results 49.8% of the time.

These observations highlight the effectiveness of LLMs in varying scenarios and emphasize the importance of selecting the right model for task requirements. Enhanced training and fine-tuning strategies can reduce invalid output rates, enhancing reliability in real-world applications.

Insufficient training data may lead to monotonous log level predictions in Fill-Mask LLMs. We observed that under the ICL 0 shots paradigm, Fill-Mask LLMs predominantly suggest either "*Info*" or "*Error*" log levels. As the amount of training data increases, these models gradually start incorporating a broader range of log levels in their suggestions. This trend underscores the impact of additional training data on enhancing the accuracy and flexibility of LLMs, particularly in adapting to a broader spectrum of log levels. This finding is crucial for improving LLM robustness and applicability in real-world scenarios requiring diverse log level suggestions.

> Fill-Mask models such as GraphCodeBERT, when fine-tuned, can outperform larger models in log level suggestions. Fine-tuning using task-specific data leads to substantial improvements in accuracy and makes these models competitive with state-of-the-art methods.

B. RQ2: How effective is including additional context in log level suggestion?

Motivation. Prior studies [23], [24] found that combining two features improves log level suggestion performance. DeepLV [23] achieved 42.0% accuracy with log messages alone, 54.0% with syntactic context, and 61.1% with both. TeLL [24] reached 71.0% using log messages and multi-level block information, compared to 67.8% with just the latter. In our comparison of LLMs and prior works, we initially included both source code and log messages. However, since TeLL [24] credited intra-block features for their success, we are considering additional contexts to improve accuracy.

Approach. The prompt string is tweaked to include the calling method as seen in the sample below.

```
### Instruction: Between debug, warn, error, trace, info, which is the appropriate log level for this logging statement?
### Input: The previous method is z, the source code is x, and the log message is y.
### Response: The log level is z.
```

Results. ***Overloading context negatively impacts LLM performance across learning paradigms.*** Table V showed that including additional context will have a detrimental effect on the performance of LLMs in suggesting log level. Both Fill-Mask and Text Generation LLMs showed a slight decline in accuracy for each of the learning paradigms, especially in In-Context Learning 5 shots, where the results plummeted 15.97% whereas other settings observed a decrease between 2.77% to 3.07%. This outcome conforms to the results in the work of He et al. [64] and Shi et al. [65], where incorporating more contexts cannot always guarantee a better result.

Short prompts decrease the risk of invalid outputs in log level suggestions using Text Generation LLMs.

We examined how including additional context affects the frequency of invalid outputs compared to cases without it. We found that longer prompts, including calling methods, led to a higher rate of invalid outputs. This finding is consistent with the research by Schäfer et al. [66] and Liu et al. [67], who noted that models struggle to robustly access and use information in long input contexts. In our case, incorporating details such as method call code can confuse the model, leading to invalid results and reduced accuracy. Our study highlights the advantages of concise prompts in improving Text Generation LLMs' effectiveness for log level suggestions.

Discussions. ***Concise Prompts for Enhancing LLM Performance in Log Level Suggestions.*** These findings suggest that overloading LLMs with excessive information can hinder their performance. By emphasizing concise prompts that focus on essential elements, such as log messages and source code, developers can improve the quality of the model's outputs. Our study highlights how prompt design affects LLM effectiveness and encourages future research on lightweight contextual features that are more focused. This could boost LLM performance without complicating input, resulting in more reliable software engineering applications.

> Including additional context reduces the accuracy of log level suggestions and increases the likelihood of invalid outputs. This suggests that concise prompts focused on relevant log messages and code are more effective, while overloading the model with undistilled context can negatively impact its performance.

C. RQ3: How is the generalizability of LLMs in suggesting log levels?

Motivation. Newly developed software systems may lack sufficient logging statements and source code to fine-tune an LLM adequately. In such cases, fine-tuning the LLM with a limited dataset can lead to suboptimal results. Therefore, leveraging data from other projects to complement the existing dataset for fine-tuning the LLM becomes essential. Such transfer learning techniques are commonly employed to overcome the challenge of limited datasets [68]. Previous studies have produced mixed findings on generalizability. DeepLV [23] reported less than a 1% increase in all metrics from enlarging or combining datasets, while other studies [24], [27] reported different outcomes. This discrepancy motivates us to investigate whether LLMs exhibit varying degrees of transferability when subjected to different dataset and combinations.

TABLE V: Comparison of Fill-Mask Language Model (GraphCodeBERT) and Text Generation Language Model (CodeLlama 7B) with and without additional context.

Settings	Fill-Mask						Text Generation					
	Acc.		AUC		AOD		Acc.		AUC		AOD	
	W/O	With	W/O	With	W/O	With	W/O	With	W/O	With	W/O	With
ICL 0 shots	38.80	38.88 (+0.08)	70.39	68.18 (-2.21)	68.39	67.20 (+0.21)	40.38	37.61 (-2.77)	67.59	65.47 (-2.12)	71.34	70.45 (-0.89)
ICL 5 shots	34.55	29.48 (-5.07)	69.38	60.45 (-8.93)	65.88	47.8 (+5.43)	48.19	32.22 (-15.97)	74.34	61.66 (-12.68)	75.72	67.83 (-7.89)
FT 5 shots	37.19	33.42 (-3.77)	69.94	66.31 (-3.63)	65.41	64.38 (-0.9)	40.42	37.48 (-2.94)	67.69	65.43 (-2.26)	71.35	70.43 (-0.92)
FT 10 shots	38.56	41.12 (+2.56)	67.87	69.84 (+1.97)	60.67	70.27 (-9.17)	40.3	37.32 (-2.98)	67.55	65.35 (-2.20)	71.26	70.37 (-0.89)
FT 20 shots	40.86	42.03 (+1.17)	71.68	70.72 (-0.96)	67.18	71.39 (-3.54)	40.39	37.33 (-3.06)	67.62	65.39 (-2.23)	71.27	70.34 (-0.93)
FT 30 shots	40.89	41.57 (+0.68)	71.85	71.08 (-0.77)	69.24	70.86 (-1.84)	40.41	37.34 (-3.07)	67.73	65.36 (-2.37)	71.41	70.40 (-1.01)
FT 60% shots	67.3	62.88 (-4.42)	83.43	81.90 (-1.53)	82.99	80.78 (+1.09)	40.43	37.61 (-2.82)	67.80	65.57 (-2.23)	71.44	70.53 (-0.91)

In this RQ, we study whether transfer learning among different studied systems is suitable for LLM with the following two sub-RQs:

RQ3-A: Can the performance improve by including training data from other studied systems?

RQ3-B: How accurate are LLMs in cross-system suggestions?

1) RQ3-A: Can the performance improve by including training data from other studied systems:

Approach. We expand the training dataset by consolidating all training data from the nine Java systems. Employing stratified sampling for each system, we partition the data into training (60%), validation (20%), and testing (20%) sets. Subsequently, we consolidate the training data from all systems and utilize it for fine-tuning the LLM. Meanwhile, the combined 20% validation dataset is employed to validate the model throughout the training phase. Finally, we utilized the fine-tuned LLM trained with the expanded dataset to the testing data of each system under study.

2) RQ3-B: How accurate are LLMs in cross-system suggestions:

Approach. For each target system, the training data from the remaining eight systems is consolidated and subjected to stratified sampling [55] with the 60%:20%:20% ratio. Subsequently, this training data is utilized to fine-tune the LLM and is tested on the remaining test data from the other eight systems.

Results. ***LLMs do not benefit from transfer learning, whether through enlarging or combining datasets.*** Table VI shows the results of enlarging the training set and combining the training set. We compared the performance of the best Fill-Mask and Text Generation LLMs identified in RQ1 with their within-system suggestions noted in RQ1, highlighting the differences in the values provided in parentheses. Overall, both cross and enlarged dataset training led to decreases in accuracy, AUC, and AOD for both LLM types. We noticed that Fill-Mask LLM performance decreased more significantly compared to the Text Generation LLM. The accuracy dropped 15.1% and 11.3% by Fill Mask LLM in cross and enlarged dataset training respectively, whereas Text Generation LLMs' accuracy reduced 14.0% and 8.2%. Despite Fill Mask LLM suffering more, the average values for all three metrics still surpassed the performance of Text Generation LLM.

Larger datasets and fine-tuning have limited impact on Text Generation LLM Performance for Log Level Suggestion.

Another observation is that the accuracy, AUC, and AOD of Text Generation LLMs are less affected by enlarging or combining datasets. This aligns with RQ1 findings, where increasing the number of sampling shots did not significantly improve Text Generation LLM performance.

Table VI shows an average decrease of 18.9% in Accuracy for Fill-Mask LLMs and 4.0% for Text Generation LLMs, indicating that fine-tuning LLMs does not benefit from the enlarged dataset. This finding aligns with previous studies [24], [27], where cross-system suggestions performed worse than within-system suggestions.

LLMs excel in suggesting log levels within the same system rather than across different systems, emphasizing the importance of specific data contexts in log level suggestion.

V. Implications and Future Works

Based on our empirical findings, we highlight actionable insights and future directions for two key audiences: ① developers, and ② researchers.

A. Developers

Approaches for Utilizing LLMs to Enhance Code Improvement Tasks. We identified key strategies for using LLMs to improve log level suggestions, which can also benefit other code enhancement tasks. While larger LLMs often provide better accuracy, smaller, code-specific models can achieve similar results with less resource use, making them more efficient. When resources are limited, Fill-Mask LLMs can also perform well despite their size. It's important to choose an LLM trained on relevant data for specific logging tasks, as this improves performance more than simply selecting larger models. Developers should focus on fine-tuning to enhance accuracy instead of relying on in-context learning. Although Text Generation LLMs show strong initial performance, they can be hard to fine-tune, so exploring other models or techniques may lead to better results. By following these strategies, developers can improve log level suggestions and support other code enhancement efforts.

B. Researchers

Enhancing LLMs through Information Slicing and Additional Code Features. In Section IV, we found that short prompts decrease the risk of invalid outputs in log level suggestions. Li et al. [69] emphasize the importance of dynamic variables in logging, noting that log variables are

TABLE VI: Comparison between Cross Dataset Training and Enlarged Dataset Training Fill-Mask (GraphCodeBERT) and Text Generation (CodeLlama 7B) LLMs in terms of Accuracy, AUC and AOD.

Project	Cross Dataset Training						Enlarged Dataset Training					
	Fill-Mask			Text Generation			Fill-Mask			Text Generation		
	Acc.	AUC	AOD	Acc.	AUC	AOD	Acc.	AUC	AOD	Acc.	AUC	AOD
Cassandra	46.7 (-18.7)	73.3 (-9.4)	72.2 (-9.7)	38.8 (-7.1)	70.9 (+2.0)	72.5 (-1.8)	55.5 (-9.8)	76.9 (-5.8)	76.4 (-5.5)	45.8 (-0.1)	75.3 (+6.4)	74.3 (+0.0)
ElasticSearch	39.0 (-29.0)	67.6 (-14.7)	68.5 (-14.0)	40.7 (+11.8)	69.2 (+8.4)	71.0 (+8.6)	42.5 (-25.5)	71.0 (-11.3)	72.6 (-9.9)	47.9 (+19.0)	74.5 (+13.7)	76.8 (+14.4)
Flink	52.8 (-19.2)	78.5 (-9.0)	73.9 (-11.9)	53.8 (+3.0)	78.0 (+3.6)	75.3 (-3.1)	56.2 (-15.8)	81.3 (-6.2)	76.9 (-8.8)	60.3 (+9.5)	81.2 (+6.8)	78.1 (-0.3)
Hbase	51.0 (-21.0)	77.6 (-8.8)	74.5 (-11.0)	48.7 (-3.4)	75.7 (+2.0)	74.4 (-3.4)	54.0 (-18.0)	78.6 (-7.8)	76.7 (-8.8)	55.6 (+3.5)	80.2 (+6.5)	79.0 (+1.2)
Jmeter	54.4 (-23.6)	79.1 (-10.4)	77.0 (-12.4)	53.7 (+7.3)	78.2 (+7.7)	77.5 (+1.8)	60.4 (-17.6)	81.2 (-8.3)	81.2 (-8.2)	61.3 (+14.9)	81.0 (+10.5)	80.8 (+5.1)
Kafka	47.7 (-18.1)	75.3 (-8.1)	72.6 (-10.5)	44.3 (+6.6)	74.1 (+8.7)	73.6 (+5.7)	46.9 (-18.9)	74.9 (-8.5)	71.7 (-11.4)	47.8 (+10.1)	77.7 (+12.3)	75.6 (+7.7)
Karaf	59.0 (-5.1)	83.3 (+0.1)	80.3 (-1.1)	46.4 (-0.3)	78.9 (+4.0)	79.0 (+0.7)	61.8 (-2.3)	83.7 (+0.5)	81.8 (+0.4)	51.9 (+5.2)	81.9 (+7.0)	80.7 (+2.4)
Wicket	48.4 (-21.1)	74.0 (-6.6)	71.6 (-11.0)	35.3 (-5.4)	69.8 (+2.7)	68.1 (-3.5)	52.4 (-17.1)	77.2 (-3.4)	74.9 (-7.7)	37.9 (-2.8)	70.8 (+3.7)	66.5 (-5.1)
Zookeeper	47.9 (-14.5)	79.6 (-3.1)	74.7 (-7.3)	37.0 (-12.2)	74.6 (+1.1)	73.9 (-2.3)	52.0 (-10.4)	81.3 (-1.4)	77.0 (-5.0)	42.1 (-7.1)	77.4 (+3.9)	76.6 (+0.4)
Average	49.7 (-18.9)	76.5 (-7.8)	73.9 (-9.9)	44.3 (+0.0)	74.4 (+4.5)	73.9 (+0.3)	53.5 (-15.1)	78.5 (-5.8)	76.6 (-7.2)	50.1 (+5.8)	77.8 (+7.9)	76.5 (+2.9)

Note: The +/- number after each data denotes the relative improvement or decline compared to the suggestions within the system as observed in RQ1.

typically shorter than calling method code. Additionally, Wang et al. [70] discovered that breaking information into slices improves LLM performance in generating test cases. By combining these findings, we can explore the potential of using shorter log variables alongside information slicing to enhance LLM effectiveness. Future research could utilize static code analysis to identify and incorporate additional code features, such as log variables, which may further improve performance.

Effective LLMs Are Not Just About Parameters. In our study, we assessed 12 open-source LLMs for log level suggestion and identified several performance patterns that provide insights for selecting the most effective models and optimizing their use. We found that larger parameters do not always perform better; instead, smaller, specialized models can be equally or more effective. Based on these insights, future research should focus on utilizing diverse training tasks, integrating dynamic variables, and employing information slicing techniques to further enhance LLM performance. Additionally, leveraging static code analysis to identify relevant code features may further improve model effectiveness.

Expanding the Scope of Training Tasks and Increasing Feature Diversity. To enhance the overall performance of LLMs in software engineering, it is essential to evaluate the incorporation of a diverse range of software development tasks beyond log level suggestion, including code enhancement, refactoring, and testing. Integrating these varied challenges into the training process allows LLMs to acquire a more profound understanding of coding practices and their complexities. This expanded training set will strengthen the models' robustness and versatility, enabling them to better meet the diverse needs of developers. Future research should prioritize this inclusion, ultimately leading to the development of more effective LLMs that can better support developers and software teams in real-world applications.

VI. Threats to Validity

Construct Validity. Our methodology assumes that the training data consists of high-quality source code that follows best logging practices. However, there is no universally accepted industrial standards for writing logging statements. To mitigate the issue, we selected large-scale, well-maintained systems of varying sizes across different domains that have been widely recognized by prior studies for their adherence to established logging standards [23], [24], [71]. We evaluate our models using the test datasets from each of these systems. It is important to note that different test datasets can yield diverse outcomes. To mitigate the impact of this variability, we employ stratified random sampling techniques, as utilized in prior studies [23], [51], [57], [55], to partition the dataset while maintaining the original dataset's distribution of labels.

Internal Validity. Randomness are observed during the inference process of LLMs. To mitigate this threat, we regulate the model temperature to 0, ensuring that LLMs consistently yield more consistent outputs for identical input text.

External Validity. The nine subjects of this study are open-source Java projects from the Apache Software Foundation. While Apache's coding style may limit the applicability of findings to other organizations, the study spans various domains, project sizes, and logging volumes for broader representativeness. Although evolving software practices and languages like C++ could affect the findings, our 0-shot experiments show minimal data leakage concerns, as low accuracy suggests the base model's unfamiliarity with these systems.

VII. Conclusion

In this study, we examined log level suggestion across nine Java systems using twelve open-source LLMs, focusing on leveraging readily available data like method source code and log messages. We found that LLM performance varies significantly by task and model type, with Code-based LLMs generally outperforming NLP-based LLMs for log level suggestion. Text Generation LLMs excelled with few-shot prompting, while Fill-Mask LLMs responded better to fine-tuning. Including the source code of calling methods decreased performance and increased invalid outputs. Our research highlights the importance of task-specific data and suggests that Text Generation LLMs are preferable when such data is unavailable. These findings offer valuable insights for enhancing LLMs in log level suggestion and guiding future research on refining LLMs for various code-related tasks.

REFERENCES

[1] T.-H. P. Chen, M. D. Syer, W. Shang, Z. M. J. Jiang, A. Hassan, M. N. Nasser, and P. Flora, "Analytics-driven load testing: An industrial experience report on load testing of large-scale systems," *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pp. 243–252, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:27051552

[2] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," *arXiv: Learning*, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:8393918

[3] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "Log20: Fully automated optimal placement of log printing statements under specified overhead threshold," *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:11263426

[4] Z. Ding, H. Li, and W. Shang, "Logentext: Automatically generating logging texts using neural machine translation," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 349–360.

[5] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:199458012

[6] A. R. Chen, T.-H. P. Chen, and S. Wang, "Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs," *IEEE Transactions on Software Engineering*, vol. 48, pp. 2905–2919, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:234296255

[7] S. Messaoudi, D. Shin, A. Panichella, D. Bianculli, and L. C. Briand, "Log-based slicing for system-level test cases," *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:235770332

[8] Z. Li, A. R. Chen, X. Hu, X. Xia, T.-H. Chen, and W. Shang, "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)*, 2023, pp. 129–140.

[9] Z. Li, "Towards providing automated supports to developers on writing logging statements," *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 198–201, 2020.

[10] Z. Ding, Y. Tang, Y. Li, H. Li, and W. Shang, "On the temporal relations between logging and code," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23, 2023, p. 843–854.

[11] D. Shin, Z. A. Khan, D. Bianculli, and L. C. Briand, "A theoretical framework for understanding the relationship between log parsing and anomaly detection," in *Runtime Verification*, 2021.

[12] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J.-G. Lou, M. Chintalapati, S. Furao, and D. Zhang, "Robust log-based anomaly detection on unstable log data," *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:191140040

[13] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," 2019. [Online]. Available: https://arxiv.org/abs/1811.03509

[14] S. Ma, J. Zhai, Y. Kwon, K. H. Lee, X. Zhang, G. Ciocarlie, A. Gehani, V. Yegneswaran, D. Xu, and S. Jha, "Kernel-supported cost-effective audit logging for causality tracking," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18. USA: USENIX Association, 2018, p. 241–253.

[15] Z. Li, T.-H. Chen, J. Yang, and W. Shang, "Studying duplicate logging statements and their relationships with code clones," *IEEE Transactions on Software Engineering*, pp. 2476–2494, 2021.

[16] H. Li, W. Shang, B. Adams, M. Sayagh, and A. E. Hassan, "A qualitative study of the benefits and costs of logging from developers' perspectives," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2858–2873, 2021.

[17] A. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Commun. ACM*, vol. 55, no. 2, p. 55–61, feb 2012. [Online]. Available: https://doi.org/10.1145/2076450.2076466

[18] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: error diagnosis by connecting clues from run-time logs," in *ASPLOS XV*, 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:1737092

[19] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," *2012 34th International Conference on Software Engineering (ICSE)*, pp. 102–112, 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:14211994

[20] "Responsible use of GitHub Copilot Chat in your IDE - GitHub Docs — docs.github.com," https://docs.github.com/en/copilot/responsible-use-of-github-copilot-features/responsible-use-of-github-copilot-chat-in-your-ide#use-cases-for-github-copilot-chat, [Accessed 28-09-2024].

[21] S. Ray, "Apple Joins A Growing List Of Companies Cracking Down On Use Of ChatGPT By Staffers—Here's Why — forbes.com," https://www.forbes.com/sites/siladityaray/2023/05/19/apple-joins-a-growing-list-of-companies-cracking-down-on-use-of-chatgpt-by-staffers-heres-why/, [Accessed 28-09-2024].

[22] ——, "Samsung Bans ChatGPT Among Employees After Sensitive Code Leak — forbes.com," https://shorturl.at/tcYr2, [Accessed 02-09-2024].

[23] Z. Li, H. Li, T.-H. Chen, and W. Shang, "Deeplv: Suggesting log levels using ordinal based neural networks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1461–1472.

[24] J. Liu, J. Zeng, X. Wang, K. Ji, and Z. Liang, "Tell: Log level suggestions via modeling multi-level code block information," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 27–38. [Online]. Available: https://doi.org/10.1145/3533767.3534379

[25] LogLevelLLM, "Loglevelllm repository," https://github.com/LogLevelLLM/LogLevelLLM, 10 2024, (Accessed on 10/02/2024).

[26] H. Li, W. Shang, and A. Hassan, "Which log level should developers choose for a new logging statement?" *Empirical Software Engineering*, vol. 22, pp. 1684 – 1716, 2016.

[27] Y. E. Ouatiti, M. Sayagh, N. Kerzazi, and A. E. Hassan, "An empirical study on log level prediction for multi-component systems," *IEEE Transactions on Software Engineering*, vol. 49, pp. 473–484, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:247137318

[28] Y. Li, Y. Huo, R. Zhong, Z. Jiang, J. Liu, J. Huang, J. Gu, P. He, and M. R. Lyu, "Go static: Contextualized logging statement generation," 2024.

[29] J. Xu, Z. Cui, Y. Zhao, X. Zhang, S. He, P. He, L. Li, Y. Kang, Q. Lin, Y. Dang, S. Rajmohan, and D. Zhang, "Unilog: Automatic logging via llm and in-context learning," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, apr 2024, pp. 129–140. [Online]. Available: https://doi.ieeecomputersociety.org/

[30] C. et al., "Evaluating large language models trained on code," *ArXiv*, vol. abs/2107.03374, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:235755472

[31] Y. Li, Y. Huo, Z. Jiang, R. Zhong, P. He, Y. Su, and M. R. Lyu, "Exploring the effectiveness of llms in automated logging generation: An empirical study," *ArXiv*, vol. abs/2307.05950, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:259837163

[32] J. Chen, X. Hu, Z. Li, C. Gao, X. Xia, and D. Lo, "Code search is all you need? improving code suggestions with code search," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 1–13.

[33] F. Lin, D. J. Kim, and T. Chen, "Soen-101: Code generation by emulating software process models using large language model agents," 2025.

[34] J. Chen, Z. Pan, X. Hu, Z. Li, G. Li, and X. Xia, "Reasoning runtime behavior of a program with llm: How far are we?" in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, 2025.

[35] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, and X. Yang, "Thinkrepair: Self-directed automated program repair," in *The 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024, pp. 1–13.

[36] Z. Ma, A. R. Chen, D. J. Kim, T.-H. Chen, and S. Wang, "Llmparser: An exploratory study on using large language models for log parsing," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, Apr. 2024.

[37] Z. Jiang, J. Liu, Z. Chen, Y. Li, J. Huang, Y. Huo, P. He, J. Gu, and M. R. Lyu, "Llmparser: A llm-based log parsing framework," *ArXiv*, vol. abs/2310.01796, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:269123283

[38] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023. [Online]. Available: https://arxiv.org/abs/2302.13971

[39] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019. [Online]. Available: https://arxiv.org/abs/1810.04805

[40] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," 2019. [Online]. Available: https://arxiv.org/abs/1907.11692

[41] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *ArXiv*, vol. abs/2001.08361, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:210861095

[42] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020.

[43] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," 2021. [Online]. Available: https://arxiv.org/abs/2009.08366

[44] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2024. [Online]. Available: https://arxiv.org/abs/2308.12950

[45] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[46] Y. Wang, Q. Yao, J. Kwok, and L. M. Ni, "Generalizing from a few examples: A survey on few-shot learning," 2020. [Online]. Available: https://arxiv.org/abs/1904.05046

[47] J. Dodge, G. Ilharco, R. Schwartz, A. Farhadi, H. Hajishirzi, and N. Smith, "Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping," 2020. [Online]. Available: https://arxiv.org/abs/2002.06305

[48] T. Gao, A. Fisch, and D. Chen, "Making pre-trained language models better few-shot learners," 2021. [Online]. Available: https://arxiv.org/abs/2012.15723

[49] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021. [Online]. Available: https://arxiv.org/abs/2106.09685

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

[51] Z. Li, T.-H. Chen, J. Yang, and W. Shang, "Dlfinder: Characterizing and detecting duplicate logging code smells," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 152–163.

[52] "SLF4J FAQ," Jun. 2023, [Online; accessed 18. Jun. 2023]. [Online]. Available: https://www.slf4j.org/faq.html

[53] S. Zhou, U. Alon, S. Agarwal, and G. Neubig, "Codebertscore: Evaluating code generation with pretrained models of code," 2023.

[54] Z. Gong, P. Zhong, and W. Hu, "Diversity in machine learning," *IEEE Access*, vol. 7, pp. 64 323–64 350, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:206491718

[55] H. Pirzadeh, S. Shanian, A. Hamou-Lhadj, and A. Mehrabian, "The concept of stratified sampling of execution traces," in *2011 IEEE 19th International Conference on Program Comprehension*, 2011, pp. 225–226.

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

[57] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Which variables should i log?" *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 2012–2031, 2021.

[58] D. J. Hand and R. J. Till, "A simple generalisation of the area under the roc curve for multiple class classification problems," *Machine Learning*, vol. 45, pp. 171–186, 2001. [Online]. Available: https://api.semanticscholar.org/CorpusID:43144161

[59] N. Ding, S. Hu, W. Zhao, Y. Chen, Z. Liu, H.-T. Zheng, and M. Sun, "Openprompt: An open-source framework for prompt-learning," *arXiv preprint arXiv:2111.01998*, 2021.

[60] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2019. [Online]. Available: https://arxiv.org/abs/1711.05101

[61] V. Rawte, A. Sheth, and A. Das, "A survey of hallucination in large foundation models," *arXiv preprint arXiv:2309.05922*, 2023.

[62] X. Shen, Z. Chen, M. Backes, and Y. Zhang, "In chatgpt we trust? measuring and characterizing the reliability of chatgpt," *arXiv preprint arXiv:2304.08979*, 2023.

[63] X. Yu, L. Liu, X. Hu, J. W. Keung, J. Liu, and X. Xia, "Fight fire with fire: How much can we trust chatgpt on source code-related tasks?" *arXiv preprint arXiv:2405.12641*, 2024.

[64] P. He, Z. Chen, S. He, and M. R. Lyu, "Characterizing the natural language descriptions in software logging statements," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 178–189. [Online]. Available: https://doi.org/10.1145/3238147.3238193

[65] F. Shi, X. Chen, K. Misra, N. Scales, D. Dohan, E. Chi, N. Schärli, and D. Zhou, "Large language models can be easily distracted by irrelevant context," 2023. [Online]. Available: https://arxiv.org/abs/2302.00093

[66] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," 2023. [Online]. Available: https://arxiv.org/abs/2302.06527

[67] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," 2023. [Online]. Available: https://arxiv.org/abs/2307.03172

[68] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" 2014. [Online]. Available: https://arxiv.org/abs/1411.1792

[69] Z. Li, C. Luo, T.-H. Chen, W. Shang, S. He, Q. Lin, and D. Zhang, "Did we miss something important? studying and exploring variable-aware log abstraction," 2023.

[70] Z. Wang, K. Liu, G. Li, and Z. Jin, "Hits: High-coverage llm-based unit test generation via method slicing," 2024. [Online]. Available: https://arxiv.org/abs/2408.11324

[71] H. Li, T.-H. P. Chen, W. Shang, and A. E. Hassan, "Studying software logging using topic models," *Empirical Software Engineering*, vol. 23, 10 2018.