# Exploring the Effectiveness of LLMs in Automated Logging Statement Generation: An Empirical Study

Yichen Li*†, Yintong Huo*†, Zhihan Jiang†, Renyi Zhong†, Pinjia He§, Yuxin Su‡,
Lionel C. Briand¶, and Michael R. Lyu†
†The Chinese University of Hong Kong, Hong Kong, China,
{ycli21, ythuo, zhjiang22, ryzhong22, lyu}@cse.cuhk.edu.hk
§The Chinese University of Hong Kong, Shenzhen, China, hepinjia@cuhk.edu.cn
‡School of Software Engineering, Sun Yat-sen University, Zhuhai, China, suyx35@mail.sysu.edu.cn
¶University of Ottawa, Canada, and the Lero SFI Centre for Software Research, University of Limerick, Ireland, lbriand@uottawa.ca

***Abstract*—Automated logging statement generation supports developers in documenting critical software runtime behavior. While substantial recent research has focused on retrieval-based and learning-based methods, results suggest they fail to provide appropriate logging statements in real-world complex software. Given the great success in natural language generation and programming language comprehension, large language models (LLMs) might help developers generate logging statements, but this has not yet been investigated.**

**To fill the gap, this paper performs the first study on exploring LLMs for logging statement generation. We first build a logging statement generation dataset, *LogBench*, with two parts: (1) *LogBench-O*: *3,870* methods with *6,849* logging statements collected from GitHub repositories, and (2) *LogBench-T*: the transformed unseen code from LogBench-O. Then, we leverage LogBench to evaluate the *effectiveness* and *generalization capabilities* (using *LogBench-T*) of 13 top-performing LLMs, from 60M to 405B parameters. In addition, we examine the performance of these LLMs against classical retrieval-based and machine learning-based logging methods from the era preceding LLMs. Specifically, we evaluate the logging effectiveness of LLMs by studying their ability to determine logging ingredients and the impact of prompts and external program information. We further evaluate LLM's logging generalization capabilities using unseen data (LogBench-T) derived from code transformation techniques.**

**While existing LLMs deliver decent predictions on logging levels and logging variables, our study indicates that they only achieve a maximum BLEU score of *0.249*, thus calling for improvements. The paper also highlights the importance of prompt constructions and external factors (e.g., programming contexts and code comments) for LLMs' logging performance. In addition, we observed that existing LLMs show a significant performance drop (*8.2%-16.2%* decrease) when dealing with logging unseen code, revealing their unsatisfactory generalization capabilities. Based on these findings, we identify five implications and provide practical advice for future logging research. Our empirical analysis discloses the limitations of current logging approaches while showcasing the potential of LLM-based logging tools, and provides actionable guidance for building more practical models.**

***Index Terms*—Logging practice, Large language model, Empirical study.**

## I. Introduction

WRITING appropriate logging statements in code is critical for documenting program runtime behavior, supporting various software development tasks. Effective logging statements can facilitate performance analysis [1], [2] and provide insights for failure identification [3], [4], [5], [6]. As shown in the example below, a logging statement typically consists of three *ingredients*: a logging level, logging variables, and logging texts [7]. Specifically, as illustrated in the example below, logging level (e.g., *warn*) indicates the severity of a log event; logging variables (e.g., *url*) contain essential run-time information from system states; and logging texts (e.g., *Failed to connect to host:* <>) provides a description of the system's activities.

```
log.warn("Failed to connect to host: {}", url)
```

To help software developers decide the contents of logging statements (i.e., *what-to-log*), logging statement generation tools are built to automatically suggest logging statements given code snippets. Conventional logging suggestion studies [8], [9] reveal that similar code tends to have similar logging statements, and thus, a retrieval-based approach is used to suggest similar logging statements from a historical code base [10]. However, such retrieval-based approaches are limited to the logging statements encountered in that code base. To overcome such limitation, recent studies employ neural-based methods to decide about *single ingredients* of logging statements (i.e., logging levels, logging variables, logging text). For example, prior work [11], [12] predicts the appropriate logging level by feeding surrounding code features to a neural network. While these tools have also shown improvements in suggesting important variables [13] or proper log levels [12], [14], they lack the ability to produce complete logging statements containing multiple ingredients simultaneously. Some tools [11] require the availability of certain ingredients to suggest others, which can be impractical for programmers who need to generate complete logging statements. However, the complete statement generation has been considered challenging as the model should analyze the code

* Co-first authors.

TABLE I
SUMMARIZATION OF KEY FINDINGS AND IMPLICATIONS IN THIS PAPER.

| Key findings | Key implications & Actionable advice |
|---|---|
| >> RQ1:<br>☞ The performance of existing LLMs in generating complete logging statements *needs to be improved* for practical logging usage.<br>☞ Comparing the LLMs' logging capabilities presents a challenge, as models perform inconsistently on different ingredients. | ➜ How to *generate proper logging text* warrants more exploration.<br>➜ Intriguing alternative, possibly *unified metrics* to assess the quality of logging statements. |
| >> RQ2 & RQ3:<br>☞ Directly applying LLMs yields *better performance* than conventional logging baselines.<br>☞ Instructions significantly impact LLMs, but there is consistency in the relative ranking of LLMs when used with same instructions.<br>☞ Demonstrations help, but more demonstrations does not always lead to a higher logging performance. | ➜ LLM-powered logging is promising. Refining prompts with instructions and demonstration selection strategies for effective few-shot learning should be investigated. |
| >> RQ4:<br>☞ Since comments provide code intentions from developers, ignoring them leads to decreased effectiveness for LLMs.<br>☞ Compared to comments, LLMs gain greater advantages from considering *additional methods* in the same file. | ➜ Providing proper *programming contexts* over the projects that reveal execution information can boost LLMs' logging performance. |
| >> RQ5:<br>☞ *Unseen code* significantly degrades all LLMs' performance, particularly in variable prediction and logging text generation. | ➜ To advance the generalization capabilities of LLMs, developing *prompt-based learning techniques* to capture code logic offers great potential of LLMs in automated logging. |

structure, comprehend the developer's intention, and produce meaningful logging text [15]. Moreover, existing neural-based tools are further restricted by training data with limited logging statements and may not generalize to unseen code.

Recent large pre-trained language models (LLMs) [16], [17] have achieved impressive performance in the field of natural language processing (NLP). Inspired by this, the latest logging-specific model, LANCE [15], treats logging statements generation as a text-to-text generation problem and trains a language model for it. LLMs have proven their efficacy in many code intelligence tasks, such as generating functional code [18], [19] or resolving bugs [20], and have even been integrated as plugins for developers [21] (e.g., Copilot [22], CodeWhisperer [23]). However, their capacity for generating complete logging statements has not been comprehensively examined. To fill this gap, we pose the following question: *To what extent can LLMs produce correct and complete logging statements for developers?* We expect LLMs, given their strong text generation abilities, can improve the quality of logging statements. Further, LLMs have exhibited a powerful aptitude for code comprehension [24], which paves the way for uncovering the semantics of logging variables.

**Our work.** To answer our research question, this empirical study thoroughly investigates how modern LLMs perform logging statement generation from two perspectives: *effectiveness* and *generalization capabilities*. We extensively evaluate and understand the effectiveness of LLMs by studying (1) their ability to generate logging ingredients, (2) the impact of input instructions and demonstrations, and (3) the influence of external program information. To assess the generalizability of LLMs, since LLMs are trained on a significant portion of publicly available code, there is a potential data leakage issue in which logging statements used for evaluation purposes may be included in the original training data [20], [25], [26]. It remains unclear whether LLMs are really inferring logging statements or merely memorizing the training data. Thus, we further evaluate the generalization capabilities of LLMs using unseen code.

In particular, we evaluate the performance of thirteen top-performing LLMs encompassing a variety of types – including five natural language and eight code-oriented models, covering both academic works and commercial coding tools on *LogBench-O*, a new dataset we collected, consisting of *2,430* Java files, *3,870* methods, and *6,849* logging statements. Additionally, we employ a lightweight code transformation technique to generate a semantics-equivalent modified dataset *LogBench-T*, which contains previously untrained data and thus can be used to evaluate the generalization capabilities of LLMs. Based on our large-scale empirical study on LogBench-O and LogBench-T, we summarize eight key findings and five implications with actionable advice in Table I.

**Contributions.** The contribution of this paper is threefold:

- We build a logging statement generation dataset, LogBench, containing the collection of *6,849* logging statements in *3,870* methods (LogBench-O), along with their functionally equivalent unseen code after transformation (LogBench-T).
- We analyze the logging effectiveness of 13 top-performing LLMs by investigating their performance over various logging ingredients, analyzing prompt information that influences their performance, and examining the generalization capabilities of these LLMs with unseen data. Additionally, we have reimplemented approaches from the pre-LLM era and compared them with current LLM-based approaches for the first time.
- We summarize our results into eight findings and draw five implications to provide valuable insights for future research on automated log statement generation. All

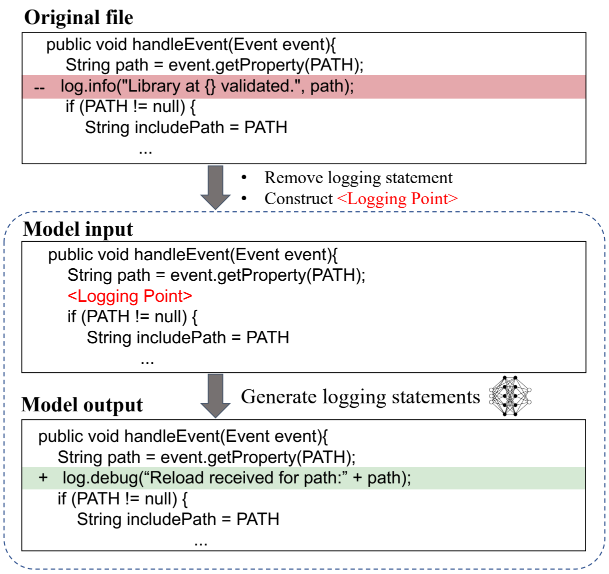



Fig. 1. Task formulation: given a method and a specific logging point, the model is asked to predict the logging statement in the point.

datasets, developed tools, source code, reimplemented baselines, and experiment results are available in a publicly accessible repository[1].

## II. Background

### A. Problem Definition

This study focuses on the *logging statement generation* task (i.e., *what-to-log*), which can be viewed as a statement completion problem: given lines of code (typically a method) and a specific logging point between two statements, the generator is then required to predict the logging statement at such point. The prediction is expected to be similar to the one removed from the original file. Figure 1 (in dashed line) illustrates an example of this task, where an effective logging statement generator should suggest `log.debug("Reload received for path:" + path)` that is highlighted with green for the specified logging point[2]. Following a previous study [15], for the code lines with *n* logging statements, we create *n-1* inputs by removing each of them one at a time.

In this paper, we do not address log location suggestions due to the varied experimental granularities observed in the literature [27], [28], [29], [30]. Additionally, evaluating line-level logging recommendations is problematic because it is difficult to determine whether logging at two distinct lines will affect logging results in program execution, particularly if the logging variables maintain unchanged values.

### B. Significance and Challenges in Logging Statement Generation

Logging is a prevalent practice in software development, essential for postmortem analysis [45], with a substantial amount of research emphasizing its quality and improvement [27], [9], [46], [47]. This is because logs are the only data to capture software runtime behavior [7].

Compared to general code generation, logging practice needs special investigation because of the following reasons. First, while code generation produces short methods with a high degree of functional similarity, logging statements are *non-functional* statements not discussed in code generation datasets (e.g., HumanEval [48], JavaScript datasets [49]). Therefore, their evaluation requires specific metrics distinct from those used in built-in test suites for code generation. Second, logging statements constitute only a minor fraction (e.g., 3,426 logging statements out of 1.7M lines of code in Hadoop [50]) of a project's total codebase; thus, randomly sampling statements from a project is unlikely to yield enough data to effectively assess logging practices. Consequently, evaluating the logging statement generation ability needs tailored benchmarks and measurements.

The composition of logging statements naturally makes the logging generation problem a joint task of code comprehension and text generation. Compared to code completion tasks, the generation of logging statements presents two distinct challenges: (1) inference of critical software runtime status and (2) the creation of complicated text that seamlessly integrates both natural language and code elements. Logging statements are indispensable in large-scale software repositories for documenting run-time system status. To log proper system status, a logging statement generator shall comprehend program structure (e.g., exception handling) and recognize critical code activities worthy of logging. Second, integrating natural language text and code variables poses a unique challenge. Logging statement generators must be mastered in two distinct languages and harmoniously aligned. Developers describe code functionalities in natural language and then incorporate relevant logging variables. Likewise, a logging statement generator should be capable of translating runtime code activities into natural language and explaining and recording specific variables.

### C. Study Subject

Motivated by the code-related text generation nature of the logging statement generation, we opt to investigate top-performing LLMs from three fields as our study subjects: LLMs designed for general natural text generation, LLMs tailored for logging activities, and LLMs for code intelligence. We also evaluate state-of-the-art logging suggestion models, which usually work on a single ingredient, to discuss whether advanced LLMs outperform conventional ones.

We summarize the details of 13 LLMs in Table II and three conventional approaches in Table III. Since we already included official models [43], [34], [32] from the GPT series, other models that have been tuned on GPT [51], [52] are not included in our study (e.g., GPT-Neo [51] and GPT-J [52]).

*1) General-purpose LLMs:* The GPT-series models are designed to produce natural language text closely resembling human language. The recent GPT models have demonstrated exceptional performance, dominating numerous natural language generation tasks, such as question-answering [53] and

[1]Available in: https://github.com/logpai/LogBench

[2]In this paper, the logging statement that the generator should predict is always highlighted by green.

TABLE II
STUDY SUBJECTS INVOLVED IN OUR EMPIRICAL STUDY.

| Model | Access | Description | Pre-trained corpus (Data size) | #Params | Year |
|---|---|---|---|---|---|
| **General-purpose LLMs** | | | | | |
| Davinci | API | Davinci is derived from InstructGPT [31] is an "instruct" model meant to generate texts with clear instructions. We access the Text-davinci-003 model by calling the official API from OpenAI. | - | 175B | 2022 |
| ChatGPT | API | ChatGPT is an enhanced version of GPT-3 models [32], with improved conversational abilities achieved through reinforcement learning from human feedback [33]. It forms the core of the ChatGPT system [34]. We access the GPT3.5-turbo model by calling the official API from OpenAI. | - | 175B | 2022 |
| GPT-4o | API | GPT-4o [35] is the latest version of GPT series models, with significantly enhanced contextual understanding and generation capabilities, achieved through extensive fine-tuning and optimization. We access the GPT-4o model by calling the official API from OpenAI. | - | - | 2024 |
| Llama2 | Model | Llama2 [36] is an open-sourced LLM trained on publicly available data and outperforms other open-source conversational models on most benchmarks. We deploy the Llama2-70B model provided by the authors. | Publicly available sources (2T tokens) | 70B | 2023 |
| Llama3.1 | Model | Llama3.1 [37] is currently the most powerful and the biggest open-sourced LLM trained on high quality public corpus. We deploy the largest version Llama3.1-405B model provided by the authors. | Publicly available sources (15.6T tokens) | 405B | 2024 |
| **Logging-specific LLMs** | | | | | |
| LANCE | Model | LANCE [15] accepts a method that needs one logging statement and outputs a proper logging statement in the right position in the code. It is built on the T5 model, which has been trained to inject proper logging statements. We re-implement it based on the replication package [38] provided by the authors. | Selected GitHub projects (6M methods) | 60M | 2022 |
| **Code-based LLMs** | | | | | |
| InCoder | Model | InCoder [18] is a unified generative model trained on vast code benchmarks where code regions have been randomly masked. It thus can infill arbitrary code with bidirectional code context for challenging code-related tasks. We deploy the InCoder-6.7B model provided by the authors. | GitHub, GitLab, StackOverflow (159GB code, 57GB StackOverflow) | 6.7B | 2022 |
| CodeGeeX | IDE Plugin | CodeGeeX [39] is an open-source code generation model, which has been trained on 23 programming languages and fine-tuned for code translation. We access the model via its plugin in VS Code. | GitHub code (158.7B tokens) | 13B | 2022 |
| StarCoder | Model | StarCoder [40] has been trained on 1 trillion tokens from 80+ programming languages, and fine-tuned on another 35B Python tokens. It outperforms every open LLM for code at the time of release. We deploy the StarCoder-15.5B model provided by the authors. | The Stack (1T tokens) | 15.5B | 2023 |
| CodeLlama | Model | CodeLlama [41] is a family of LLMs for code generation and infilling derived from Llama2. After they have been pretrained on 500B code tokens, they are all fine-tuned to handle long contexts. We deploy the CodeLlama-34B model provided by the authors. | Publicly available code (500B tokens) | 34B | 2023 |
| TabNine | IDE Plugin | TabNine [42] is an AI code assistant that can suggest the following lines of code. It can automatically complete code lines, generate entire functions, and produce code snippets from natural languages. We access the model via its plugin in VS Code. | - | - | 2022 |
| Copilot | IDE Plugin | Copilot [21] is a widely-studied AI-powered code generation tool relying on the CodeX [43]. It can extend existing code by generating subsequent code trunks based on natural language descriptions. We access the model via its plugin in VS Code. | - | - | 2021 |
| CodeWhisperer | IDE Plugin | CodeWhisperer [23], developed by Amazon, serves as a coding companion for software developers. It can generate code snippets or full functions in real time based on comments written by developers. We access the model via its plugin in VS Code. | - | - | 2022 |

text summarization [54]. Recently, Meta researchers built an open model, LLaMa, as a family member of LLMs [36], which showed more efficient and competitive results with GPT-series models. In our paper, we select the three most capable GPT-series models based on previous work [55], i.e., Davinci, ChatGPT, GPT-4o for evaluation. We also choose one competitive open-sourced model, Llama2, as the representative of general-purpose LLMs.

*2) Logging-specific LLMs:* To the best of our knowledge, LANCE [15] is the only work on training LLMs for automatically generating logging statements, which has been published in top-tier software venues (i.e., FSE, ICSE, ASE, ISSTA, TSE, and TOSEM). Consequently, we choose it as logging-specific LLMs.

*3) Code-based LLMs:* Inspired by the considerable success of LLMs in the natural language domain, researchers also derive code-based LLMs that can support code understanding and generation tasks, so as to assist developers in completing codes. These LLMs are either commercial models powered by companies, or open-access models in academia. For the open-access models with publicly available weights, we follow the selection of code models on recent comprehensive evaluation studies [41], [40], [56], and reserve the LLMs with larger sizes than 6B. The process leads to four LLMs as our

TABLE III
CONVENTIONAL LOGGING APPROACH FOR SINGLE INGREDIENT RECOMMENDATIONS.

| Ingredient | Model | Description | #Params | Venue | Year |
|---|---|---|---|---|---|
| Logging levels | DeepLV | DeepLV [11] leverages syntactic context and message features of the logging statements extracted from the source code to make suggestions on choosing log levels by feeding all the information into a deep learning model. We reimplement the model based on the replication package provided by the authors*. | 0.2M | ICSE | 2021 |
| Logging Variables | WhichVar | WhichVar [13] applies an RNN-based neural network with a self-attention mechanism to learn the representation of program tokens, then predicts whether each token should be logged through a binary classifier. We reimplement the model based on its paper due to missing code artifacts*. | 40M† | TSE | 2021 |
| Logging Text | LoGenText-Plus | LoGenText-Plus [44] generates the logging texts by neural machine translation models (NMT). It first extracts a syntactic template of the target logging text by code analysis, then feeds such templates and source code into Transformer-based NMT models. We reproduce the model based on the replication package provided by the authors. | 22M | TOSEM | 2023 |

† The number of parameters (40M) includes the embedding module of the model.
* All the baselines we have reimplemented has been organized in our artifacts.

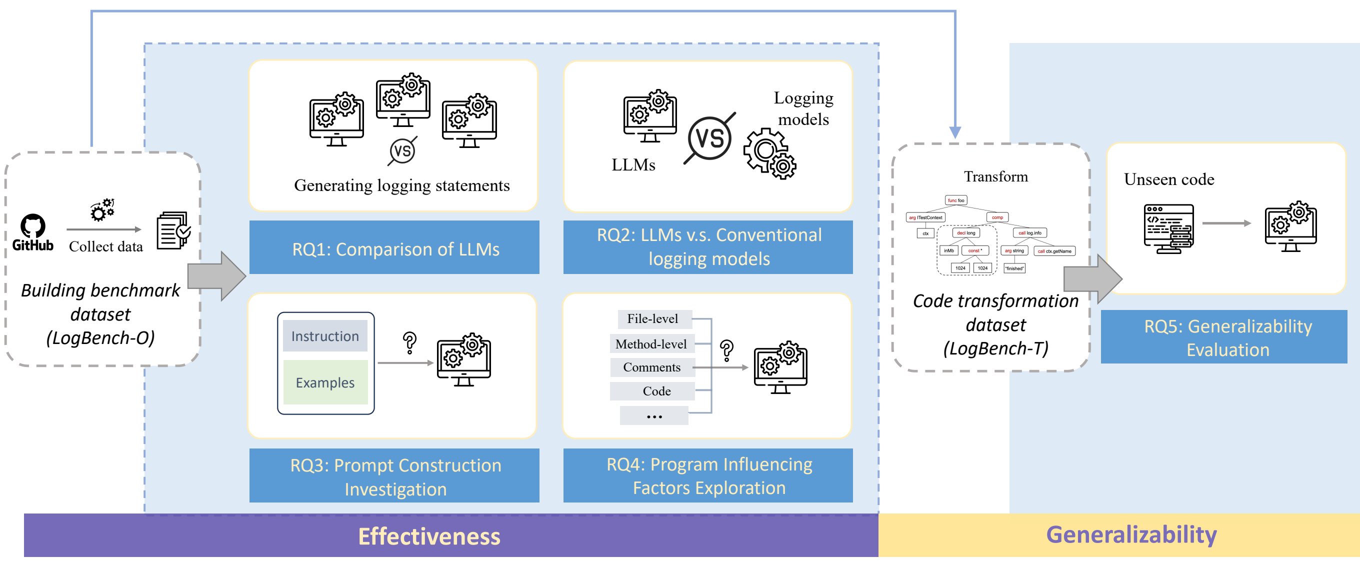



Fig. 2. The overall framework of this study involving five research questions.

subjects, i.e., InCoder [18], CodeGeex [39], StarCoder [40], and CodeLlama [41]. In terms of the commercial models, we select three popular developer tools as the study subjects, i.e., TabNine [42], Copilot [21], and CodeWhisperer [23] from Amazon.

*4) Conventional Logging Approaches:* Apart from LLMs that can offer complete logging statements, we also select conventional logging approaches that work on *single* logging ingredients for comparison. Specifically, for each ingredient, we choose the corresponding state-of-the-art logging approaches from the top-tier software venues: DeepLV [11] for log level prediction, Liu et al.'s [13] (denoted as WhichVar) for logging variable prediction, and LoGenText-Plus [44] for logging text generation. These approaches learn the relationships between specific logging ingredients and the corresponding code features based on deep learning techniques. Details are summarized in Table III.

## III. STUDY METHODOLOGY

### A. Overview

Fig. 2 depicts the overview framework of this study involving five research questions from two perspectives: (1) *effectiveness*: how do LLMs perform in logging practice? and (2) *generalizability*: how well do LLMs generate logging statements for unseen code?

To start, we develop a benchmark dataset LogBench-O comprising *6,849* logging statements in *3,870* methods by crawling high-quality GitHub repositories. Inspired by the success of LLMs in NLP and code intelligence tasks, our focus is on assessing their efficacy in helping developers with logging tasks. This study first evaluates the effectiveness of state-of-the-art LLMs in terms of multiple logging ingredients (RQ1). We then conduct a comparative analysis between state-of-the-art conventional logging tools and LLMs, elucidating differences and providing insights into potential future model directions

TABLE IV
OUR CODE TRANSFORMATION TOOLS WITH EIGHT CODE TRANSFORMERS, DESCRIPTIONS, AND ASSOCIATED EXAMPLES.

| Transformer | Descriptions | Example |
|---|---|---|
| Condition-Dup | Add logically neutral elements (e.g., *&& True* or \|\| *False*) | if (exp0) $\rightarrow$ if (exp0 \|\| false) |
| Condition-Swap | Swap the symmetrical elements of condition statements | if (var0 != null) $\rightarrow$ if (null != var0) |
| Local variable | Extract constant values and assign them to local variables | var0 = const0; $\rightarrow$ int var1 = const0; var0 = var1; |
| Assignment | Separate variable declaration and assignment | int var0 = var1; $\rightarrow$ int var0; var0 = var1; |
| Constant | Replace constant values with equivalent expressions | int var0 = const0 $\rightarrow$ int var0 = const0 + 0 |
| For-While | Convert *for-loops* to equivalent *while-loops* | for (var0 = 0; var0 < var1; var0++) {} $\leftrightarrow$ |
| While-For | Convert *while-loops* to equivalent *for-loops* | var0 = 0; while (var0++ < var1) {} |
| Parenthesis | Add redundant parentheses to expression | var0 = arithExpr0 $\rightarrow$ var0 = (arithExpr0) |

(RQ2). Next, we investigate the impact of instructions and demonstrations as inputs for LLMs, offering guidance for effectively prompting LLMs for logging (RQ3). Furthermore, we investigate how external influencing factors can enhance LLM performance, identifying effective program information that should be input into LLMs to improve logging outcomes (RQ4). Last but not least, we explore the generalizability of LLMs to assess their behavior in developing new and unseen software. To this end, we evaluate models on an unseen code dataset, LogBench-T, which contains code derived from LogBench-O that was transformed to preserve readability and semantics (RQ5).

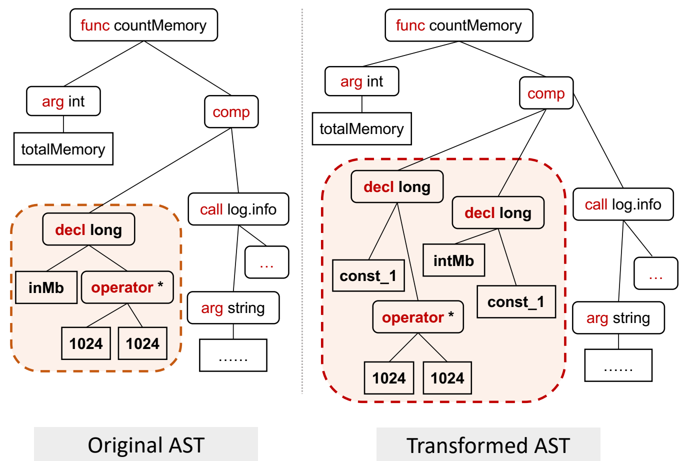



Fig. 3. An example of how the code (constant) transformer works. The constant checker firstly detects transformation points, then the Local variable transformer replaces the constant expression {`inMb=1024*1024`} by {`const_1=1024*1024; inMb=const_1`} involving a new variable *const_1*. The AST changes via transformation are highlighted in red area.

## *B. Benchmark Datasets*

Due to the lack of an existing dataset that can meet the benchmark requirements, we developed the benchmark dataset LogBench-O and LogBench-T for logging statement generation in this section. Although we chose Java as the target language of our study, due to its wide presence in industry and research [57], the experiments and findings can be extended to other programming languages.

*1) Creation of LogBench-O:* We build a benchmark dataset, consisting of high-quality and well-maintained Java files with logging statements, by mining open-source repositories from GitHub. As the largest host of source code in the world, GitHub contains a great number of repositories that reflect typical software development processes. In particular, we begin by downloading high-quality Java repositories that meet the following requirements[3]:

- Gaining more than *20* stars, which indicates a higher level of attention and interest in the project.
- Receiving more than *100* commits, which suggests the project is actively maintained and not likely to be disposable.
- Engaging with at least *5* contributors, which demonstrates the quality of its logging statements by simulating the collaborative software development environment.

We then extract the files that contain logging statements in two steps. We first select the projects whose POM file includes popular logging utility dependencies (e.g., Log4j, SLF4J), resulting in *3,089* repositories. We then extract the Java files containing at least one logging statement by matching them with regular expressions [58], because logging statements are always written in specified syntax (e.g., `log.info()`). Afterward, we randomly sample the collected files across various repositories, resulting in a dataset of *2,420* files containing *3,870* methods and *6,849* logging statements, which we refer to as LogBench-O.

*2) Creation of LogBench-T Dataset to Avoid Data Leakage:* LLMs deliver great performance in multiple tasks; however, evaluating their performance solely on publicly available data can be problematic. Since LLMs are trained on datasets that are obtained through large-scale web scraping [59], these models may have already seen the benchmark data during their training, raising concerns about assessing their generalization abilities [20], [25], [26]. This issue, commonly known as *data leakage*, requires particular attention since most code models [18] have been trained on public code.

To fairly evaluate the generalization ability of LLMs, we further develop an unseen code dataset LogBench-T that consists of the code transformed from LogBench-O for two

[3] All repositories were archived on July 2023

reasons. First, it is impractical to collect public data that is always excluded from any LLM training. With the increasing training of LLMs, using code from after the LLM release could still pose data leakage risks for newer models, making it an unreliable benchmark for evaluating emergent LLM-based methods and future logging techniques. Second, program transformation techniques have been also suggested to build unseen datasets in other code intelligence tasks [60]. For example, VJBench [61] and EvoEval benchmark [62] transform the current vulnerability fix dataset and code generation dataset to evaluate models' generalizability in unseen data, respectively.

Prior works have developed *semantics-preserving* code transformation techniques that do not change the functionality of the original code, for the purpose of evaluating the robustness of code models [63], [64], [65], [66]. However, these approaches randomly replace informative identifiers with meaningless ones, degrading the *readability* of the code. For example, after transforming an informative variable name (e.g., `totalMemory`) to a non-informative name (e.g., `var0`), even a programmer can hardly understand the variables and log properly. Such transformations make the transformed code less likely to appear in daily programming and not suitable for logging practice studies. To avoid this issue, we devise an AST-based code transformation tool that generates semantics-preserving and readability-preserving variations of the original code.

In particular, our code transformation tool employs eight carefully engineered, lightweight code transformers motivated by previous studies [63], [65], [67], [68], whose descriptions, together with their examples, are illustrated in Table IV. These code transformation rules work at the Abstract Syntax Tree (AST) level, ensuring that the transformed code remains semantically equivalent to the original code. Besides, readability-degrading transformations, such as injecting dead code [69] and modifying the identifier names, are eliminated. Additionally, to affirm the soundness of our transformations, we have limited our selection to widely-used transformation rules that have been proven effective in various code-related tasks [66], [63], [70] over time. Since our transformation techniques are based on AST, they would not change the functionality of the program code. Transformation rules are further verified by executing unit tests on sample projects, which confirms that our code transformations will not hurt functionality.

The process of transformation begins with converting the source code into an AST representation using JavaParser [71]. To detect potential transformation points (i.e., specific nodes and subtrees) for each transformer, a series of predefined checkers traverse the AST in a top-down manner. Once the transformation points are identified, each checker will independently call its corresponding transformer to perform a one-time transformation. We denote one-time transformation as $T : x \rightarrow x'$, where $x$ and $x'$ represent the source AST and the transformed AST, respectively. Each transformer functions independently, allowing multiple transformations to be applied to the same code snippet without conflicts. These single transformations are chained together to form the overall transformation: $\mathbb{T} = T_1 \circ T_2 \circ ... \circ T_n$. Once all the identified

TABLE V
THE STATISTICS AND SIMILARITY COMPARISON BETWEEN THE ORIGINAL CODE AND TRANSFORMED CODE.

| | LOC | Avg. #Variable | SL | ED | # Files | # Methods | # Logging statements |
|---|---|---|---|---|---|---|---|
| LogBench-O | 18.8 | 8.4 | 5.16 | 76.4 | 2,420 | 3,870 | 6,849 |
| LogBench-T | 25.3 | 10.1 | 5.19 | | 2,420 | 3,870 | 6,849 |

points have been transformed or the number of transformations reaches a predetermined threshold, the AST is converted back into the source code to complete the transformation process. Fig. 3 exhibits a case concerning how a `Local variable` transformer works.

## C. Dataset Summary

Table V reports the statistics of LogBench-O and LogBench-T, in terms of their number of files (# Files), methods (# Methods), and logging statements (# Logging statements). The table further presents a comparative analysis of code before and after transformation to understand the changes using various metrics. These metrics include Lines of Code in Method (LOC), Average Number of Variables in each method (Avg. # Variable), and String Length in the Logging Statements (SL). Additionally, the table compares method code similarity with metrics like Edit Distance (ED).

While our AST-based transformation approach ensures functionality equivalence, the discrepancy in LOC indicates significant alterations to the control flow and code structure of the target method. Their change in the Avg. # Variable metric demonstrates that our variable-related transformation strategies effectively modified the structure of variable assignments and uses. The minor change in SL reflects that the transformations are not applied to logging texts, as string-related transformations were not included. Furthermore, the high ED value of 76.4, alongside the average LOC of 18.8, evidences that the code has undergone extensive transformation, resulting in a significantly lower similarity. These transformed data can be considered as unseen data, providing a more accurate evaluation of the LLM's ability in logging for general scenarios.

## D. Implementations

*1) Evaluation Setting:* We evaluate all prompt-taken LLMs with the following settings: As we discussed in Sec. IV-D, we choose the median value of all metrics across the top five instructions, as determined by voting of all participated developers, to approximate the instructions most commonly utilized by developers. We set the temperature to 0 so that LLMs would generate the same output for the same query to ensure reproducibility. Based on the access ways of different LLMs (Table II), we evaluated them as follows.

(1) Released models (Llama2, LANCE, InCoder, StarCoder, CodeLlama, Llama3.1, GPT-4o): we ran them on a *32*-Core workstation with an Intel Xeon Platinum *8280* CPU, *256* GB RAM, and *4*x NVIDIA GeForce RTX *4090* GPUs in Ubuntu *20.04.4* LTS, using the default bit precision settings for each model. All LLM-based models share the same basic

(2) APIs (ChatGPT, Davinci, GPT-4o, Llama3.1): We called their official APIs to generate the logging statement. For ChatGPT, Davinci and GPT-4o, we use the public APIs provided by OpenAI with *gpt-3.5-turbo-0301*, *text-davinci-003* and *gpt-4o-2024-05-13*, respectively. For Llama3.1, we use the public APIs provided by DeepInfra [72] corresponding to the model *LLaMa-3.1-405b*.

(3) Plugins (Copilot, CodeGeeX, TabNine, CodeWhisperer): we purchased accounts for each author to obtain the logging statement manually at the logging point that starts with the original logging API (e.g., `log.`). This starting point forces these plugins to generate logging statements instead of other functional codes.

For conventional logging approaches, we reproduced them based on the replication packages released by the authors, or the paper descriptions if the replication package is missing. For all experiments that may introduce randomness, to avoid potential random bias, we repeat them three times and report the median results following previous works [73], [74], [75].

In experiments, we choose the median value of all metrics across the top five instructions, as determined by voting of all participated developers, to approximate the instructions most commonly utilized by developers. We set the temperature to 0 so that LLMs would generate the same output for the same query to ensure reproducibility.

*2) Code Transformation:* Our code transformation technique (Sec. III-B2) was implemented using *4,074* lines of Java code, coupled with the JavaParser library [71], a widely-used parser for analyzing, transforming, and generating Java code. All transformations were performed on the same workstation as in the evaluation.

## IV. Result analysis

### A. Metrics

In line with prior work [7], we evaluate the logging statement generation performance concerning three ingredients: *logging levels*, *logging variables*, and *logging texts*. Although different ingredients emphasize various aspects of runtime information, they are indispensable and complementary resources for engineers to reason about system behavior.

(1) Logging levels. Following previous studies [11], [12], we use the level accuracy *(L-ACC)* and Average Ordinal Distance Score *(AOD)* for evaluating logging level predictions. L-ACC measures the percentage of correctly predicted log levels out of all suggested results. AOD [11] considers the distance between logging levels. Consequently, given the five logging levels in their severity order, i.e., `error, warn, info, debug, trace`, the distance of $Dis(error, warn) = 1$ is shorter than the distance of $Dis(error, info) = 2$. AOD takes the average distance between the actual logging level $a_i$ and the suggested logging level (denoted as $Dis(a_i, s_i)$). AOD is therefore formulated as $AOD = \frac{\sum_{i=1}^{N}(1-Dis(a_i,s_i)/MaxDis(a_i))}{N}$, where $N$ is the number of logging statements and $MaxDis(a_i)$ refers to the maximum possible distance of the actual log level.

(2) Logging variables. Evaluating predictions from LLMs is different from neural-based classification networks, as the predicted probabilities of each variable are not known. We thus employ *Precision*, *Recall*, and *F1* to evaluate predicted logging variables. For each predicted logging statement, we use $S_{pd}$ to denote variables in LLM predictions and $S_{gt}$ to denote the variables in the actual logging statement. We report the proportion of correctly predicted variables (precision=$\frac{S_{pd} \cap S_{gt}}{S_{pd}}$), the proportion of actual variables predicted by the model (recall=$\frac{S_{pd} \cap S_{gt}}{S_{gt}}$), and their harmonic mean (F1=$2 * \frac{Precision*Recall}{Precision+Recall}$).

(3) Logging texts. To align with previous research [15], [76], we assess the quality of the produced logging texts using two well-established machine translation evaluation metrics: *BLEU* [77] and *ROUGE* [78]. These n-gram metrics compute the similarity between generated log messages and the actual logging text crafted by developers, yielding a percentage score ranging from *0* to *1*. A higher score indicates greater similarity between the generated log messages and the actual logging text. In particular, we use BLEU-K ($K = \{1, 2, 4\}$) and ROUGE-K ($K = \{1, 2, L\}$) to compare the overlap concerning K-grams between the generated and the actual logs. In addition to the token-based match in a sparse space, we also incorporate *semantic similarity* in our evaluation. Following prior works [79], [80], [74],we also leverage widely-used code embedding models, UniXcoder [19] and OpenAI embedding [81], to embed the logging texts to calculate the semantics similarity between generated and original logging texts, offering another evaluation metric from a semantic perspective.

TABLE VI
The effectiveness of LLMs in predicting logging levels and logging variables.

| Model | Logging Levels | | Logging Variables | | |
|---|---|---|---|---|---|
| | **L-ACC** | **AOD** | **Precision** | **Recall** | **F1** |
| *General-purpose LLMs* | | | | | |
| Davinci | 0.631 | 0.834 | 0.634 | 0.581 | 0.606 |
| ChatGPT | 0.651 | 0.835 | 0.693 | 0.536 | 0.604 |
| GPT-4o | 0.719 | 0.868 | 0.715 | 0.647 | 0.679 |
| Llama2 | 0.595 | 0.799 | 0.556 | 0.608 | 0.581 |
| Llama3.1 | 0.704 | 0.817 | 0.721 | 0.624 | 0.669 |
| *Logging-specific LLMs* | | | | | |
| LANCE[†] | 0.612 | 0.822 | 0.667 | 0.420 | 0.515 |
| *Code-based LLMs* | | | | | |
| InCoder | 0.608 | 0.800 | 0.712 | 0.655 | 0.682 |
| CodeGeex | 0.673 | 0.855 | 0.704 | 0.616 | 0.657 |
| TabNine | 0.734 | 0.880 | 0.729 | 0.670 | 0.698 |
| Copilot | **<u>0.743</u>** | **<u>0.882</u>** | 0.722 | **<u>0.703</u>** | 0.712 |
| CodeWhisperer | 0.741 | 0.881 | **<u>0.787</u>** | 0.668 | **<u>0.723</u>** |
| CodeLlama | 0.614 | 0.814 | 0.583 | 0.603 | 0.593 |
| StarCoder | 0.661 | 0.829 | 0.656 | 0.649 | 0.653 |

[†] Since LANCE decides logging point and logging statements simultaneously, we only consider its generated logging statements with correct locations.

### B. *RQ1: How do different LLMs perform in deciding ingredients of logging statements generation?*

To answer RQ1, we evaluate 13 top-performing LLMs on the benchmark dataset LogBench-O. The evaluation results are shown in Table VI (levels, variables) and Table VII (logging texts), where we **<u>underline</u>** the best performance score for each metric.

TABLE VII
THE EFFECTIVENESS OF LLMS IN PRODUCING LOGGING TEXTS.

| Model | Logging Texts | | | | | | |
|---|---|---|---|---|---|---|---|
| | **BLEU-1** | **BLEU-2** | **BLEU-4** | **ROUGE-1** | **ROUGE-2** | **ROUGE-L** | **Semantics Similarity** |
| General-purpose LLMs | | | | | | | |
| Davinci | 0.288 | 0.211 | 0.138 | 0.295 | 0.127 | 0.286 | 0.617 |
| ChatGPT | 0.291 | 0.217 | 0.149 | 0.306 | 0.142 | 0.298 | 0.633 |
| GPT-4o | 0.384 | 0.249 | 0.174 | 0.415 | 0.236 | 0.408 | 0.694 |
| Llama2 | 0.235 | 0.168 | 0.102 | 0.264 | 0.116 | 0.261 | 0.569 |
| Llama3.1 | 0.372 | 0.243 | 0.168 | 0.377 | 0.185 | 0.372 | 0.688 |
| Logging-specific LLMs | | | | | | | |
| LANCE† | 0.306 | 0.236 | 0.167 | 0.162 | 0.078 | 0.162 | 0.347 |
| Code-based LLMs | | | | | | | |
| InCoder | 0.369 | 0.288 | 0.203 | 0.390 | 0.204 | 0.383 | 0.640 |
| CodeGeex | 0.330 | 0.248 | 0.160 | 0.339 | 0.149 | 0.333 | 0.598 |
| TabNine | 0.406 | 0.329 | 0.242 | 0.421 | 0.241 | 0.415 | 0.669 |
| Copilot | **0.417** | **0.338** | 0.244 | **0.435** | 0.247 | **0.428** | **0.703** |
| CodeWhisperer | 0.415 | 0.338 | **0.249** | 0.430 | **0.248** | 0.425 | 0.672 |
| CodeLlama | 0.216 | 0.146 | 0.089 | 0.258 | 0.103 | 0.251 | 0.546 |
| StarCoder | 0.353 | 0.278 | 0.195 | 0.378 | 0.195 | 0.369 | 0.593 |

† Since LANCE decides logging point and logging statements simultaneously, we only consider its generated logging statements with correct locations.

**Intra-ingredient.** Regarding the logging levels, we observe that Copilot achieves the best L-ACC performance, i.e., *0.743*, indicating that it can accurately predict 74.3% of the logging levels. While other baselines do not perform as well as Copilot, they also accurately suggest logging levels for at least *60%* logging statements. Compared with logging levels, there are greater differences among models when recommending logging variables. While *70%* of the variables are recommended by Copilot, LANCE can only correctly infer *42%* of them. The recall rate for variable prediction is consistently lower than the precision rate across models, indicating the difficulty of identifying many of the variables. Predicting variables is more challenging than logging levels, as variables are diverse, customized, and have different meanings across systems. To address this challenge, logging variables should be inferred based on a deeper comprehension of code structure, such as control flow information.

Concerning logging text generation shown in Table VII, both Copilot and CodeWhisperer demonstrate comparable performance across syntax-based metrics (BLEU, ROUGE) and semantic-based metrics, outperforming other baselines by a wide margin. The comparison between syntax-based metrics and semantic-encoding metrics reveals a consistent trend across various LLMs: models exhibiting strong syntax similarity also exhibit high semantic similarity. On average, the studied models produce logging statements with a similarity of *0.175* and *0.338* for BLEU-4 and ROUGE-L scores, respectively. The result indicates that recommending appropriate logging statements remains a great challenge.

Surprisingly, Table VII reveals that Llama2 surpasses CodeLlama in generating logging text, even though CodeLlama was trained on a large-scale code corpus. This outcome is attributed to the "zero-shot" settings in our evaluation, suggesting that CodeLlama may need more examples to effectively learn the in-context code for enhanced performance. With a number of the same examples provided for demonstration, CodeLlama can outperform Llama2. This phenomenon has also been noted in other code intelligence tasks [82].

Meanwhile, we observe that commercial code plugins like TabNine, Copilot, and CodeWhisperer, deliver better logging abilities compared to other LLMs, though there is still potential for enhancement. Given the significant influence of prompts on model outcomes, we highlight the need for integrating more sophisticated prompt technologies, such as in-context learning, into open-sourced LLMs to optimize their performance.

These latest advanced models, Llama3.1 and GPT-4o, have successfully predicted 70% of log levels and variables, surpassing the performance of other general-purpose LLMs in the logging statement generation task. Nevertheless, some code-specific LLMs still outperform both GPT-4o and Llama3.1, likely due to the extensive programming knowledge embedded in their training datasets. Another plausible reason is the effects of prompt design, where simple prompts are employed in this experiment. However, more advanced prompts could potentially boost the performance of these open-source LLMs.

**Finding 1.** *While existing models correctly predict levels for 74.3% of logging statements, there is significant room for improvement in producing logging variables and logging texts.*

**Inter-ingredient.** From the inter-ingredient perspective, we observe that LLM performance trends are *not consistently the same* across various ingredients, e.g., models that perform well in logging level prediction do not necessarily excel in generating logging texts. For instance, Incoder fares worst in predicting logging levels but performs better in generating logging texts (the fourth best performer). Upon manual investigation, we observe that Incoder predicts *41%* of the cases with a `debug` level, most of which are actually intended for the `info` level statements. Nevertheless, either Copilot or CodeWhisperer outperforms other baselines in all reported

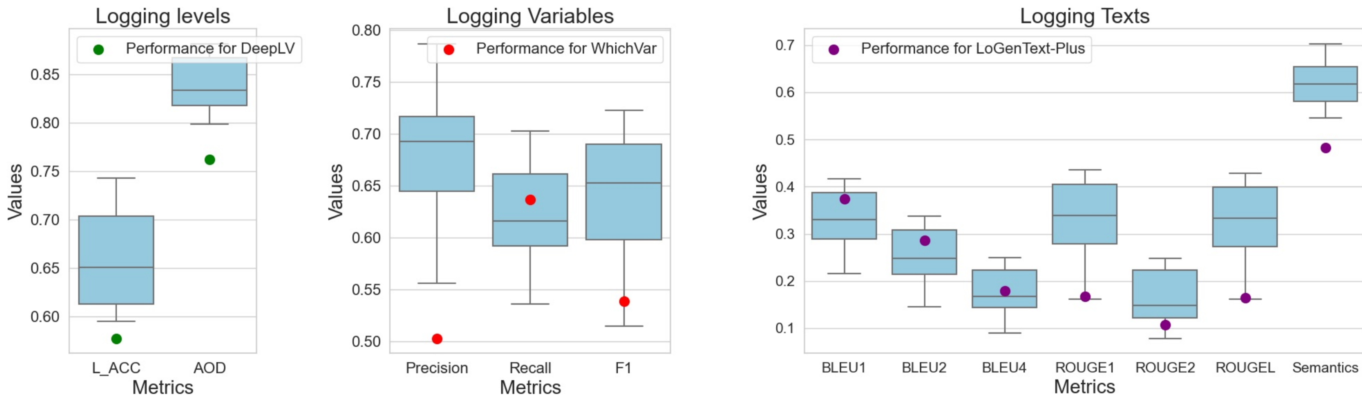



Fig. 4. The box plot (whisker plot) for traditional logging models and LLM-powered models. The box denotes the performance range of LLM-powered models, whereas the dot denotes the performance of traditional approaches. The box is drawn from the first quartile to the third quartile. A vertical line goes through the box at the median. The whiskers go from each quartile to the minimum or maximum.

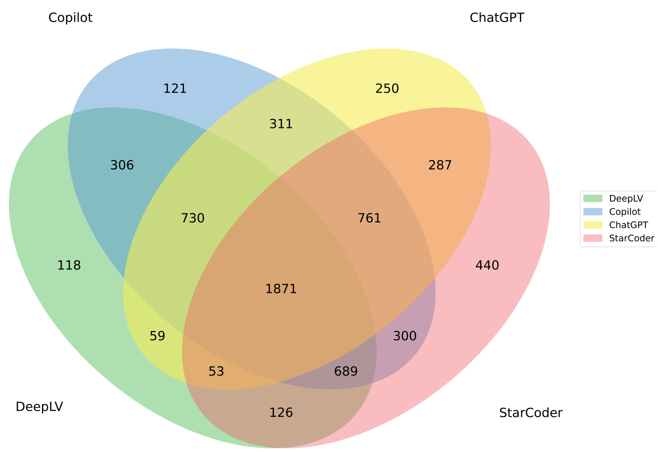



Fig. 5. Venn diagram for logging levels prediction.

```
public Object addingBundle(Bundle bundle, BundleEvent event)
{       ...
        List<Driver> drivers = loadDrivers(bundle, providerURL);
        register(drivers);
+       LOG.debug("Registered {} drivers in bundle {}", drivers, bundle);
        ...
```

| Models | Generated Logging Statements |
|---|---|
| ChatGPT | LOG.info("Bundle {0} has been added in event.", bundle.getName()) |
| InCoder | LOG.info("Registered JDBC drivers: {}", drivers); |
| TabNine | LOG.info("Registered " + drivers.size() + " drivers") |
| Copilot | LOG.debug("Found " + drivers.size() + " registered drivers in bundle" + bundle); |

Fig. 6. An example of the generation results from eight models.

metrics. This is likely because suggesting the three ingredients requires similar code comprehension capabilities, such as understanding data flows, specific code structures, and inferring code functionalities.

**Finding 2.** *LLMs may perform inconsistently on deciding different ingredients, making model comparisons more difficult based on multiple ingredient-wise metrics.*

### *C. RQ2: How do LLMs compare to conventional logging models in logging ability?*

We compare the results of directly using LLMs for logging against conventional logging models on LogBench-O. As conventional logging models can only predict one ingredient, we opt for state-of-the-art models for each one (i.e., DeepLV, WhichVar, and LoGenText-Plus) and present their performance against LLMs in Fig. 4. The boxplot illustrates the performance range of LLM-powered models, while the points depict conventional logging models.

Despite being carefully designed for the logging task, the conventional logging models do not surpass LLMs. As shown in Fig. 4, conventional models exhibit inferior performance compared to any LLMs on five metrics (i.e., below the lower whiskers) and fall below the median on the other three metrics (i.e., below the line in the box). In terms of logging level prediction, DeepLV performs worse than any of our studied LLMs, correctly predicting only 57.7% of statements. Regarding generating logging variables and texts, WhichVar and LoGenText-Plus show comparable performance to LANCE, but lag behind other studied LLMs. While the most effective model (Copilot) achieves a 0.703 semantic-based similarity in logging texts, the state-of-the-art logging model, LoGenText-Plus, only produces a 0.485 similarity (yielding a 21.8% drop). These surprising results show that, *without any specific change or fine-tuning, directly applying LLMs for logging statement generation yields better performance compared to conventional logging baselines.*

Figure 5 displays the Venn diagram illustrating the logging levels correctly predicted by DeepLV in comparison to three chosen LLMs on the LogBench-O dataset. Notably, 97% of the cases handled by DeepLV can also be predicted by LLMs. In contrast, DeepLV can only handle 70%, 62%, and 60% of the cases successfully predicted by Copilot, ChatGPT, and StarCoder, respectively.

To demonstrate the ability of LLMs, we present Fig. 6 to illustrate some statements produced by ChatGPT, InCoder, Copilot, and TabNine, respectively. Through pre-training, these LLMs gain a basic understanding of method activity in adding bundles with drivers, leading to the generation of relevant logging variables. Notably, code-based LLMs produce more accurate logging statements compared to models pre-trained for general purposes. In Fig. 6, general-purpose LLMs (i.e., ChatGPT) mispredict the logging statement by focusing on the `event` variable in the method declaration, overlooking the driver registration process preceding the logging point. Conversely, most code models (e.g., InCoder) capture such processes, recognizing that `drivers` are critical variables describing a device status. We attribute the performance difference to the gap between natural and programming languages. Training on a code base enables these models to acquire programming knowledge, bridging the gap and enhancing logging performance.

**Finding 3.** *When directly applying LLMs to logging statement generation, without fine-tuning, they still yield better performance than conventional logging baselines.*

*D. RQ3: How do the prompts for LLMs affect logging performance?*

Previous literature has identified the variance of input prompts can significantly affect the performance of LLMs' [79]. For the LLMs that can take prompts (i.e., ChatGPT, LLaMa2), we investigate the influences of instructions and demonstrate examples for their logging purpose.

**Impact of different instructions.** LLMs have been shown to be sensitive to the instructions used to query the LLM sometimes. To compare the impact of different instructions, we conducted a two-round survey involving 54 developers from a world-leading technical company, each possessing a minimum of two years of development experience. To begin with, we ask the developers to individually propose 10 instructions that they would consider when utilizing LLMs for generating logging statements. Subsequently, we distributed a second questionnaire, asking developers to choose the top 5 instructions from the initial round that they are likely to employ. Eventually, instructions receiving the top 5 votes will be considered for evaluation, shown as follows.

1) Your task is to generate the logging statement for the corresponding position.
2) You are an expert in software DevOps; please help me write the informative logging statement.
3) Complete the logging statement while taking the surrounding code into consideration.
4) Your task is to write the corresponding logging statement. Note that you should keep consistent with current logging styles.
5) Please help me write an appropriate logging statement below.

We then feed these representative instructions into two studied LLMs, that is, ChatGPT, and LLaMa2, respectively. The box plot in Fig. 7 exhibits logging performance associated with different instructions. The selected instructions result in approximately 3% performance variance for each metric, revealing the importance of designing prompts. Among all metrics, the difference in logging variable prediction for ChatGPT is slightly larger, but still in the range of 4% variation. Despite there being small variations due to different instructions, these variances do not alter the consistent superiority of ChatGPT over LLaMa2. In summary, as long as the logging ability of LLMs is evaluated using the same instructions, such evaluation and comparison are meaningful.

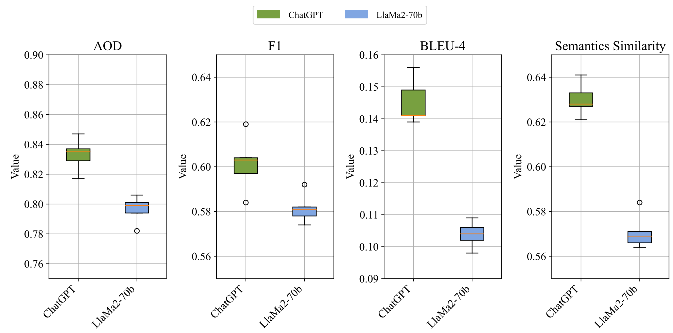



Fig. 7. The selected metrics of LLMs' logging performance with different instructions.

**Finding 4.** *Although instructions influence LLMs to varying extents, there is cohesiveness in the relative ranking of LLMs with the same instructions.*

**Impact of different numbers of logging examples.** In-context learning (ICL) is a prevalent prompt strategy, enabling LLMs to glean insights from few-shot examples in the context. Many studies have shown that LLMs can boost complicated code intelligence tasks through ICL implementation [79]. Despite being promising, there are intriguing properties that require further exploration, for example, the effects of parameter settings in ICL.

Fig. 8 presents the logging performance (i.e., logging level, variable, texts) in terms of different numbers of demonstration examples provided. In this experiment, we vary the number of demonstrations for ChatGPT and LLaMa2 from 1 to 9. We select and order demonstration examples measured by using BM25 retrieval methods, as previous works have demonstrated its effectiveness in code tasks [79]. All the examples were sourced from the LogBench-O datasets and ranked according to their similarity scores calculated using BM25 metrics. For any given number of examples set to k, the top-k most similar examples were chosen.

The figure illustrates the impact of the number of demonstration examples on LLMs' logging performance, resulting in a increment of 2%-8%. Initially, the performance of ICL improves across all metrics as the number of demonstration examples increases. However, when the number of examples surpasses 5, divergent trends emerge for different tasks. For instance, in determining logging levels (AOD) and logging variables (F1), the LLaMa performance peaks at 5 demonstration examples but experiences a decline with further increments to 7. Conversely, in logging text generation (BLEU-4, Semantics Similarity), LLaMa performance continues to rise and stabilizes beyond 7 examples. We attribute these diverse trends to the *model distraction problem* [83]. Tasks involving predicting logging levels and variables demand an

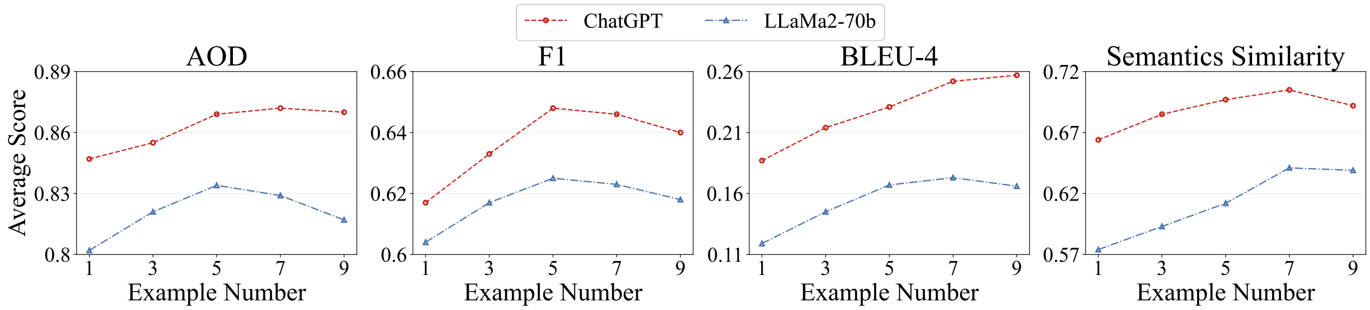



Fig. 8. The selected metrics of LLMs' logging performance with different numbers of examples.

TABLE VIII
THE RESULTS OF LOGGING STATEMENT GENERATION WITHOUT COMMENTS.

| Model | Logging Levels | Logging Variables | Logging Texts | | |
|---|---|---|---|---|---|
| | AOD | F1 | BLEU-4 | ROUGE-L | Semantics Similarity |
| Davinci | 0.834 (0.0%-) | 0.587 (3.1%↓) | 0.133 (3.6%↓) | 0.283 (1.0%↓) | 0.608 (1.5%↓) |
| ChatGPT | 0.833 (0.2%↓) | 0.592 (2.0%↓) | 0.149 (0.0%-) | 0.294 (1.3%↓) | 0.614 (3.0%↓) |
| GPT-4o | 0.861 (0.8%↓) | 0.664 (2.2%↓) | 0.168 (3.4%↓) | 0.397 (2.7%↓) | 0.678 (2.3%↓) |
| Llama2 | 0.789 (1.3%↓) | 0.574 (1.2%↓) | 0.099 (2.9%↓) | 0.255 (2.3%↓) | 0.544 (4.4%↓) |
| Llama3.1 | 0.811 (0.7%↓) | 0.652 (2.5%↓) | 0.161 (4.2%↓) | 0.358 (3.8%↓) | 0.671 (2.5%↓) |
| InCoder | 0.789 (1.4%↓) | 0.674 (1.2%↓) | 0.201 (1.0%↓) | 0.377 (9.2%↓) | 0.622 (2.8%↓) |
| CodeGeex | 0.848 (0.8%↓) | 0.617 (6.1%↓) | 0.149 (6.9%↓) | 0.306 (8.1%↓) | 0.578 (3.3%↓) |
| TabNine | 0.876 (0.5%↓) | 0.690 (1.1%↑) | 0.239 (1.2%↓) | 0.412 (0.7%↓) | 0.655 (2.1%↓) |
| Copilot | **0.878** (0.5%↓) | 0.696 (2.2%↓) | 0.241 (1.2%↓) | **0.419** (2.1%↓) | **0.689** (2.0%↓) |
| CodeWhisperer | 0.877 (0.7%↓) | **0.718** (0.7%↓) | **0.244** (2.0%↓) | 0.418 (1.6%↓) | 0.661 (1.6%↓) |
| CodeLlama | 0.804 (1.2%↓) | 0.581 (2.0%↓) | 0.087 (2.2%↓) | 0.247 (1.6%↓) | 0.544 (0.3%↓) |
| StarCoder | 0.823 (0.7%↓) | 0.647 (0.9%↓) | 0.193 (1.0%↓) | 0.369 (2.4%↓) | 0.591 (**0.3%**↓) |
| Avg.Δ | - (0.7%↓) | - (2.1%↓) | - (2.5%↓) | - (3.1%↓) | - (2.2%↓) |

intricate analysis of individual program structures and variable flows, and the introduction of additional examples with longer input lengths can potentially distract the model, leading to performance degradation. In contrast, logging text generation involves a high-level program understanding and summarization. More examples allow LLMs to learn proper logging styles from other demonstrations.

**Finding 5.** *More demonstration examples in the prompt do not always improve performance. It is recommended to use 5-7 examples in the demonstration to achieve optimal results.*

### *E. RQ4: How do external factors influence the effectiveness in generating logging statements*

While RQ3 discusses the prompt construction for LLMs, some external program information is likely to affect their effectiveness in logging generation. In particular, we focus on how *comments* and *the scope of programming contexts* will impact the model performance.

**With comment v.s. without comment.** Inspired by the importance of human-written comments for intelligent code analysis [19], [84], [85], we also explore the utility of comments for logging. To this end, we feed the original code (with comment) and comment-free code into LLMs separately, compare their results, and analyze the corresponding performance drop rate (Δ) in Table VIII in terms of AOD, F1, BLEU, and ROUGE score. The results show that LLMs consistently encounter performance drops without comments, with an average drop rate on *0.7%*, *2.1%*, *2.5%*, and *3.1%* for AOD, F1, BLEU-4, and ROUGE-L, respectively. The reason is that, comments are used to describe the functionalities of the corresponding code, thus sharing similarities to logging practices that record system activities.

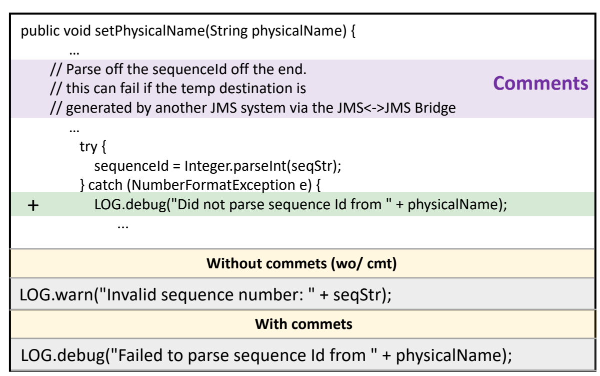



Fig. 9. A logging statement generation case using code comments.

Fig. 9 presents an example with CodeWhisperer that can be facilitated by reading the comment of `parse sequence Id`. Without the comment, CodeWhisperer only concentrates on the invalid sequence number but fails to involve parsing descriptions, which may further mislead maintainers on parsing failure diagnosis. Moreover, the comments highlight that the exception is a foreseeable and potentially common issue, which helps the LLMs in correctly selecting the log level,

TABLE IX
THE RESULTS OF LOGGING STATEMENT GENERATION WITH FILE-LEVEL CONTEXTS.

| Model | Logging Levels | Logging Variables | Logging Texts | | |
|---|---|---|---|---|---|
| | AOD | F1 | BLEU-4 | ROUGE-L | Semantics Similarity |
| Davinci | 0.854 (2.6%↑) | 0.638 (5.3%↑) | 0.156 (13.0%↑) | 0.318 (11.2%↑) | 0.635 (2.9%↑) |
| ChatGPT | 0.858 (2.8%↑) | 0.650 (7.6%↑) | 0.253 (51.5%↑) | 0.389 (30.5%↑) | 0.704 (11.2%↑) |
| GPT-4o | 0.905 (4.3%↑) | 0.724 (6.6%↑) | 0.318 (82.8%↑) | 0.552 (35.3%↑) | 0.809 (16.6%↑) |
| Llama2 | 0.832 (4.1%↑) | 0.617 (6.2%↑) | 0.149 (46.1%↑) | 0.392 (50.2%↑) | 0.669 (17.6%↑) |
| Llama3.1 | 0.848 (3.8%↑) | 0.714 (6.7%↑) | 0.301 (79.1%↑) | 0.493 (32.5%↑) | 0.794 (15.4%↑) |
| InCoder | 0.815 (1.9%↑) | 0.745 (9.2%↑) | 0.307 (51.2%↑) | 0.521 (35.3%↑) | 0.734 (11.7%↑) |
| CodeGeex | 0.869 (1.6%↑) | 0.696 (5.9%↑) | 0.241 (50.6%↓) | 0.395 (18.6%↑) | 0.644 (7.7%↑) |
| TabNine | 0.912 (3.6%↑) | 0.767 (9.9%↑) | 0.375 (55.0%↑) | 0.530 (27.7%↑) | 0.783 (17.0%↑) |
| Copilot | **0.916** (3.9%↑) | 0.742 (4.2%↑) | 0.346 (41.8%↑) | 0.522 (22.0%↑) | **0.816** (16.1%↑) |
| CodeWhisperer | 0.913 (3.6%↑) | **0.792** (9.6%↑) | **0.401** (61.0%↑) | **0.559** (31.5%↑) | 0.811 (20.7%↑) |
| CodeLlama | 0.817 (0.4%↑) | 0.607 (2.4%↑) | 0.144 (61.8%↑) | 0.378 (50.6%↑) | 0.642 (17.6%↑) |
| StarCoder | 0.847 (2.2%↑) | 0.714 (9.3%↑) | 0.314 (61.0%↑) | 0.517 (40.1%↑) | 0.679 (14.5%↑) |
| Avg.$\Delta$ | 2.9%↑ | 6.8%↑ | 54.6%↑ | 32.1%↑ | 14.1%↑ |

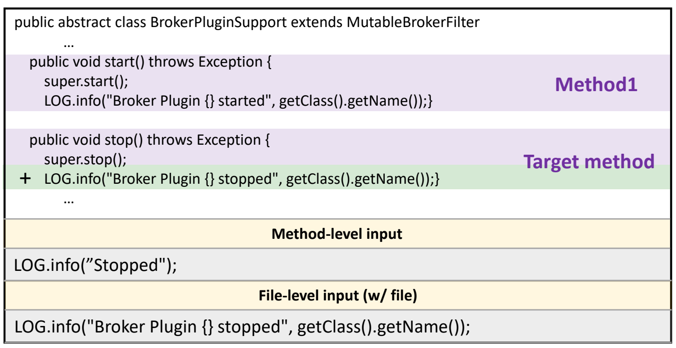



Fig. 10. A logging statement generation case using different programming contexts.

changing the logging level from `warn` to `debug`.

**Finding 6.** *Ignoring code comments impedes LLMs in generating logging statements, resulting in an average 2.2% semantics decrease when recommending logging texts.*

**Programming contexts: method v.s. file.** Current logging practice tools restrict their work on code snippets or methods [15], [76], [13], and ignore the information from other related methods [86]. However, methods that implement similar functionalities can contain similar logging statements [10], which can be used as references to resolve logging statements. In past works, this constraint was mainly due to the limits in input size in previous neural-based models. But since LLMs can now process thousands of input tokens without suffering from such limitations, we aim to assess the benefits of larger programming contexts, i.e., file-level input.

In this regard, we feed *an entire Java file* for generating logging statements rather than *the target method*. The result in Table IX presents the effectiveness of file-level input (w/ File) and the corresponding increment ratio ($\Delta$). The result suggests that file-level programming contexts consistently enhance performance in terms of all metrics where, for example, TabNine increases *3.6%*, *9.9%*, and *55.0%* for AOD, F1, and BLEU score, respectively. On average, all models generate logging statements that are *54.6%* more similar to actual ones (reflected by BLEU-4) than using a single method as input. We take Fig. 10 as an example from CodeWhisperer to illustrate how LLMs can learn from an additional method, where the green line represents the required logging statements. The model learned logging patterns from Method1, which includes the broker plugin name and its status (i.e., `start`). Regarding `stop()`, CodeWhisperer may refer to Method1 and write similar logging statements by changing the status from `started` to `stopped`. Additionally, by analyzing the file-level context, LLMs can identify pertinent variables, learn relationships between multiple methods, and recognize consistent logging styles within the file. Last but not least, the comparison of Table VIII and Table IX implies that expanding the range of programming texts has a stronger impact than incorporating comments, even though certain models (e.g., Copilot) are trained to generate code from natural language.

**Finding 7.** *Compared to comments, incorporating file-level programming contexts leads to a greater improvement in logging practice by providing access to additional functionality-similar methods, variable definitions and intra-project logging styles.*

### *F. RQ5: How do LLMs perform in logging unseen code?*

In this RQ, we assess the generalization capabilities of language models by evaluating them on the LogBench-T (Table IV). As stated in Section III-B2, predicting accurate logging statements does not necessarily imply that a model can be generalized to unseen cases well. As the modern software codebase is continuously evolving, we must explore LLMs' ability to handle these unseen cases in daily development.

We present the result in Table X, where we **underline** the best performance for each metric and the lowest performance drop rate ($\Delta$) compared to corresponding results in LogBench-O. Our experiments show that all models experience different degrees of performance degradation when generating logging statements on unseen code. LANCE has the smallest average decrease of *6.9%* across metrics, while CodeGeex is most impacted with a *16.2%* drop. Copilot exhibits the greatest

TABLE X
THE GENERALIZATION ABILITY OF LLMS IN PRODUCING LOGGING STATEMENTS FOR UNSEEN CODE.

| Model | Levels | | Variables | | Texts | | | | | | Average |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **AOD** | **Δ** | **F1** | **Δ** | **BLEU-4** | **Δ** | **ROUGE-L** | **Δ** | **Semantics** | **Δ** | **Avg. Δ** |
| General-purpose LLMs | | | | | | | | | | | |
| Davinci | 0.820 | 1.7%↓ | 0.523 | 13.7%↓ | 0.116 | 15.9%↓ | 0.234 | 20.7%↓ | 0.533 | 13.6%↓ | 13.1%↓ |
| ChatGPT | 0.830 | 0.6%↓ | 0.532 | 11.9%↓ | 0.118 | 20.8%↓ | 0.240 | 19.5%↓ | 0.541 | 14.5%↓ | 13.5%↓ |
| GPT-4o | 0.852 | 1.8%↓ | 0.624 | 8.1%↓ | 0.152 | 12.6%↓ | 0.341 | 16.4%↓ | 0.615 | 11.4%↓ | 10.1%↓ |
| Llama2 | 0.788 | 1.4%↓ | 0.568 | 2.2%↓ | 0.094 | 7.8%↓ | 0.213 | 18.4%↓ | 0.513 | 9.8%↓ | 7.9%↓ |
| Llama3.1 | 0.806 | 1.3%↓ | 0.645 | 3.6%↓ | 0.147 | 12.5%↓ | 0.310 | 17.8%↓ | 0.625 | 9.2%↓ | 8.9%↓ |
| Logging-specific LLMs | | | | | | | | | | | |
| LANCE | 0.817 | 0.6%↓ | 0.475 | 7.5%↓ | 0.153 | 8.4%↓ | 0.144 | **11.1%↓** | 0.301 | 13.3%↓ | **8.2%↓** |
| Code-based LLMs | | | | | | | | | | | |
| InCoder | 0.778 | 2.8%↓ | 0.587 | 13.9%↓ | 0.175 | 13.8%↓ | 0.316 | 17.5%↓ | 0.584 | 8.8%↓ | 11.4%↓ |
| CodeGeex | 0.850 | 0.6%↓ | 0.534 | 18.7%↓ | 0.115 | 28.1%↓ | 0.253 | 25.4%↓ | 0.549 | 8.2%↓ | 16.2%↓ |
| TabNine | 0.869 | 1.3%↓ | 0.596 | 14.6%↓ | 0.202 | 16.5%↓ | 0.342 | 18.8%↓ | 0.608 | 9.1%↓ | 12.1%↓ |
| Copilot | **0.881** | **0.1%↓** | 0.610 | 14.3%↓ | **0.234** | **4.1%↓** | **0.377** | 13.3%↓ | **0.641** | 8.8%↓ | 8.2%↓ |
| CodeWhisperer | 0.871 | 1.1%↓ | **0.629** | 13.0%↓ | 0.219 | 12.0%↓ | 0.362 | 14.6%↓ | 0.612 | 8.9%↓ | 9.9%↓ |
| CodeLlama | 0.801 | 1.6%↓ | 0.574 | **3.2%↓** | 0.078 | 12.6%↓ | 0.211 | 15.9%↓ | 0.482 | 11.7%↓ | 9.0%↓ |
| StarCoder | 0.811 | 2.2%↓ | 0.619 | 5.2%↓ | 0.175 | 10.3%↓ | 0.309 | 16.3%↓ | 0.546 | **7.9%↓** | 8.4%↓ |
| Avg. Δ | - | 1.3%↓ | - | 10.0%↓ | - | 13.5%↓ | - | 17.4%↓ | | 10.4%↓ | 10.8%↓ |

generalization capabilities by outperforming other baselines for four out of five metrics on unseen code.

Additionally, we observe that predicting logging levels has the smallest degradation in performance (*1.3%*), whereas predicting logging variables and logging text (BLEU-4) experience significant performance drops, add*10.0%* and add *13.5%*, respectively. Such experiments indicate that resolving logging variables and logging texts is more challenging than predicting logging levels, thus warranting more attention in future research.

Fig. 11 illustrates a transformation case where we highlight code differences in red and demonstrate how LLMs (CodeWhisperer, ChatGPT, Incoder) log accordingly. Regarding the original code, all models correctly predict that `inMB` should be used to record memory. However, after transforming the constant expression `1024*1024` to a new variable `const_1` and then assigning `const_1` to `inMB`, all models fail to understand and identify `inMB` (or const_1) as a logging variable. CodeWhisperer and Incoder mistakenly predict `totalMemory` and `heapMemoryUsage` as the memory size indicator without dividing it by 1024*1024 to be converted into MB units, while ChatGPT does not suggest any variables. As a result, the logging intent and the selection and usage of the logging variable are misunderstood, leading to a significant difference between the previously generated logging statement and the new one generated with the transformed data.

Even though the transformation retains code semantics, existing models exhibit a significant performance drop, indicating their limited generalization abilities.

**Finding 8.** *LLMs' performance on variable prediction and logging text generation drops significantly for unseen code by 10.0% and 13.5% on average across models, respectively, highlighting the need to improve the generalization capabilities of these models.*

Original code

```
public void countMemory(ITestContext ctx, int totalMemory) {
    long inMb = 1024 * 1024;
+   log.info("Total memory : " + totalMemory / inMb + " MB");
    MemoryMXBean memoryMXBean = ManagementFactory.getMemoryMXBean();
    MemoryUsage heapMemoryUsage = memoryMXBean.getHeapMemoryUsage();
    ...
```

Code after transformation

```
public void countMemory(ITestContext ctx, int totalMemory) {
    long const_1 = 1024 * 1024;
    long inMb = const_1;
+   log.info("Total memory : " + totalMemory / inMb+ " MB");
    MemoryMXBean memoryMXBean = ManagementFactory.getMemoryMXBean();
    MemoryUsage heapMemoryUsage = memoryMXBean.getHeapMemoryUsage();
    ...
```

| | |
|---|---|
| **Original code** | |
| CodeWhisperer | log.info("Memory usage: " + totalMemory / inMb + " MB"); |
| ChatGPT | log.info("Current heap memory usage: " + heapMemoryUsage.getUsed() / inMb + " MB"); |
| Incoder | log.debug("Memory used: " + totalMemory / inMb + "MB"); |
| **Tansformed code** | |
| CodeWhisperer | log.info("Total Memory: " + totalMemory + " MB"); |
| ChatGPT | log.info("Starting memory count..."); |
| Incoder | log.warn("Memory usage: " + heapMemoryUsage); |

Fig. 11. A case of code transformation and its corresponding predicted logging statement from multiple models.

## V. IMPLICATIONS AND ADVICE

**Pay more attention to logging texts.** According to Section IV-B, while existing models offer satisfactory predictions for logging levels, recommending proper logging variables and logging texts is difficult, particularly the latter. Since LLMs have shown stronger text generation ability than previous neural networks, future research should focus on using LLMs for the challenging problem of logging text generation instead of simply predicting logging levels.

**Implication 1.** *Future logging studies are encouraged to take advantage of prompting LLMs and focus on the challenging problem of logging text generation.*

**Devise alternative evaluation metrics.** Section IV-B extensively evaluates the performance of LLMs in generating logging statements using twelve metrics over three ingredients. We observe that a model may excel in one ingredient while performing poorly in others, and such inconsistency makes any comparison and selection of LLMs difficult. Existing metrics like BLEU and ROUGE, while suitable and being widely-used [15], [76], may not be optimal for logging statements evaluation because they do not consider semantics when assessing similarity between texts: they aggressively penalize lexical differences, even if the predicted logging statements are synonymous to the actual ones [87].

An alternative perspective to assessing the quality of logging statements involves examining the information entropy for operation engineers. Past research has highlighted that a small number of logging statements often dominate an entire log file [88], posing challenges for engineers in figuring out failure-indicating logs. These limitations underscore the need for a succinct and precise logging strategy in practical applications.

**Implication 2.** *It is recommended to investigate better, possibly unified metrics addressing all ingredients, to evaluate logging statement generation quality.*

**Refine prompts with domain knowledge.** In Section IV-D, we highlight that effective example demonstrations play a crucial role in enhancing the logging performance of LLM by imparting domain knowledge for few-shot learning. Nevertheless, our experiments reveal that augmenting the number of examples does not consistently result in improved performance. These insights elicit the development of an advanced selection strategy for choosing demonstrations, aiming to include the most informative ones in the prompt. The selection strategy can draw inspiration from program structure similarity (e.g., try-catch), syntax text similarity (e.g., TF-IDF), or code functional similarity [89].

**Implication 3.** *Designing a demonstration selection framework for effective few-shot learning can yield better results.*

**Provide broader programming contexts for LLMs.** In Section IV-E, we investigate how expanding programming contexts can significantly enhance the logging performance of LLMs. Such a finding implies that extending the context to the file level, rather than the method level, is beneficial for acquiring extra information as well as learning logging styles. However, including the entire repository as input for LLMs may be impractical for large programs due to input token limitations. Additionally, LLM performance tends to decline with longer inputs, even when within the specified context length [90], [91]. To capture effective programming contexts for specific methods, a promising solution involves identifying methods with associated calling relationships and variable definitions. Providing methods spanning multiple classes can also contribute to generating logging statements consistent with existing ones, thereby learning intra-project logging styles.

**Implication 4.** *When using LLMs for logging, future research could broaden the programming context by incorporating information from function invocations and variable definitions.*

**Enhance generalization capabilities of LLMs.** In Section IV-E, we observe that current LLMs show significantly worse performance on unseen code, reflecting their limited generalization capabilities. The result can be attributed to the capacity of parameters in LLMs to memorize large datasets [25]. This issue will become more severe when tackling code in a rapidly evolving software environment, resulting in more unseen code. One effective idea is to apply a prompt-based method with few chain-of-thought demonstrations [92], [93] to foster the generalization capabilities of ever-growing LLMs. The chain-of-thought strategy allows models to decompose complicated multi-step problems into several intermediate reasoning steps. For example, we can ask models to focus on special code structures (e.g., if-else), then advise them to elicit key variables and system activities to log. While the chain-of-thought strategy has shown success in natural language reasoning tasks [94], future work should explore such prompt-based approaches to enhance generalization capabilities.

**Implication 5.** *We should investigate prompt-based strategies with zero-shot or few-shot learning to improve the generalization ability of LLMs.*

## VI. Threats to Validity

**Internal Threats.** (1) A concern of this study is the potential bias introduced by the limited size of the LogBench-O dataset, which consists of *3,840* methods. This limitation arises due to the fact that those plugin-based code completion tools impose usage restrictions to prevent bots; therefore, human efforts are needed. To address the threat, we acquired and sampled LogBench-O and LogBench-T datasets from well-maintained open projects, which we believe are representative. Note that existing Copilot testing studies also have used datasets of comparable sizes [84], [95].

(2) Another concern involves the context length limitations of certain language models [18], [34], [32] (e.g., *4,097* tokens for Davinci), which may affect the file-level experiment. To address this concern, we analyze the collected data and reveal that *98.6%* of the Java files fall within the *4096*-token limit, and *94.3%* of them are within the *2048*-token range. Such analysis implies that the majority of files in our dataset remain unaffected by the context length restrictions.

(3) The other threat is the potential effect of various prompts on Davinci and ChatGPT. To address this, we invited four authors to independently provide three prompts according to their usage habits. These prompts were evaluated using a dataset of 100 samples, and the one that demonstrated the best performance was selected. This approach ensures that the chosen prompt is representative of daily development.

(4) There are concerns about comparing DL-based methods with LLMs in "untrained" repositories, since DL-based methods are typically tested within the same trained project, which could impact their performance. However, we anticipate that the future of automatic logging will lean towards project-agnostic solutions to enhance practicality and generalizability. To rigorously assess the effectiveness across general logging codebases, we strictly follow the design and training process of the DL-based approaches, as well as benchmark their performance against LLMs using the same evaluation dataset.
**External Threats.** One potential external threat stems from the fact that the LogBench-O dataset was mainly based on the Java language, which may affect the generalizability of our findings to other languages. However, according to previous works [11], [12], [15], Java is among the most prevalent programming languages for logging research purposes, and both SLF4J and Log4j are highly popular and widely adopted logging APIs within the Java ecosystem. We believe the representativeness of our study is highlighted by the dominance of Java languages and these APIs in the logging domain. The core idea of the study can still be generalized to other logging frameworks or languages.

## VII. Related Work

### A. Logging Statement Automation

The logging statement automation studies focus on automatically generating logging statements, which can be divided into two categories: *what-to-log* and *where-to-log*. *What-to-log* studies are interested in producing concrete logging statements, which include deciding the appropriate log level (e.g., warn, error) [11], [12], [14], choosing suitable variables [13], [96], [97], and generating proper logging text [15], [76]. For example, ordinal-based neural networks [11] and graph neural networks [12] have been applied to learn syntactic code features and semantic text features for log-level suggestions. LogEnhancer [97] aims to reduce the burden of failure diagnosis by inserting causally-related variables in a logging statement from a programming analysis perspective, whereas Liu et al. [13] predicts logging variables for developers using a self-attention neural network to learn tokens in code snippets. *Where-to-log* studies concentrating on suggesting logging points in source code [30], [98]. Excessive logging statements can enhance unnecessary efforts in software development and maintenance, while insufficient logging statements lead to missing key system behavior information for potential system diagnosis [27], [28]. To automate logging points, previous studies solve the log placement problem in specific code construct types, such as catch [99], if [99], and exception [100]. Li et al. [30] proposes a deep learning-based framework to suggest logging locations by fusing syntactic, semantic and block features extracted from source code. The most recent model in T5 architecture, LANCE [15], provides a one-stop logging statements solution for deciding logging points and logging contents for code snippets.

Although these works tried new emerging deep-learning models to determine logging statements, they have certain limitations: some focus solely on specific logging ingredients or are designed for particular scenarios. Consequently, these work and their proposed datasets, holding different experimental settings, which are not well-suited for evaluating logging ability for daily development. Moreover, they lack the analysis of the model itself (e.g., potential influencing factors) and comprehensive evaluation (e.g., performance across multiple ingredients). To fill the gap, our study is the first one that investigates and compares current LLMs for automated logging generation, which facilitates future research in developing, applying, and integrating these large models in practice.

### B. Empirical Study on Logging Practice

Logging practices have been widely studied to guide developers in writing appropriate logging statements, because modern log-based software maintenance highly depends on the quality of logging code [97], [101], [102]. Logging too little or too much will both hinder the failure diagnosis process [103]. To reveal how logging practices in the industry help engineers make logging decisions, Fu et al. [27] analyzes two large-scale online service systems involving 54 experienced developers at Microsoft, providing six insightful findings concerning the logging code categories, decisional factors, and auto-logging feasibility. Another industrial study [104] indicates that the logging process is developer-dependent and thus strongly suggested standardizing event logging activities company-wide. Exploration studies on logging statements' evolution over open software projects have also been conducted [103], [105], [106], revealing that paraphrasing, inserting, and deleting logging statement operations are prevalent during software evolution. Chen et al. [101] revisits the logging instrumentation pipeline with three phases, including logging approach, logging utility integration, and logging code composition. While some studies [7], [101] introduce the existing what-to-log approaches with technical details, their main emphasis lies in the overall log workflow, encompassing proactive logging generation and reactive log management. However, they do not offer a qualitative comparison or a discussion on the characteristics of the logging generation tools.

In summary, even though logging practices have been widely studied as a crucial part of software development, there exists neither a benchmark evaluation of logging generation models nor a detailed analysis of them. To bridge the gap, this study is the first empirical analysis of LLM-based logging statement generation tools by benchmarking existing solutions. The findings and implications can further guide researchers to build more effective and practical automated logging models.

### C. Large Language Models for Code

The remarkable success of LLMs in the NLP has prompted the development of pre-trained models in other areas, particularly in intelligent code analysis [20], [22], [107]. CodeBERT [108] adopts the transformer architecture [109] and has been trained on a blend of programming and natural languages to learn a general representation for code, which can further support generating a program from a natural language specification. In addition to sequence-based models, GraphCodeBERT [110] considers the code property of its structural and logical relationship (e.g., data flow, control flow), creating a more effective model for code understanding tasks [111]. Furthermore, Guo et al. [19] presents UniXCoder, which is a unified cross-modal pre-trained model for programming language. UniXcoder employs a mask attention mechanism to regulate the model's behavior and trains with cross-modal contents such as AST and code comment to enhance code representation. The recent work, InCoder [18], is adept at handling generative tasks (e.g., comment generation) after learning bidirectional context for infilling arbitrary code lines.

As the use of large code models grows, many of them have been integrated into IDE plugins [39], [42], [22], [112] to assist developers in their daily programming. Nonetheless, existing code intelligence research focuses on functional code and these non-functional logging statements have never been explored. By extensively examining the performance of LLMs in writing logging statements, this paper contributes to a deeper understanding of the potential applications of LLMs in automated logging.

## VIII. Conclusion

In this paper, we present the first extensive evaluation of LLMs for generating logging statements. To achieve this, we introduce a logging statement generation benchmark dataset, LogBench, and assess the effectiveness and generalization capabilities of 13 top-performing LLMs. While LLMs are promising in generating complete logging statements, they can still be promoted in multiple ways.

First, our evaluation indicates that existing LLMs are not yet adept at generating complete logging statements, particularly in producing effective logging texts. Nonetheless, their direct application surpasses the performance of conventional logging models, indicating a promising future for leveraging LLMs in logging practices.

In addition, we delve into the construction of prompts that influence LLMs' logging performance, considering factors such as instructions and the number of example demonstrations. While our experiments demonstrate the advantages of incorporating demonstrations, we observe that an increased number of demonstrations does not consistently result in improved logging performance. Thus, we recommend the development of a demonstration selection framework in future research. Furthermore, we identify external factors, such as comments and programming contexts, that enhance model performance. We encourage the incorporation of such factors to enhance LLM-based logging tools.

Last but not least, we evaluate LLMs' generalization ability using a dataset that includes transformed code. Our findings indicate that directly applying LLMs to unseen code results in a significant decline in performance, highlighting the necessity to enhance their inference abilities. We suggest employing the chain-of-thought technologies to break down the logging task into smaller logical steps as a future step, unlocking LLMs' full potential. We hope this paper can stimulate more work in the promising direction of using LLMs for automatic logging.

## IX. Data Availability

The datasets LogBench-O and LogBench-T, source code, and code transformation tool are available at the anonymous Github link: https://github.com/logpai/LogBench.

## X. Acknowledgement

We thank all reviewers and editors for their valuable comments and suggestions. This research was supported by several funding bodies. We would like to thank the Research Grants Council of the Hong Kong Special Administrative Region, China, for their support under the General Research Fund (Project No. CUHK 14206921). Pinjia He acknowledges the support from the Guangdong Basic and Applied Basic Research Foundation (Grant No. 2024A1515010145). Lionel Briand was supported by the Canada Research Chair and Discovery Grant programs of the Natural Sciences and Engineering Research Council of Canada (NSERC), as well as the Science Foundation Ireland (Grant No. 13/RC/2094-2).

## References

[1] B. Chen, "Improving the software logging practices in devops," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 194–197.

[2] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*, 2009, pp. 117–132.

[3] Y. Huo, Y. Su, C. Lee, and M. R. Lyu, "Semparser: A semantic parser for log analytics," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 881–893.

[4] Y. Huo, C. Lee, Y. Su, S. Shan, J. Liu, and M. Lyu, "Evlog: Evolving log analyzer for anomalous logs identification," *arXiv preprint arXiv:2306.01509*, 2023.

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

[6] Z. A. Khan, D. Shin, D. Bianculli, and L. Briand, "Impact of log parsing on log-based anomaly detection," *arXiv preprint arXiv:2305.15897*, 2023.

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

[8] S. Gholamian, "Leveraging code clones and natural language processing for log statement prediction," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1043–1047.

[9] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 102–112.

[10] 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 (ASE)*, 2018, pp. 178–189.

[11] 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)*. IEEE, 2021, pp. 1461–1472.

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

[13] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Which variables should i log?" *IEEE Transactions on Software Engineering (TSE)*, vol. 47, no. 9, pp. 2012–2031, 2019.

[14] H. Li, W. Shang, and A. E. Hassan, "Which log level should developers choose for a new logging statement?" *Empirical Software Engineering (ESE)*, vol. 22, pp. 1684–1716, 2017.

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

[16] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, pp. 681–694, 2020.

[17] 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," *arXiv preprint arXiv:1907.11692*, 2019.

[18] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022.

[19] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[20] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, 2023.

[21] GitHub, "Github copilot: Parrot or crow? a first look at rote learning in github copilot suggestions." Mar 2023. [Online]. Available: https://github.blog/2021-06-30-github-copilot-research-recitation/

[22] ——, "Github copilot: Your ai pair programmer," Mar 2023. [Online]. Available: https://github.com/features/copilot

[23] Amazon, "Codewhisperer," Mar 2023. [Online]. Available: https://aws.amazon.com/cn/codewhisperer/

[24] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

[25] M. R. I. Rabin, A. Hussain, M. A. Alipour, and V. J. Hellendoorn, "Memorization and generalization in neural code intelligence models," *Information and Software Technology (Inf. Softw. Technol.)*, vol. 153, p. 107066, 2023.

[26] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," *arXiv preprint arXiv:2302.05020*, 2023.

[27] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014, pp. 24–33.

[28] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, vol. 1. IEEE, 2015, pp. 415–425.

[29] H. Li, T.-H. Chen, W. Shang, and A. E. Hassan, "Studying software logging using topic models," *Empirical Software Engineering*, vol. 23, pp. 2655–2694, 2018.

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

[31] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[32] OpenAI., "Gpt-3.5," Mar 2022. [Online]. Available: https://platform.openai.com/docs/models/gpt-3-5

[33] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," *Advances in neural information processing systems*, vol. 30, 2017.

[34] OpenAI, "Chatgpt," Mar 2023. [Online]. Available: https://openai.com/blog/chatgpt/

[35] OpenAI, "Gpt-4o," 2024, accessed: 2024-06-18. [Online]. Available: https://openai.com/index/hello-gpt-4o/

[36] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[37] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[38] LANCE, "Replication package of lance." Jan 2022. [Online]. Available: https://github.com/antonio-mastropaolo/LANCE#using-deep-learning-to-generate-complete-log-statements

[39] CodeGeeX, "Codegeex," Mar 2023. [Online]. Available: https://models.aminer.cn/codegeex/blog/

[40] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[41] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[42] Tabnine, "Tabnine," Mar 2023. [Online]. Available: https://www.tabnine.com/

[43] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[44] Z. Ding, Y. Tang, X. Cheng, H. Li, and W. Shang, "Logentext-plus: Improving neural machine translation-based logging texts generation with syntactic templates," *ACM Trans. Softw. Eng. Methodol.*, sep 2023, just Accepted. [Online]. Available: https://doi.org/10.1145/3624740

[45] C. Zhi, J. Yin, S. Deng, M. Ye, M. Fu, and T. Xie, "An exploratory study of logging configuration practice in java," in *2019 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2019, pp. 459–469.

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

[47] Z. Ding, Y. Tang, X. Cheng, H. Li, and W. Shang, "Logentext-plus: Improving neural machine translation based logging texts generation with syntactic templates," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–45, 2023.

[48] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[49] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt, "Measuring massive multitask language understanding," *arXiv preprint arXiv:2009.03300*, 2020.

[50] Y. Huo, Y. Li, Y. Su, P. He, Z. Xie, and M. R. Lyu, "Autolog: A log sequence synthesis framework for anomaly detection," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 497–509.

[51] S. Black, S. Biderman, E. Hallahan, Q. Anthony, L. Gao, L. Golding, H. He, C. Leahy, K. McDonell, J. Phang *et al.*, "Gpt-neox-20b: An open-source autoregressive language model," *arXiv preprint arXiv:2204.06745*, 2022.

[52] EleutherAI., "Gpt-j," Mar 2022. [Online]. Available: https://huggingface.co/EleutherAI/gpt-j-6B.

[53] Y. Tan, D. Min, Y. Li, W. Li, N. Hu, Y. Chen, and G. Qi, "Can chatgpt replace traditional kbqa models? an in-depth analysis of the question answering performance of the gpt llm family," in *International Semantic Web Conference*. Springer, 2023, pp. 348–367.

[54] T. Goyal, J. J. Li, and G. Durrett, "News summarization and evaluation in the era of gpt-3," *arXiv preprint arXiv:2209.12356*, 2022.

[55] J. Ye, X. Chen, N. Xu, C. Zu, Z. Shao, S. Liu, Y. Cui, Z. Zhou, C. Gong, Y. Shen *et al.*, "A comprehensive capability analysis of gpt-3 and gpt-3.5 series models," *arXiv preprint arXiv:2303.10420*, 2023.

[56] D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, W. Yongji, and J.-G. Lou, "Large language models meet nl2code: A survey," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023, pp. 7443–7464.

[57] B. Chen and Z. M. Jiang, "Studying the use of java logging utilities in the wild," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 397–408.

[58] B. Chen, J. Song, P. Xu, X. Hu, and Z. M. Jiang, "An automated approach to estimating code coverage measures via execution logs," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018, pp. 305–316.

[59] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima *et al.*, "The pile: An

800gb dataset of diverse text for language modeling," *arXiv preprint arXiv:2101.00027*, 2020.

[60] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: the threats of using llms in software engineering," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 102–106.

[61] Y. Wu, N. Jiang, H. V. Pham, T. Lutellier, J. Davis, L. Tan, P. Babkin, and S. Shah, "How effective are neural networks for fixing security vulnerabilities," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1282–1294.

[62] C. S. Xia, Y. Deng, and L. Zhang, "Top leaderboard ranking= top coding proficiency, always? evoeval: Evolving coding benchmarks via llm," *arXiv preprint arXiv:2403.19114*, 2024.

[63] E. Quiring, A. Maier, K. Rieck *et al.*, "Misleading authorship attribution of source code using adversarial learning." in *USENIX Security Symposium (USENIX Security)*, 2019, pp. 479–496.

[64] Y. Li, S. Qi, C. Gao, Y. Peng, D. Lo, Z. Xu, and M. R. Lyu, "A closer look into transformer-based code intelligence through code transformation: Challenges and opportunities," *arXiv preprint arXiv:2207.04285*, 2022.

[65] Z. Li, C. Wang, Z. Liu, H. Wang, S. Wang, and C. Gao, "Cctest: Testing and repairing code completion systems," *arXiv preprint arXiv:2208.08289*, 2022.

[66] Z. Li, J. Tang, D. Zou, Q. Chen, S. Xu, C. Zhang, Y. Li, and H. Jin, "Towards making deep learning-based vulnerability detectors robust," *arXiv preprint arXiv:2108.00669*, 2021.

[67] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[68] H. Cheers, Y. Lin, and S. P. Smith, "Spplagiarise: A tool for generating simulated semantics-preserving plagiarism of java source code," in *2019 IEEE 10th International conference on software engineering and service science (ICSESS)*. IEEE, 2019, pp. 617–622.

[69] A. Balakrishnan and C. Schulze, "Code obfuscation literature survey," *CS701 Construction of compilers*, vol. 19, p. 31, 2005.

[70] H. Zhang, Y. Pei, J. Chen, and S. H. Tan, "Statfier: Automated testing of static analyzers via semantic-preserving program transformations," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 237–249.

[71] JavaParser, "Javaparser," Mar 2019. [Online]. Available: https://javaparser.org

[72] Deepinfra, "Deepinfra," 2024. [Online]. Available: https://deepinfra.com/

[73] Z. A. Khan, D. Shin, D. Bianculli, and L. Briand, "Guidelines for assessing the accuracy of log message template identification techniques," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 1095–1106.

[74] J. Xu, R. Yang, Y. Huo, C. Zhang, and P. He, "Prompting for automatic log template extraction," *arXiv preprint arXiv:2307.09950*, 2023.

[75] Y. Huang, Y. Li, W. Wu, J. Zhang, and M. R. Lyu, "Do not give away my secrets: Uncovering the privacy issue of neural code completion tools," *arXiv preprint arXiv:2309.07639*, 2023.

[76] 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)*. IEEE, 2022, pp. 349–360.

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

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

[79] S. Gao, X.-C. Wen, C. Gao, W. Wang, and M. R. Lyu, "Constructing effective in-context demonstration for code intelligence tasks: An empirical study," *arXiv preprint arXiv:2304.07575*, 2023.

[80] Y. Ding, Z. Wang, W. U. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth *et al.*, "Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion," *arXiv preprint arXiv:2310.11248*, 2023.

[81] OpenAI., "Openai embeddings," Aug 2023. [Online]. Available: https://platform.openai.com/docs/guides/embeddings

[82] L. Fan, J. Liu, Z. Liu, D. Lo, X. Xia, and S. Li, "Exploring the capabilities of llms for code change related tasks," *arXiv preprint arXiv:2407.02824*, 2024.

[83] Z. Yuan, J. Liu, Q. Zi, M. Liu, X. Peng, and Y. Lou, "Evaluating instruction-tuned large language models on code comprehension and generation," *arXiv preprint arXiv:2308.01240*, 2023.

[84] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota, "On the robustness of code generation techniques: An empirical study on github copilot," *arXiv preprint arXiv:2302.00438*, 2023.

[85] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering (ASE)*, 2018, pp. 397–407.

[86] J. H. Dawes, D. Shin, and D. Bianculli, "Towards log slicing," in *International Conference on Fundamental Approaches to Software Engineering*. Springer Nature Switzerland Cham, 2023, pp. 249–259.

[87] J. Wieting, T. Berg-Kirkpatrick, K. Gimpel, and G. Neubig, "Beyond bleu: training neural machine translation with semantic similarity," *arXiv preprint arXiv:1909.06694*, 2019.

[88] G. Yu, P. Chen, P. Li, T. Weng, H. Zheng, Y. Deng, and Z. Zheng, "Logreducer: Identify and reduce log hotspots in kernel on the fly," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1763–1775.

[89] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, G. T. Leavens, A. Garcia, and C. S. Pasareanu, Eds. ACM, 2018, pp. 141–151. [Online]. Available: https://doi.org/10.1145/3236024.3236068

[90] F. Shi, X. Chen, K. Misra, N. Scales, D. Dohan, E. H. Chi, N. Schärli, and D. Zhou, "Large language models can be easily distracted by irrelevant context," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 210–31 227.

[91] 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," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024.

[92] O. Rubin, J. Herzig, and J. Berant, "Learning to retrieve prompts for in-context learning," *arXiv preprint arXiv:2112.08633*, 2021.

[93] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. Chi, Q. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," *arXiv preprint arXiv:2201.11903*, 2022.

[94] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *arXiv preprint arXiv:2205.11916*, 2022.

[95] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 754–768.

[96] S. Dai, Z. Luan, S. Huang, C. Fung, H. Wang, H. Yang, and D. Qian, "Reval: Recommend which variables to log with pre-trained model and graph neural network," *IEEE Transactions on Network and Service Management (TNSM)*, 2022.

[97] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, pp. 1–28, 2012.

[98] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "Log20: Fully automated optimal placement of log printing statements under specified overhead threshold," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 565–581.

[99] S. Lal, N. Sardana, and A. Sureka, "Logoptplus: Learning to optimize logging in catch and if programming constructs," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2016, pp. 215–220.

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

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

[102] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A cost-aware logging mechanism for performance diagnosis," in *2015 USENIX Annual Technical Conference (USENIX ATC)*, 2015, pp. 139–150.

[103] B. Chen and Z. M. Jiang, "Characterizing and detecting anti-patterns in the logging code," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 71–81.

[104] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo, "Industry practices and event logging: Assessment of a critical software development process," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, vol. 2. IEEE, 2015, pp. 169–178.

[105] S. Kabinna, C.-P. Bezemer, W. Shang, M. D. Syer, and A. E. Hassan, "Examining the stability of logging statements," *Empirical Software Engineering (ESE)*, vol. 23, pp. 290–333, 2018.

[106] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora, "An exploratory study of the evolution of communicated information about the execution of large software systems," *Journal of Software: Evolution and Process (J. Softw.: Evol. Process)*, vol. 26, no. 1, pp. 3–26, 2014.

[107] C. B. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan, "Pymt5: multi-mode translation of natural language and python code with transformers," *arXiv preprint arXiv:2010.03150*, 2020.

[108] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

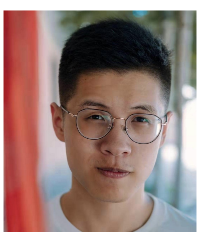
[109] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[110] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

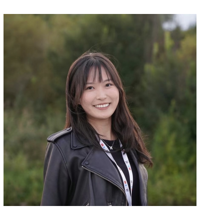
[111] A. Karmakar and R. Robbes, "What do pre-trained code models know about code?" in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1332–1336.

[112] aiXcoder, "aixcoder," Mar 2023. [Online]. Available: https://www.aixcoder.com


## BIOGRAPHIES



**Yichen Li** received his B.Eng. degree from Huazhong University of Science and Technology of China; and was pursuing his Ph.D. in Computer Science and Engineering from The Chinese University of Hong Kong. His research interests span intelligent software development and operation. He published several papers on top software engineering conferences such as ICSE, FSE, ASE and ISSTA.



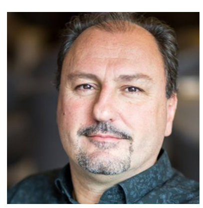
**Yintong Huo** received her B.Eng. degree from The University of Electronic Science and Technology of China; and her Ph.D. in Computer Science and Engineering from The Chinese University of Hong Kong. She is currently an assistant professor of Computer Science at Singapore Management University. Her research interests are on automatic software development and IT operations. She published several papers on top software engineering conferences such as ICSE, FSE, ASE, and ISSTA.

**Zhihan Jiang** received his B.Eng. degree from Sun Yat-sen University. He is currently a Ph.D. candidate at the Chinese University of Hong Kong. His research focuses on ensuring the reliability of large-scale cloud systems. He has published several papers in top software engineering conferences, including FSE, ISSTA, and ASE.

**Renyi Zhong** is a Ph.D student in Computer Science and Engineering, The Chinese University of Hong Kong. His research interests include intelligent and automated code-related tasks.

**Pinjia He** is an Assistant Professor at The Chinese University of Hong Kong, Shenzhen. He has been a postdoctoral researcher at ETH Zurich after receiving his PhD degree from the Chinese University of Hong Kong in 2018. His main research interests include software engineering, AI for SE, large language model, and trustworthy AI. He received the first IEEE Open Software Services Award and an ISSRE Most Influential Paper Award. The open-source projects led by him has been starred 6,000+ times on GitHub and downloaded 60,000+ times by 450+ organizations.

**Yuxin Su** received his Ph.D. degree from the Chinese University of Hong Kong in 2019. He is an Associate Professor and the Deputy Dean of the School of Software Engineering at Sun Yat-sen University. He works at the intersection of software engineering and artificial intelligence. His main research interests include software reliability, cloud computing, AI for software systems, and AIOps. He has more than 30 high-quality publications, including ICSE, ASE, ISSTA, SOSP, FAST, and TOSEM, and is the recipient of two Best Paper/Tool Awards.



**Lionel C. Briand** is professor of software engineering and has shared appointments between (1) The University of Ottawa, Canada, and (2) The Lero SFI Centre—the national Irish centre for software research—hosted by the University of Limerick, Ireland. In collaboration with colleagues, for over 30 years, he has run many collaborative research projects with companies in the automotive, satellite, aerospace, energy, financial, and legal domains. Lionel has held various engineering, academic, and leading positions in seven countries. He currently holds a Canada Research Chair (Tier 1) on "Intelligent Software Dependability and Compliance" and is the director of Lero, the national Irish centre for software research. Lionel was elevated to the grades of IEEE Fellow and ACM Fellow for his work on software testing and verification. Further, he was granted the IEEE Computer Society Harlan Mills award, the ACM SIGSOFT outstanding research award, and the IEEE Reliability Society engineer-of-the-year award. He also received an ERC Advanced grant in 2016 on modelling and testing cyber-physical systems, the most prestigious individual research award in the European Union and was elected a fellow of the Academy of Science, Royal Society of Canada in 2023.

**Michael R. Lyu** received his B.S. in Electrical Engineering from National Taiwan University, Taipei, Taiwan; his M.S. in Computer Science from University of California, Santa Barbara, USA; and his Ph.D. in Computer Science from University of California, Los Angeles, USA. He is currently Choh-Ming Li Professor of Computer Science and Engineering in The Chinese University of Hong Kong. Prof. Lyu's research interests include software engineering, software reliability, machine learning, cloud and mobile computing, and distributed systems. He has published over 600 refereed journal and conference papers in his research areas. His Google Scholar citation is over 46,000, with an h-index of 104. Prof. Lyu initiated the first International Symposium on Software Reliability Engineering (ISSRE) in 1990. He was an Associate Editor of IEEE Transactions on Reliability, IEEE Transactions on Knowledge and Data Engineering, IEEE Transactions on Services Computing, and Journal of Information Science and Engineering. He is currently on the editorial board of IEEE Access, Wiley Software Testing, Verification and Reliability Journal (STVR), and ACM Transactions on Software Engineering Methodology (TOSEM). Prof. Lyu was elected to IEEE Fellow (2004), AAAS Fellow (2007), ACM Fellow (2015), and named IEEE Reliability Society Engineer of the Year (2010). He was granted with China Computer Federation (CCF) Overseas Outstanding Contributions Award in 2018, and the 13th Guanghua Engineering Science and Technology Award in 2020. He was also named in The AI 2000 Most Influential Scholars Annual List with three appearances in 2020.