UniLog: Automatic Logging via LLM and In-Context Learning

Junjielong Xu
junjielongxu@link.cuhk.edu.cn
School of Data Science, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China

Ziang Cui
ziang_cui@seu.edu.cn
Southeast University, Nanjing, China

Yuan Zhao
zhaoyuan@stu.pku.edu.cn
Peking University, Beijing, China

Xu Zhang
xuzhang2@microsoft.com
Microsoft, Beijing, China

Shilin He
shilin.he@microsoft.com
Microsoft, Beijing, China

Pinjia He
hepinjia@cuhk.edu.cn
School of Data Science, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China

Liqun Li
liqun.li@microsoft.com
Microsoft, Beijing, China

Yu Kang
yu.kang@microsoft.com
Microsoft, Beijing, China

Qingwei Lin*
qlin@microsoft.com
Microsoft, Beijing, China

Yingnong Dang
Dang.Yingnong@microsoft.com
Microsoft, Redmond, USA

Saravan Rajmohan
saravan.rajmohan@microsoft.com
Microsoft, Redmond, USA

Dongmei Zhang
dongmeiz@microsoft.com
Microsoft, Beijing, China

ABSTRACT

Logging, which aims to determine the position of logging statements, the verbosity levels, and the log messages, is a crucial process for software reliability enhancement. In recent years, numerous automatic logging tools have been designed to assist developers in one of the logging tasks (*e.g.*, providing suggestions on whether to log in try-catch blocks). These tools are useful in certain situations yet cannot provide a comprehensive logging solution in general. Moreover, although recent research has started to explore end-to-end logging, it is still largely constrained by the high cost of fine-tuning, hindering its practical usefulness in software development. To address these problems, this paper proposes UniLog, an automatic logging framework based on the in-context learning (ICL) paradigm of large language models (LLMs). Specifically, UniLog can generate an appropriate logging statement with only a prompt containing five demonstration examples without any model tuning. In addition, UniLog can further enhance its logging ability after warmup with only a few hundred random samples. We evaluated UniLog on a large dataset containing 12,012 code snippets extracted from 1,465 GitHub repositories. The results show that UniLog achieved the state-of-the-art performance in automatic logging: (1) 76.9% accuracy in selecting logging positions, (2) 72.3% accuracy in predicting verbosity levels, and (3) 27.1 BLEU-4 score in generating log messages. Meanwhile, UniLog requires less than 4% of the parameter tuning time needed by fine-tuning the same LLM.

CCS CONCEPTS

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

KEYWORDS

Logging, Large Language Model, In-Context Learning

ACM Reference Format:
Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liqun Li, Yu Kang, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. 2024. UniLog: Automatic Logging via LLM and In-Context Learning. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3623326

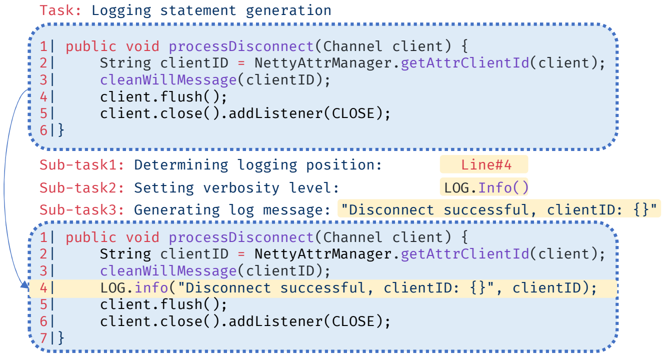
1 INTRODUCTION

With the rapid growth of software size and complexity, logging has become an increasingly indispensable practice for software reliability assurance in recent years [5, 16]. As illustrated in Fig. 1, logging involves the construction of logging statements via three distinct subtasks: (1) determining a logging position, (2) setting a verbosity level, and (3) generating a log message. During runtime, these logging statements produce software logs that record information about fault events or metrics about various KPIs, making them an important source for a series of downstream tasks in automatic log analysis, such as anomaly detection [13, 18, 30, 53], fault diagnosis [46, 56], root cause analysis [2, 17, 26], and program verification [10, 39]. The performance of these downstream log analysis tasks largely depends on the quality of software logs [15]. Thus,

[*] Qingwei Lin is the corresponding author.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3623326



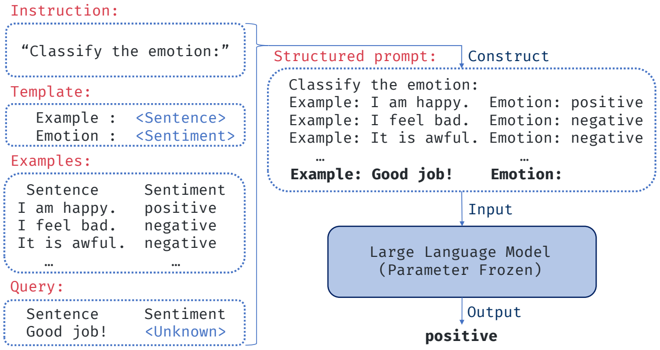
Task: Logging statement generation

```
1| public void processDisconnect(Channel client) {
2|     String clientID = NettyAttrManager.getAttrClientId(client);
3|     cleanWillMessage(clientID);
4|     client.flush();
5|     client.close().addListener(CLOSE);
6|}
```

Sub-task1: Determining logging position: Line#4
Sub-task2: Setting verbosity level: LOG.Info()
Sub-task3: Generating log message: "Disconnect successful, clientID: {}"

```
1| public void processDisconnect(Channel client) {
2|     String clientID = NettyAttrManager.getAttrClientId(client);
3|     cleanWillMessage(clientID);
4|     LOG.info("Disconnect successful, clientID: {}", clientID);
5|     client.flush();
6|     client.close().addListener(CLOSE);
7|}
```

Figure 1: An example of logging statement generation.



Instruction:

"Classify the emotion:"

Template:

```
Example :  <Sentence>
Emotion :  <Sentiment>
```

Examples:

Sentence	Sentiment
I am happy.	positive
I feel bad.	negative
It is awful.	negative
…	…

Query:

Sentence	Sentiment
Good job!	<Unknown>

Structured prompt:

```
Classify the emotion:
Example: I am happy.   Emotion: positive
Example: I feel bad.   Emotion: negative
Example: It is awful.  Emotion: negative
  …                      …
Example: Good job!     Emotion:
```

Figure 2: A simple example of in-context learning workflow on sentiment classification task.

appropriate logging is of great importance in modern software development and maintenance.

However, logging is a non-trivial task for developers. According to Yuan *et al.* [50], most unreported errors are due to the *lack of logging statements* at critical positions in the code where developers can not judge the necessity of logging. Furthermore, Zhu *et al.* [55] mentioned that *logging too little* may miss the runtime information necessary for postmortem analysis, while *logging too much* can incur heavy maintenance overhead, consume additional system resources, and produce useless logs that mask the truly important information. Li *et al.* [20] pointed out that developers often cannot evaluate the benefits of investing time in designing *verbosity levels* and *log messages* in logging statements. Yuan *et al.* [51] have also observed that developers often struggle to write correct *log messages* and require significant effort to modify logging statements later. To address these problems, a line of research has been conducted on the single subtasks in automatic logging, such as determining the logging positions [24, 49, 55], setting verbosity levels [21, 25, 28], and constructing log messages [11, 15, 23].

Although previous logging methods have shown promising results across various subtasks, they do not consider the logging tasks as a whole. Thus, these existing approaches have limited application scenarios. For example, LogAdvisor [55] can provide suggestions on whether to log in a code snippet but it only works for two kinds of codes: exception snippets and return-value-check snippets. Moreover, LogAdvisor cannot suggest suitable verbosity levels and log messages. DeepLV [25] can predict verbosity levels. However, it requires both the surrounding code and the log message as input, which assumes that developers already know the logging position and the log message.

Recently, Mastropaolo *et al.* [32] explored an all-in-one solution LANCE for logging by fine-tuning a large language model, T5 [37]. However, LANCE involves two phases of model tuning, namely multi-task pre-training and fine-tuning, before generating logging statements; both phases require considerable computational resources (more than 6M code snippets for tuning) and a large amount of time, leading to its limited practical benefits. As revealed by an empirical study, the benefits of software approaches are often constrained due to the computational cost in the training stage [14]. In addition, fine-tuning an LLM trained on natural language might achieve inferior results for logging because logging requires knowledge of programming language, and inconsistent inputs and objectives between pre-training and fine-tuning often lead to sub-optimal results for downstream tasks [42]. Therefore, we believe existing approaches are of limited practical relevance and an end-to-end, lightweight, and code-related logging solution is highly in demand.

To this end, this paper proposes UniLog, an automatic, end-to-end logging framework. UniLog adopts the in-context learning (ICL) paradigm [12], which allows LLMs to directly generate the expected answers by analogizing the related examples in the prompt demonstration. Thus, UniLog can learn the common patterns of code snippets in a prompt and finish the logging process (*i.e.*, simultaneously determine logging positions, predict verbosity levels, and generate log messages). Specifically, UniLog employs Codex [7], a large language model (LLM) pre-trained on multiple programming language tasks, as the backbone. Without model tuning, UniLog can directly generate an appropriate logging statement with only five ordered examples structured in a specially designed line-aware prompt. Moreover, UniLog can further leverage a warmup mechanism that enhances LLM's ICL ability to improve the logging performance with only 500 random training samples (*i.e.*, 0.07% of LANCE's training data [32]). UniLog has been evaluated on a large dataset containing 12,012 code snippets extracted from 1,465 GitHub repositories. The results show that UniLog achieves (1) 76.9% accuracy on logging position selection, (2) 72.3% accuracy on verbosity level prediction, and (3) 27.1 BLEU-4 score on log message generation, all exhibiting SOTA performance compared with existing approaches. Meanwhile, UniLog with warmup only costs at most 20 minutes for model adjustment, which is less than 4% of the time required for Codex with fine-tuning (*i.e.*, more than 10 hours).

This paper makes the following main contributions:

- It proposes UniLog, the first general logging framework that exploits the in-context learning ability of an LLM and enables fast deployment without large tuning costs.
- It designs a line-aware prompt format that explicitly provides the line information of code snippets to an LLM.
- It presents the evaluation of UniLog on end-to-end logging using 12,012 code snippets. The results show that UniLog

achieves the SOTA performance and outperforms all existing logging tools.

2 BACKGROUND AND MOTIVATION

In this section, we will first introduce the overview of the current research on logging statement generation tasks, followed by the technical background of large language model (LLM) and in-context learning (ICL), and explain how ICL paradigm, as compared to fine-tuning (FT) paradigm, can significantly improve the practicality of logging methods based on LLMs.

2.1 Logging Statement Generation

Logging, which involves generating appropriate log messages and verbosity levels at suitable positions in code snippets, is a classic problem in software engineering [16]. Many researchers have invested much effort in logging to assist developers in constructing better logging statements during program design, thereby facilitating more efficient software maintenance and testing in the future. For a long time, studies on logging have been limited to solving sub-problems partially under strong assumptions. For example, Li *et al.* [25] proposed *DeepLV* to recommend verbosity levels based on existing log messages, Liu *et al.* [28] proposed *TeLL* to further leverage flow graphs to determine verbosity levels, Zhu *et al.* [55] proposed *LogAdvisor* to help judge whether developers should add logging statements in a specific position, and Ding *et al.* [11] proposed *LoGenText* to provide logging text suggestions by summarizing the target code snippets. However, none of them can solve the problems as a whole (*i.e.*, solve (1) determining logging position, (2) generating log message, and (3) setting verbosity level, simultaneously), resulting in less practicality in real-world scenarios. Recently, Mastropaolo *et al.* [32] proposed *LANCE*, which tried to solve the logging problem in an end-to-end manner by using LLM. However, due to the significant training cost and the objective gap between pre-training and fine-tuning, it still suffers limitations from the practice perspective as demonstrated in Sec. 1. Therefore, we believe that an end-to-end, lightweight, and code-related automatic logging method is required to address logging tasks in a practical way.

2.2 Large Language Model

A large language model (LLM) is a language model consisting of a neural network with many parameters (typically billions of weights or more) trained on large quantities of the unlabelled corpus using self-supervised learning [45]. The large language models usually adopt the Transformer [41] architecture or its sub-structures (*i.e.*, *encoder* or *decoder*), and typical LLMs include BERT, GPT-1,2,3, T5, *etc.* For example, BERT [9] is a masked language model in the encoder structure and performs great in many natural language understanding tasks, and GPT-3 [3] is an autoregressive language model that uses the decoder structure and shows surprising performance in text generation tasks.

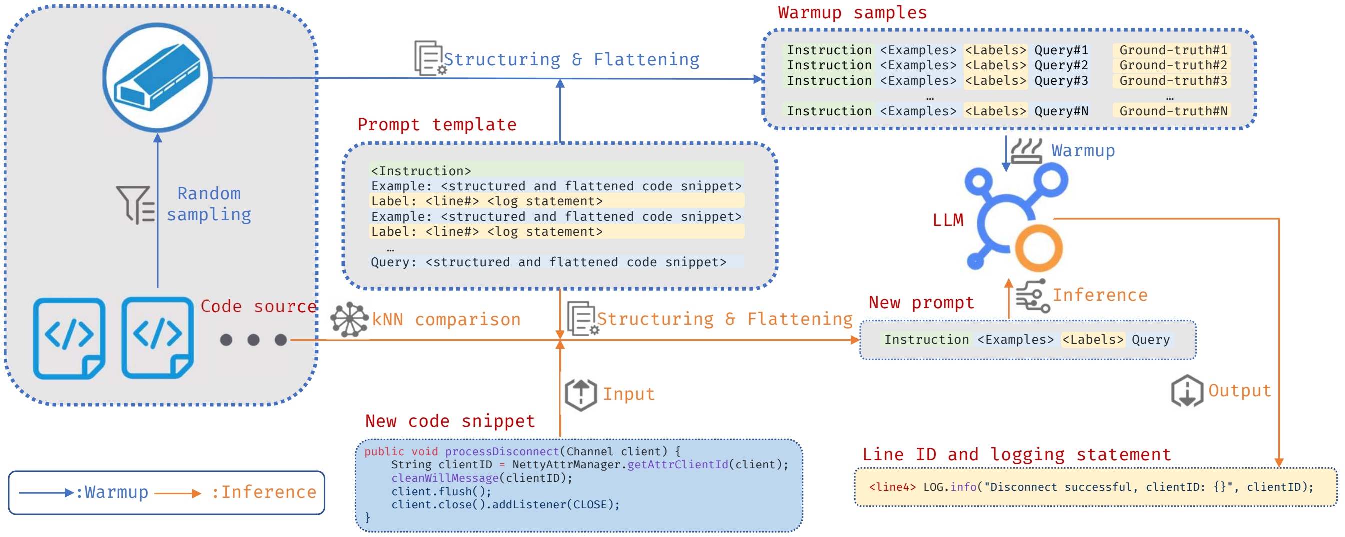
Since the advent of LLMs, natural language processing has evolved into a new learning paradigm, namely, *pre-training & fine-tuning*. In this paradigm, instead of training an LLM from scratch [3, 36, 37] in a highly costly way, people opt to fine-tune a pre-trained LLM with task-specific data in different downstream scenarios. Compared to pre-training, fine-tuning requires much fewer computation resources and thereby is more cost-saving. Specifically, in the software engineering area, LLMs [3, 7, 43] together with the fine-tuning techniques have made tremendous achievements in tasks such as code review generation [22], fuzzing test case generation [8], automated program repair [47, 48] and root cause analysis [1].

2.3 In-Context Learning

With the significant increase in parameter scale and corpus size, LLMs have become more and more powerful. However, it also brings higher requirements on the computation resources for model fine-tuning. For example, the GPT-3 Davinci model requires at least several tens of A100 GPUs in a data center for fine-tuning. Meanwhile, researchers [29] found that the capability of LLMs could be unleashed with the help of a textual prompt. For example, when identifying the sentiment for the sentence "I love the movie.", we can append a prompt "Overall it was a $[Z]$ movie." and ask the LLM to fill the blank $[Z]$ with an emotion-bearing word.

Furthermore, the prompt can be designed in a complex way. Recently, LLMs demonstrate new abilities that learn from the demonstration consisting of a few examples in the prompt (in-context learning for short). Specifically, in the ICL paradigm, the prompt input to the LLM usually contains (1) an instruction to specify the specific task, (2) several examples containing ground truth to provide task-specific knowledge, and (3) a query to the LLM with the expectation of an appropriate answer. For instance, Fig. 2 illustrates the basic workflow of in-context learning on sentiment classification. Multiple studies have demonstrated that LLMs perform well on various complex problems under the ICL paradigm, such as fact retrieval [54] and mathematical reasoning [44]. People typically believe that the essence of ICL is that LLMs have gained the ability of complex reason from a large amount of pre-training data, which enables LLMs to analog the expected answer of the query based on the demonstration examples in the prompt [12].

As a new paradigm of LLMs, ICL has numerous inherent advantages. *First*, since ICL does not require network structures exploration (*i.e.*, architecture engineering) or tuning object design (*i.e.*, objective engineering), the LLMs can be adapted to new tasks at low computational costs with the task-specific prompt. *Second*, since all instructions and descriptions are in natural language, the interaction process between users and LLMs is interpretable and easy to understand. *Third*, since ICL directly leverages the knowledge obtained from pre-training along with the information in the prompt demonstration, there is no significant gap between pre-training and downstream tasks. This approach avoids the issues of overfitting in fine-tuning due to over-reliance on specific downstream task data, and enables users to fully explore the knowledge of LLMs [42]. These advantages are well-aligned with the needs of the software engineering (*e.g.*, fast deployment and easy interactivity as discussed in Sec. 1). After considering the significant advantage of ICL, we proposed UniLog, the first attempt to complete the logging statement generation task in the ICL paradigm. Specifically, UniLog no longer needs to explicitly learn logging from a training process that consumes a large amount of log data. Instead, it leverages the analogical reasoning ability of the LLMs to directly infer the expected logging statement from code examples provided in the



```
Instruction <Examples> <Labels> Query#1    Ground-truth#1
Instruction <Examples> <Labels> Query#2    Ground-truth#2
Instruction <Examples> <Labels> Query#3    Ground-truth#3
                    …                            …
Instruction <Examples> <Labels> Query#N    Ground-truth#N
```

```
<Instruction>
Example: <structured and flattened code snippet>
Label: <line#> <log statement>
Example: <structured and flattened code snippet>
Label: <line#> <log statement>
  …
Query: <structured and flattened code snippet>
```

```
Instruction <Examples> <Labels> Query
```

```
public void processDisconnect(Channel client) {
    String clientID = NettyAttrManager.getAttrClientId(client);
    cleanWillMessage(clientID);
    client.flush();
    client.close().addListener(CLOSE);
}
```

```
<line4> LOG.info("Disconnect successful, clientID: {}", clientID);
```

Figure 3: The basic workflow of UniLog framework. The definition of warmup is introduced in Sec. 3.3. The blue arrows represent the processes in warmup stage while the yellow arrows represent the processes in inference stage.

prompt. We will provide experimental evidence to demonstrate the accuracy and efficiency advantage of UniLog in Sec.4.

3 UNILOG

Our UniLog builds on the in-context learning paradigm with the large language model, Codex [7]. It aims to generate appropriate verbosity levels and log messages for target code snippets at appropriate positions, based on a small number of labeled logging statement examples through analogy. Specifically, when presented with a new code snippet, UniLog extracts the top-5 most similar examples from the training set using kNN [27], sorts them in ascending order, assembles them into a line-aware prompt, and inputs it into Codex for inference. By analogizing the examples provided in the prompt, UniLog outputs the corresponding logging statement for the given query and indicates the code line ID where the statement should be inserted. The workflow above assumes excluding any model parameter tuning. However, if using merely 500 randomly selected prompt samples from the training set for model warmup, the analogical reasoning ability of UniLog on logging tasks can be further enhanced significantly. The workflow of UniLog is shown in Fig. 3 In the following, we will introduce the design details of the backbone (Sec. 3.1), prompt (Sec. 3.2), and the optional warmup (Sec. 3.3).

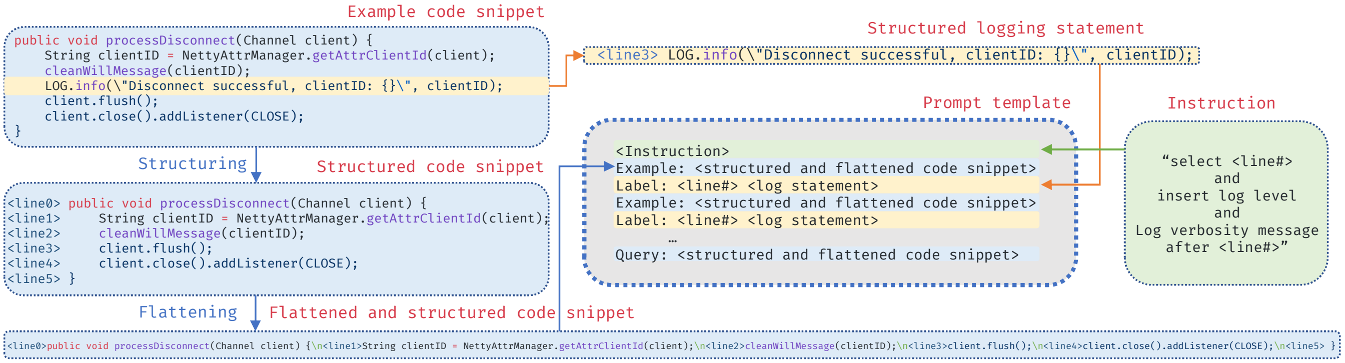
3.1 Model Backbone

The performance of the large language model backbone is a fundamental aspect of in-context learning. Considering that logging statements contain both programming language tokens (*i.e.*, logger and verbosity level) and natural language tokens (*i.e.*, log message), we chose Codex [7], an LLM based on GPT-3 [3] that leverages a large number of repositories on GitHub to enhance the code generation ability, as the backbone for UniLog. We will not dive into the architecture details of Codex in this paper, but it is worth noting that there is no objective discrepancy between the pre-training stage and the inference stage of Codex, as it is pre-trained on programming language corpora and aims to solve code-related downstream tasks. By adopting Codex as the backbone, UniLog can take full advantage of the model's knowledge and mitigate performance degradation resulting from this discrepancy. Moreover, due to the significant parameter scale of Codex and the amount of data used in pre-training, UniLog can infer answers directly from the prompt without the need of model fine-tuning. As stated in Sec. 1, the properties of Codex well align the practicality requirements of logging tasks, which led us to choose it as the default backbone of UniLog.

In our implementation, UniLog uses the `cushman` version of Codex. Since UniLog utilizes the LLM in a black-box manner, the backbone can be replaced at will. For instance, UniLog can be extended to specialized code models (*e.g.*, CodeT5-6B or InCoder-6B), as well as to general dialogue models whose pre-training corpora consists of codes (*e.g.*, ChatGPT and GPT-4). Additionally, it is important to note that when working with closed-source models that do not provide fine-tuning APIs (*e.g.*, GPT-4), UniLog can not perform model warmup, which could potentially impact performance. For the sake of training objectives discussed in Sec. 1, we still recommend using code-related LLM as the backbone to avoid compromising logging effectiveness. For reference, we also provide comparative experiments about using different backbones in Sec. 4.4.

3.2 Prompt Strategy

Prompt strategy is the most crucial part of in-context learning. Designing an appropriate prompt requires consideration of many aspects, such as how to design instructions, how to design prompt templates, how to select in-context examples, and how to arrange the order of examples and queries. In UniLog, we particularly focus on the sensitivity of the inserted target line for the logging task and design a line-aware prompt template as shown in Fig. 4. Additionally, we utilized special example selection and permutation



```
public void processDisconnect(Channel client) {
    String clientID = NettyAttrManager.getAttrClientId(client);
    cleanWillMessage(clientID);
    LOG.info(\"Disconnect successful, clientID: {}\", clientID);
    client.flush();
    client.close().addListener(CLOSE);
}
```

```
<line3> LOG.info(\"Disconnect successful, clientID: {}\", clientID);
```

```
<line0> public void processDisconnect(Channel client) {
<line1>     String clientID = NettyAttrManager.getAttrClientId(client);
<line2>     cleanWillMessage(clientID);
<line3>     client.flush();
<line4>     client.close().addListener(CLOSE);
<line5> }
```

```
<Instruction>
Example: <structured and flattened code snippet>
Label: <line#> <log statement>
Example: <structured and flattened code snippet>
Label: <line#> <log statement>
    …
Query: <structured and flattened code snippet>
```

```
"select <line#>
and
insert log level
and
Log verbosity message
after <line#>"
```

```
<line0>public void processDisconnect(Channel client) {\n<line1>String clientID = NettyAttrManager.getAttrClientId(client);\n<line2>cleanWillMessage(clientID);\n<line3>client.flush();\n<line4>client.close().addListener(CLOSE);\n<line5> }
```

Figure 4: The prompt templates, using the code block in Fig. 1 as an example. UniLog includes several labeled examples and one query in each prompt. The last example is most similar to the query, while the first example is least similar to the query.

methods based on the similarity between the target code snippet and candidate examples. These strategies enable LLM to complete the logging statement generation task in a more targeted manner.

3.2.1 *Prompt Format.* Before designing the prompt template, there is a question that must be considered first: *What form should we model logging statement generation in, to enable LLMs to understand it?* Although most LLMs' pre-training objectives include text generation tasks, they do not explicitly consider where the generated text should be inserted into. In order to enable LLMs to complete the "*where-to-log*" task, an intuitive solution is to transform the logging task into a code completion task: predict whether a logging statement should be generated after each line of code. However, this paradigm would lead to great inefficiency: if the target code snippet has N lines, the model would need to perform N times of inferences, in which the query contains the first N lines of the original code snippet. This computational inefficiency, which was mentioned in Sec. 1, greatly contradicts our original intention and is intolerable. Furthermore, there is a significant imbalance in the proportion of code lines between logging and non-logging statements, with the number of logging statements being far fewer than non-logging ones. For instance, Mastropalo *et al.* [32] pointed out that almost 96% of Java methods do not contain logging statements. The imbalance in these samples may result in a low recall rate, meaning that almost every line is more likely to be predicted as not requiring a logging statement, which could severely compromise our logging performance.

To ensure "one query, one inference" principle and sample equalization, we choose to explicitly represent the line information of the code snippet and transfer problem into a multi-choice problem. Specifically, we add a special token `<line#>` at each newline position to indicate the line ID, and then flatten the code snippet to a sequence format. Meanwhile, in the instruction, UniLog requires the model to provide both the generated logging statement and the target line ID to be inserted. This line-aware prompt format ensures both efficiency and accuracy in inference, especially in the position aspect. The prompt format is shown in Fig. 4

3.2.2 *Prompt Example.* Liu *et al.* [27] pointed out that the selection of examples used in the prompt significantly affects the downstream task performance of LLMs under ICL paradigm. Therefore, many researchers have put in a lot of effort to select better in-context examples to improve the ICL ability [27, 34, 38]. In UniLog, we choose *KATE* [27], a simple kNN-based sampling algorithm that does not involve much computational overhead in practice, for in-context example augmentation. Specifically, we begin by embedding all code snippet candidates x_i from training data into vector representations v_i. Then, for each vectorized query v_q, we calculate the similarity metric $d(v_q, v_i)$ between it and all candidates, outputting the top-5 results as examples. Note that in our implementation, the similarity metric $d(v, v_i)$ represents the cosine distance as shown in Eq. 1, and the overall algorithm can be summarized as shown in Algo. 1.

Algorithm 1 kNN In-Context Example Augmented Selection

```
Input: query prompt $x_q$, candidate set $C = \{x_i\}_{i=1}^{N}$, distance map
    $\mathcal{D} = \emptyset$, and code snippet encoder $\epsilon(\cdot)$
$v_q = \epsilon(x_q)$
for $x_i \in C$ do
    $v_i = \epsilon(x_i)$
    $d(v_q, v_i) = \cos(v_q, v_i)$
    add key-value pair $\{d(v_q, v_i) : x_i\}$ to $\mathcal{D}$
end for
sort the key of $\mathcal{D}$ in descending order
Output: top-5 value of $\mathcal{D}$
```

$$d(v_q, v_i) \coloneqq \cos(v_q, v_i) = \frac{v_q \cdot v_i}{\|v_q\|_2 \|v_i\|_2}, \tag{1}$$

Moreover, some studies [19, 31, 54] have also shown that the permutation of different examples in the context can also affect the performance of ICL seriously. For example, Zhao *et al.* [54] pointed out that the model's prediction for a query tends to be biased towards the closest example (*i.e.*, *recency bias*), which means if the example closest to the query in the prompt is similar enough to the query, the model's prediction for the query will tend towards the results closest to the query (*i.e.*, obtaining the correct prediction according to the nearest example's label supervision). Therefore, we choose to directly use the similarity measure d obtained in the previous step to arrange these examples in *ascending order*, so that the example closest to the query is most similar to the query.

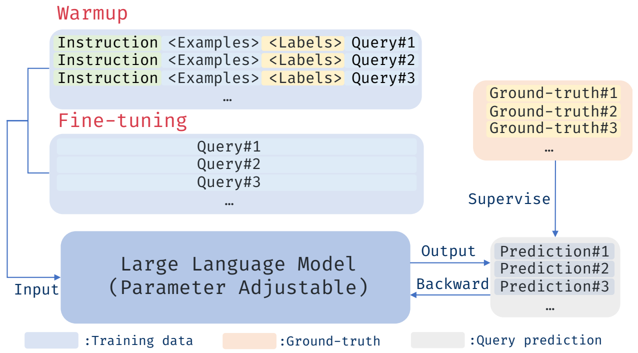


Figure 5: The difference between warmup and fine-tuning

The advantages of this permutation will be detailed in subsequent sections in the form of experimental results.

3.3 Warmup Strategy

While LLMs have demonstrated strong ICL capabilities in a train-free manner, many studies have found that the model's ICL abilities can be further improved by adding several prompt samples for additional tuning prior to ICL inference [6, 33]. This process is known as "warmup" [12]. Warmup serves a distinct purpose from fine-tuning: the latter improves the performance of LLMs on specific tasks by training with relevant samples, while the former enhances LLMs' general analogy ability to find common features of the limited contextual examples based on each instruction in the prompt. Fig. 5 shows the difference between warmup and fine-tuning process for a pre-trained model. Specifically, warmup uses complete prompts (shown in Fig. 4) as input, each comprising an instruction, several demonstration example, and a query, while fine-tuning uses code snippets without logging statements (*i.e.*, the queries in the complete prompt) for training. Moreover, warmup requires only a very small amount of training data, significantly less than data used in fine-tuning, which helps mitigate the risk of overfitting. Although Min *et al.* [33] and Chen *et al.* [6] have coincidentally pointed out that using multiple different tasks and instructions for model warmup is a better way to improve the ICL ability, it is not the best warmup strategy for UniLog. As a logging technique, UniLog does not necessitate general reasoning ability on multiple downstream tasks but rather concentrates solely on enhancing logging proficiency. Therefore, in our implementation, we set the instruction of all warmup samples to the same target (*i.e.*, logging statement generation) and set the labeled code snippets as prompt examples to improve the ICL ability in a task-specific manner. Thus, the entire warmup process is as follows: By default (Sec. 4), UniLog first randomly selects 500 samples from the validation set as the queries of the prompts for warmup. Then, for each query, UniLog adopts Algo. 1 to search for another five most similar samples in the training set as the prompt examples, attaches their ground-truth labels, and packs them together with fixed instructions into a complete prompt. After that, all of the prompts will be submitted to Codex for parameter tuning in batches. Considering that the query and example in the prompt may overlap, the actual number of consumed code snippets is less than 2,500, which is a negligible additional cost compared with the whole training set scale.

It should be noted that warmup is not a mandatory step in UniLog, and the number of samples used in warmup may affect the subsequent ICL ability of LLMs [12]. In order to optimize the performance of UniLog, we provide experiments in Sec. 4.4 to embody our consideration of the warmup strategy in detail.

4 EVALUATION

To comprehensively and systematically study the actual performance of UniLog with ICL paradigm in various aspects of logging statement generation, we set the following three research questions:

- **RQ1:** How does UniLog compare against existing LLM-based logging method?
- **RQ2:** How much improvement can ICL bring to logging compared to fine-tuning?
- **RQ3:** How do different ICL strategies affect logging effectiveness?

Specifically, RQ1 will investigate the extent to which UniLog can support and assist the logging statement generation task compared to LANCE [32]. RQ2 will further explore the advantages of ICL paradigm over fine-tuning paradigm when using Codex as the backbone to address logging problems. RQ3 will analyze the impact of different settings and strategies of UniLog on its effectiveness, which is mentioned previously in Sec. 3. We will also provide global guidance to users on migrating UniLog to their own scenarios by analyzing its performance on the specific dataset with different configurations.

4.1 Experiment Setup

4.1.1 Environment. In our experiments, we utilized HTTP requests to invoke the OpenAI APIs and interact with the LLMs and official embedding models (*i.e.*, `text-embedding-babbage-001`), including the warmup and inference processes of UniLog, as well as the fine-tuning and inference processes in other comparative experiments. Additionally, we employed Python 3.9 to implement prompt construction, the Algo. 1, and various metrics used in the evaluation.

4.1.2 Datasets. Our experimental data was sourced from the Java method dataset provided by Mastropaolo *et al.* [32], which was mined from a total of 1,465 GitHub open repositories. To ensure the data quality and mitigate the potential threats of data leakage in Codex's pre-training, we performed extra data cleaning and reformatting. Specifically, we got rid of the invalid code snippets (without logging statements) or incomplete code snippets. Then, we wrapped them in a fixed anonymous class (*i.e.*, `class A`) and used the `Google-Java-Format` tool [1] to rebuild their structure into a standard format. After that, the dataset contains totally 125,888 Java method samples, including 101,405 training samples, 12,470 validation samples, and 12,012 testing samples. Next, we used regex to extract all log statements from the original samples to build the complete dataset: (1) The extracted logging statements were marked with a line ID tag (*i.e.*, `<line#>`) at the prefix to indicate their position in the initial method. They served as ground truth labels for their respective methods. (2) The methods from which

[1] Google-Java-Format. https://github.com/google/google-java-format/

logging statements have been extracted will be flattened into single-line sequences after adding a line ID tag to the prefix of each line. They were used as input data for incoming model tuning. This entire process is consistent with the example shown in Fig. 4.

4.1.3 Metrics. Our evaluation methodology follows the same approach as previous studies [15, 25, 32], which assess the effectiveness of logging methods in terms of logging position, verbosity level, and log message. To be specific, we utilize three traditional metrics – position accuracy (PA), level accuracy (LA), and message accuracy (MA) – to evaluate the performance of UniLog on different sub-tasks. These metrics are also consistent with those adopted by LANCE [32], measuring the proportion of correctly predicted samples for each feature (*i.e.*, position, level, and message) out of the total number of samples. Furthermore, we introduce two new metrics, conditional level accuracy (CLA) and conditional message accuracy (CMA) to show the percentage of correctly predicted verbosity levels and log messages, respectively, in samples where the position prediction is accurate. These metrics are more relevant in real-world logging scenarios because adding an accurate logging statement at the wrong position may renders it meaningless.

In addition, logging statements are the combinations of programming language (PL) logger and natural language (NL) description. Therefore, we adopt BLEU [35], a commonly used evaluation metric based on n-gram feature in NLP, to further evaluate UniLog's performance on message-level. In our implementation, we uses the BLUE-DM variant [4, 40], *i.e.*, the sentence-level BLEU without any smoothing method. The calculation methods for BLEU-DM is detailed below as shown in Eq. 2.

$$\text{BLEU-DM} = BP \cdot \exp(\sum_{n=1}^{4} w_n \log p_n) \tag{2}$$

where w_n denotes the weight of each n-gram and p_n represents the precision of each n-gram, which can be calculated as Eq. 3

$$p_n = \frac{\#\text{n-grams in the reference}}{\#\text{n-grams in the candidate}} \tag{3}$$

BP is the penalty value parameter for short candidate texts, and its calculation method is shown in Eq. 4:

$$BP = \begin{cases} 1 & \text{if } c \geq r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases} \tag{4}$$

where r is the length of reference and c is the length of the candidate text.

Specifically, BLEU provides an automated and standardized approach for assessing textual similarity, which is widely used in machine translation. It is also a relatively simple metric that is insensitive to the word order in the sentences. These properties make BLEU more practical than exact match in evaluating generated text quality. However, BLEU primarily focuses on lexical matching, meaning it does not account for the semantic coherence or contextual accuracy of the generated log messages. It also ignores the synonyms or paraphrases in the message [52]. Thus, human evaluation remains invaluable in assessing the quality of generated log messages, and we conducted a qualitative analysis in Sec. 5.3.

Table 1: Accuracy comparison on different logging methods.

Method	PA	LA	MA	CLA	CMA	BLEU
LANCE-Full	60.2	60.4	13.7	73.2	21.4	N/A
LANCE-Best	65.4	66.2	16.9	76.8	24.3	N/A
UniLog-w/o warmup	66.5	66.8	20.2	75.7	28.7	24.5
UniLog-w/ warmup	76.9	72.3	22.4	77.9	28.1	27.1

4.1.4 Baselines.

We choose LANCE [32], the first method that utilizes LLM to generate complete logging statements, as the primary baseline for comparing the overall performance of UniLog on the logging task. Additionally, since LANCE is based on T5-Small [37], whose parameter scale is much smaller than that of UniLog's Codex, we also use several models from the GPT-3 series [2] for extra comparison to ensure experimental fairness, including Codex (*i.e.*, `cushman`) itself with fine-tuning paradigm, as well as GPT-3-A (*i.e.*, `ada`), GPT-3-B (*i.e.*, `babbage`), and GPT-3-C (*i.e.*, `curie`) with ICL paradigm. In particular, we do not choose to conduct experiments with `davinci`, which has the biggest parameter scale and the most powerful inference ability in the GPT-3 series, because of our limited experimental resources. Note that all GPT-3 and Codex have inherent randomness during inference process. Thus, we set their `temperature=0` to ensure consistent output for the same input [3]. Moreover, to mitigate the impact of randomness in demonstration selection, we conducted three separate experiments for each setting and calculated the mean value as the final result. We performed extra examinations on the repeated experiment results to ensure the stability of the results. After that, we found that the UniLog's results had minimal fluctuations, with all metrics controlled within a 5% range, illustrating acceptable randomness in the experiments.

4.2 RQ1: How does UniLog compare against existing LLM-based logging method?

To demonstrate the effectiveness of UniLog in the logging statement generation task, we adopt both versions of UniLog with and without warmup (*i.e.*, UniLog-w/ warmup and UniLog-w/o warmup) to compare with other logging methods that are specifically designed for logging tasks. Specifically, UniLog-w/ warmup uses 500 random complete prompts as training samples for model warmup, while UniLog-w/o warmup does not involve any parameter tuning before inference. However, as mentioned in Sec. 2.1, previous logging tools have significant usage restrictions and cannot directly generate complete logging statements. The only method that we can compare in an *apple-to-apple* manner is LANCE [32], the first logging tool based on LLM (*i.e.*, T5-Small [37]). Considering the difficulty of reproducing an LLM-based method requiring fine-tuning, we directly use the evaluation results reported in the original paper rather than re-train it again. Since the original paper did not adopt BLEU for evaluation, this metric is replaced by `N/A` in result tables. Additionally, LANCE has multiple pre-training and fine-tuning steps, and its performance varies significantly depending on the training strategies. Surprisingly, the best-performing version of

[2] https://platform.openai.com/docs/models/gpt-3
[3] https://platform.openai.com/docs/api-reference/completions

Table 2: Accuracy comparison on different paradigms.

Paradigm	PA	LA	MA	CLA	CMA	BLEU
Codex-ZeroShot	N/A	N/A	N/A	65.7	9.8	13.2
Codex-FT	86.9	70.0	8.1	71.7	9.3	10.8
Codex-ICL	76.9	72.3	22.4	77.9	28.1	27.1

Table 3: Tuning cost comparison on different paradigms.

Paradigm	Running Time (s)	Training Data
Fine-Tuning	37,021	113,875
In-Context Learning	1,178	500
ICL/FT	3.17%	4.39%

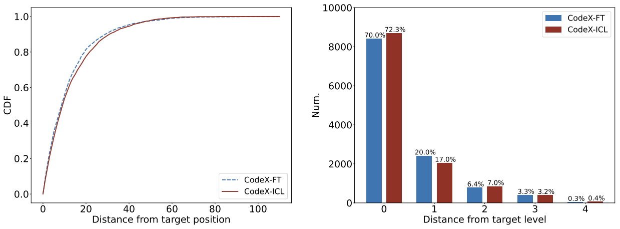


Figure 6: Distance of target position (left) and level (right).

LANCE is not the full version as presented in the original paper (*i.e.*, LANCE-Full). Therefore, in addition to the full version of LANCE, we also use the best-performing version's results of LANCE (*i.e.*, LANCE-Best) from the original paper for comparison.

The results are presented in Table 1, demonstrating that UniLog outperforms LANCE without model tuning in all metrics. Moreover, by applying model warmup, UniLog further improves its logging performance significantly. The effectiveness advantages of UniLog over LANCE may be attributed to the following aspects.

First, UniLog's ICL paradigm may be more suitable for logging. Specifically, UniLog designs unique prompts for each query based on their characteristics, selecting similar code snippets as demonstrations, while LANCE performs general fine-tuning on all code snippets without differentiation. Moreover, fine-tuning on a large amount of data may pose risks of overfitting, whereas ICL requires little data, resulting in potentially stronger generalization ability in common logging scenarios.

Second, UniLog's task modeling strategy may be more appropriate for logging. Specifically, UniLog models the logging task as a single-line prediction task, generating solely "`<LineID> <Logging Statement>`" as output, whereas LANCE models the task as generating complete code snippets with logging statements. Thus, UniLog's output can be seamlessly inserted into the original code snippet without affecting other parts of the code. However, LANCE generates complete code snippets with logging statements, resulting in a risk of altering other non-logging code parts and causing syntax errors. This issue is also reported in the paper [32], where 3%-5% of the results contain syntax errors.

Third, UniLog employs a code-related model as the backbone, while LANCE uses T5-small, which is not pretrained on a large corpus of code. As mentioned in Sec. 1, logging involves both natural language and programming language, so relying solely on a model trained on natural language may result in suboptimal performance.

However, due to the backbone of LANCE, T5-Small, has a much smaller model scale than the backbone of UniLog, Codex, we cannot determine whether the advantage of UniLog is due to the superiority of Codex over T5-Small, or the superiority of the ICL paradigm over the fine-tuning (FT) paradigm. To investigate how much ICL can enhance the logging ability compared with FT, we further investigated RQ2 and provided a detailed experimental explanation.

4.3 RQ2: How much improvement can ICL bring to logging compared to fine-tuning?

To demonstrate the advantages of UniLog stems from the superiority of ICL paradigm, we fine-tuned Codex and compared it (*i.e.*, Codex-FT) with UniLog (*i.e.*, Codex-ICL). Specifically, for Codex-ICL, we randomly selected 500 candidates and constructed prompts for model warmup; for Codex-FT, we use 113,875 samples (all code snippets from training and validation set) for supervised training. We also attempted to directly use Codex without fine-tuning as a blank control. However, we found that without explicit supervised training or demonstration guidance, Codex cannot generate line IDs stably, and thus unable to complete logging in an end-to-end manner. Therefore, we can only evaluate Codex's zero-shot ability by providing correct logging positions, which obtain an additional reference for our experimental results in terms of CLA and CMA. As a result, its PA/LA/MA are all marked as `N/A`.

The experimental results are shown in Table.2, which demonstrate that Codex-ICL outperforms Codex-FT with a MA advantage 14.3% and a BLEU advantage of 16.3. These results clearly indicate the superiority of Codex-ICL in log message generation. However, we observed that Codex-ICL did not achieve a significant advantage in simpler tasks, *i.e.*, position prediction and level prediction. As a result, we conducted further experiments to investigate this phenomenon.

Analysis on logging position prediction Although Codex-FT exhibits 10% higher PA, the metric itself has limited practical guidance because moving a logging statement several lines does not make a significant difference in most real-life scenarios (*e.g.*, Fig. 1). Thus, we utilize *position distance* to further evaluate the line intervals between the prediction and the ground-truth. For instance, if the predicted line ID is `<line4>` and the ground-truth line ID is `<line3>`, the position distance is 1. As shown in Fig. 6, the distribution of position distance of error predictions on position (*i.e.*, with non-zero position distance) is represented by the cumulative distribution function (CDF). It illustrates that there is no significant difference in distribution between Codex-ICL and Codex-FT. Specifically, both of them have nearly 40% of samples where the predicted position is within 10 lines of the ground truth position, and nearly 90% of incorrect samples have position distances within 40 lines. Moreover, Table. 2 reveals that Codex-FT has significantly lower CLA and CMA compared to Codex-ICL when predicting accurate positions (*i.e.*, lower by 6.2% and 18.8% accuracy, respectively). This indicates that some of the correctly predicted positions in Codex-FT

Table 4: Accuracy comparison on different backbones.

Backbone	PA	LA	MA	CLA	CMA	BLEU
Codex	76.9	72.3	22.4	77.9	28.1	27.1
GPT-3-A	43.6	53.2	9.1	58.4	16.9	11.9
GPT-3-B	44.3	55.7	9.8	63.8	17.7	12.6
GPT-3-C	49.5	58.4	11.2	67.1	19.4	14.5

have little practical significance. Therefore, we believe that these two paradigms have little practical gap in position prediction.

Analysis on verbosity level prediction There are five logging verbosity levels in the dataset: *Fatal*, *Error*, *Warn*, *Info*, and *Debug*. These levels serve different purposes. For instance, *Fatal* and *Error* deal with serious issues and generate error messages, while *Info* and *Debug* provide ignorable or trivial information in most cases. However, LA is inadequate for level evaluation because predicting *Debug* as *Info* or as *Fatal* can have vastly different impacts in practice. Therefore, we introduce *level distance* to fine-grained quantify the gap between the predicted level and the ground-truth level. For example, if the predicted result is *Debug* and the ground-truth is also *Debug*, then the level distance is 0; if the predicted result is *Debug* and the ground-truth is *Error*, then the level distance is 3 because there are three levels between them. As shown in Fig. 6, the level distance distribution is similar between Codex-ICL and Codex-FT. For example, both of the major predictions are only one level from the ground-truth (about 90.0%). However, Codex-ICL still yields a higher proportion of exactly matched predictions (*i.e.*, LA). Therefore, we believe that the ICL-based method has a slight advantage in level prediction.

Additionally, we studied the *computational costs* of model tuning for both paradigms. Specifically, we conducted five experiments for fine-tuning and warmup for each paradigm and took the average training time and data consumption. The results are shown in Table 3, where the time cost by Codex-ICL is less than 4% of Codex-FT, and the data consumption is less than 5% of Codex-FT. This illustrates that ICL paradigm is a more lightweight and economical choice for logging methods.

4.4 RQ3: How do different ICL strategies affect logging effectiveness?

This section analyzes the impact of different configurations on four logging effectiveness aspects: (1) types of model backbone, (2) number of prompt examples, (3) permutation methods for prompt examples, and (4) number of warmup samples. By default, UniLog uses Codex as the model backbone, conducts model warmup with 500 random samples before inference, selects 5 most similar code snippets as prompt examples in each inference, and sorts examples in ascending order of query similarity in each prompt. For each subsequent experiment, only one setting is modified to control variables. We aim to provide users with global guidance for migrating UniLog to their specific scenarios by analyzing its performance on the LANCE dataset when these optional configurations are changed in the UniLog framework.

Table 5: Accuracy comparison on prompt example numbers.

Example Num.	PA	LA	MA	CLA	CMA	BLEU
1	76.4	69.1	20.0	74.6	25.3	24.4
3	76.8	71.4	21.6	77.0	27.2	26.3
5	76.9	72.3	22.4	77.9	28.1	27.1
10	77.1	73.3	22.4	78.8	28.3	26.9
15	77.6	73.7	22.5	78.6	28.0	27.1

Table 6: Accuracy comparison on prompt permutations.

Permutation	PA	LA	MA	CLA	CMA	BLEU
random	74.8	64.6	8.8	68.7	11.3	12.3
*k*NN-descend	76.5	71.2	21.6	76.6	27.2	26.0
*k*NN-ascend	76.9	72.3	22.4	77.9	28.1	27.1

Table 7: Accuracy comparison on warmup sample numbers.

Sample Num.	PA	LA	MA	CLA	CMA	BLEU
100	66.6	64.5	18.8	73.3	26.4	22.5
300	76.2	69.7	21.1	75.8	26.7	25.2
500	76.9	72.3	22.4	77.9	28.1	27.1
700	77.8	73.6	22.4	78.2	28.0	26.9
900	76.1	73.0	22.7	78.9	28.8	27.2

4.4.1 Model Backbone Type. As discussed in Sec. 1, simply using general-purpose LLMs can only achieve suboptimal performance because logging involves both natural language and programming language. Thus, we compare Codex with other LLM that pre-trained on natural language to evaluate the impact of backbone types. To ensure that ICL paradigm can be utilized, the size of LLMs must be large enough to be guaranteed [3]. Therefore, in the experiment, we will not consider LLMs with relatively small scales, such as T5-Small [37], the backbone of LANCE. To ensure fairness at the scale (*i.e.*, >100M) and structure (*i.e.*, decoder-only), we chose models from the GPT-3 series as backbones, namely GPT-3-A, GPT-3-B, GPT-3-C, and Codex.

The experimental results are shown in Table 4. It can be seen that UniLog has a relatively large accuracy advantage when using Codex as the backbone compared to other LLMs. For example, in terms of logging position, Codex achieves about 30% higher accuracy compared with the GPT-3 series, and for the BLEU, it is even more than twice of others. The results indicate that the use of a code-related model as the backbone greatly influences UniLog’s performance, while simply increasing the parameters of a general-purpose LLM does not effectively enhance logging capability.

4.4.2 Prompt Example Number. To investigate the impact of prompt example numbers, we considered five hierarchies of example numbers for comparative experiments: 1, 3, 5, 10, and 15. In particular, we observed that the effectiveness metrics do not exhibit significant improvement when the number of examples exceeds 15. Consequently, we choose not to include these results in the table.

The experimental results are shown in Table 5. As the number of prompt examples increases, all metrics gradually improve. However, as the number of prompt examples exceeds 5, the growth rate of metrics gradually slows down, and a trend of saturation emerges. For example, when the number of prompt examples increased from 5 to 10, BLEU decreased slightly; when it increased to 15, its BLEU remained the same as when the number was 5. In addition, more prompt examples may lead to heavier model computational consumption and slower inference speed, resulting in practical inefficiency. Therefore, we ultimately designed UniLog with only five prompt examples.

4.4.3 Prompt Example Permutation. To analyze the effect of the permutation method of warmup sample, we select three main methods for comparative experiments, namely (1) randomly selecting and then randomly sorting (*i.e.*, random), (2) selecting based on *k*NN similarity according to Algo. 1 and then sorting in ascending order (*i.e.*, *k*NN-ascend), and (3) selecting based on similarity and then sorting in descending order (*i.e.*, *k*NN-descend).

The experimental results are shown in Table 6. We found that *k*NN-ascend outperforms the other two selection strategies in all metrics, indicating its indispensable role in the UniLog prompt. This result is in line with the observation made by Zhao *et al.* [54] that LLMs exhibit significant induction bias (*i.e.*, *recency bias*) towards characteristics and labels of examples that are closest to the query. Based on this result, we adopted *k*NN-ascend for example selection in UniLog. We also suggest considering arranging prompt examples in ascending order of similarity as a universal prompt strategy.

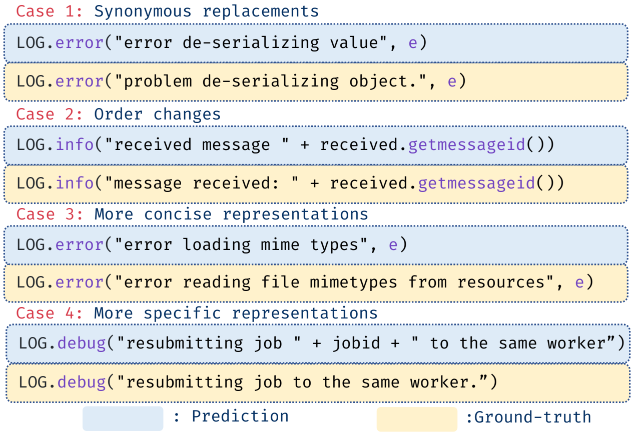
4.4.4 Warmup Sample Number. To find out the impact of warmup sample numbers, we considered five hierarchies of sample numbers for the comparative experiment: 100, 300, 500, 700, and 900. In particular, we observed that the effectiveness metrics do not exhibit significant improvement when the number of samples exceeds 900. Consequently, we choose not to include these results in the table.

The experimental results are shown in Table 7. When the number of samples is greater than 500, overall metrics no longer improved significantly, and some metrics even showed declines, which indicates that the effectiveness improvement of UniLog through warmup is limited and tends to saturate. For example, when the sample number increased from 500 to 700, the MA did not increase, and both CMA and BLEU decreased slightly. In addition, the training overhead gradually increased along with the sample number, which no longer met the fast deployment requirement as illustrated in Sec. 1. Therefore, taking into account the computational cost, we ultimately designed UniLog with only 500 samples.

5 DISCUSSION

5.1 Threats to Vadility

- **Data leakage:** UniLog uses Codex as the backbone, which is pre-trained on GitHub repositories up until May 2020, while our data was collected from GitHub in 2022. This means that the repositories that remain unchanged after 2020 in the dataset may cause data leakage. To mitigate this threat, we performed the reformatting on all collected raw data to extract them from the original contexts and rebuild their structure, which is discussed in Sec. 4.1.2.
- **Randomness:** Randomness may affect the performance in two aspects: (1) the randomness in LLMs inference and (2) the randomness introduced in a random selection of prompt demonstrations and warmup samples in RQ3. To mitigate the former threat, we set the model temperature to 0, making LLMs always generate consistent outputs for the same input text. To mitigate the latter threat, we performed three separate experiments for each setting, averaged the results as the final result, and checked the stability of results as discussed in Sec. 4.1.4.



```
Case 1: Synonymous replacements
LOG.error("error de-serializing value", e)
LOG.error("problem de-serializing object.", e)
Case 2: Order changes
LOG.info("received message " + received.getmessageid())
LOG.info("message received: " + received.getmessageid())
Case 3: More concise representations
LOG.error("error loading mime types", e)
LOG.error("error reading file mimetypes from resources", e)
Case 4: More specific representations
LOG.debug("resubmitting job " + jobid + " to the same worker")
LOG.debug("resubmitting job to the same worker.")
```

Figure 7: Examples of log messages with similar semantics

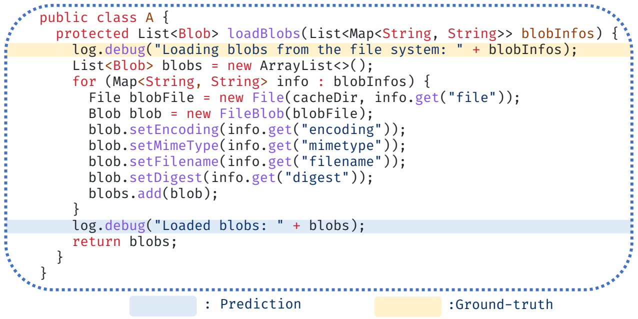
5.2 Ethical Issues

UniLog may encounter ethical issues associated with LLM usage [7], namely (1) Using third-party LLM models as a backbone may result in privacy leakage on codes or logs. (2) Utilizing codes containing stereotypes or racial biases as candidates may induce generating harmful log messages. To mitigate these issues, we recommend users employ trusted LLMs as backbones and filter out harmful code candidates carefully. To explore whether UniLog has this issue, we randomly selected 500 results and manually inspect them, finding that UniLog did not produce any harmful content among these samples. Meanwhile, we think it is important to systematically explore the potential ethical issues in LLM-based logging, and we regard it as important future work.

5.3 Error Analysis

As discussed in Sec. 4.1.3, BLEU may face limitations on log message evaluation. To provide more guidance of LLM-based logging for developers, we randomly selected and manually checked 500 log messages from the results that failed in exact match for qualitative analysis. Specifically, we categorized error messages into following three cases: (1) messages with similar semantics, (2) messages with different records, and (3) messages with meaningless information.

Messages with similar semantics: Among the sampled error messages, 217 error messages (43.4%) own similar semantics



```
public class A {
  protected List<Blob> loadBlobs(List<Map<String, String>> blobInfos) {
    log.debug("Loading blobs from the file system: " + blobInfos);
    List<Blob> blobs = new ArrayList<>();
    for (Map<String, String> info : blobInfos) {
      File blobFile = new File(cacheDir, info.get("file"));
      Blob blob = new FileBlob(blobFile);
      blob.setEncoding(info.get("encoding"));
      blob.setMimeType(info.get("mimetype"));
      blob.setFilename(info.get("filename"));
      blob.setDigest(info.get("digest"));
      blobs.add(blob);
    }
    log.debug("Loaded blobs: " + blobs);
    return blobs;
  }
}
```

Figure 8: An example of log message with different records

to the ground truth but differ in their expressions. Specifically, these semantic identical messages were deemed erroneous based on the exact match criterion due to the synonymous replacements, changes in word order, or the inclusion of more concise or detailed descriptions. As shown in Fig. 7, although the expressions are not exactly identical, they share similar semantics. Thus, we believe it may not have a significant impact during development.

Messages with different records: Among the error log messages we extracted, 116 error messages (23.2%) involved incorrect position prediction, resulting in the recording of entirely different variables or descriptions compared to the ground truth. However, we found these messages may still be useful. For example, in Fig. 8, UniLog records the blob's status after loading, while the ground-truth records the information that will be imported to blob before loading. To further enhance the logging ability at the correct position, we suggest introducing more warmup samples with diverse logging positions, thereby enhancing the LLM's sensitivity to the logging position.

Messages with meaningless information: We identified that 167 error messages (33.4%) recoded useless variables or provided completely meaningless descriptions. Upon analyzing their prompt inputs, we found it is due to the significant difference in coding style between the demonstrations and queries. The divergence in coding style can impede UniLog's effective learning of logging through ICL inference and analogy with the demonstrations. To mitigate these issues, we suggest increasing the diversity of the candidate set, incorporating more code snippets with different coding styles to augment the quality of prompt demonstration for various queries.

6 CONCLUSION

In conclusion, this paper introduces UniLog, a novel logging framework that employs the in-context learning (ICL) paradigm of large language models (LLMs). Unlike most existing logging studies that focus on a single logging task, UniLog performs automated logging in an end-to-end manner, which includes selecting logging positions, predicting verbosity levels, and generating log messages. Moreover, UniLog does not require fine-tuning for the target code style before each usage. UniLog has been evaluated on 12,012 code snippets extracted from 1,465 GitHub repositories and achieved the SOTA accuracy in all logging tasks. Furthermore, UniLog requires less than 4% of the parameter tuning time needed by the SOTA approach. As the first work to adopt the ICL paradigm in logging, we believe UniLog can facilitate better logging practices and inspire the design of efficient frameworks in general log analysis studies.

7 DATA AVAILABILITY

Due to the confidential policy of the company, we will not release the source code. The implementation details can be found in Sec. 3 and the data are public available.

ACKNOWLEDGMENTS

This paper was supported by the National Natural Science Foundation of China (No. 62102340) and Shenzhen Science and Technology Program.

REFERENCES

[1] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. 2023. Recommending Root-Cause and Mitigation Steps for Cloud Incidents using Large Language Models. *ICSE* (2023).

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

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[4] Boxing Chen and Colin Cherry. 2014. A systematic comparison of smoothing techniques for sentence-level BLEU. In *Proceedings of the ninth workshop on statistical machine translation*. 362–367.

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

[6] Mingda Chen, Jingfei Du, Ramakanth Pasunuru, Todor Mihaylov, Srini Iyer, Veselin Stoyanov, and Zornitsa Kozareva. 2022. Improving In-Context Few-Shot Learning via Self-Supervised Training. *arXiv preprint arXiv:2205.01703* (2022).

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[8] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2022. Fuzzing Deep-Learning Libraries via Large Language Models. *arXiv preprint arXiv:2212.14834* (2022).

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[10] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. 2015. Log2: A cost-aware logging mechanism for performance diagnosis. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*. 139–150.

[11] Zishuo Ding, Heng Li, and Weiyi Shang. 2022. Logentext: Automatically generating logging texts using neural machine translation. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 349–360.

[12] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A Survey for In-context Learning. *arXiv preprint arXiv:2301.00234* (2022).

[13] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 1285–1298.

[14] Wei Fu and Tim Menzies. 2017. Easy over hard: A case study on deep learning. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 49–60.

[15] Pinjia He, Zhuangbin Chen, Shilin He, and Michael R Lyu. 2018. Characterizing the natural language descriptions in software logging statements. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 178–189.

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

[17] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. 2018. Identifying impactful service system problems via log analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software*

Engineering Conference and Symposium on the Foundations of Software Engineering. 60–70.

[18] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. 2016. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*. IEEE, 207–218.

[19] Sawan Kumar and Partha Talukdar. 2021. Reordering examples helps during priming-based few-shot learning. *arXiv preprint arXiv:2106.01751* (2021).

[20] Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E Hassan. 2020. A qualitative study of the benefits and costs of logging from developers' perspectives. *IEEE Transactions on Software Engineering* 47, 12 (2020), 2858–2873.

[21] Heng Li, Weiyi Shang, and Ahmed E Hassan. 2017. Which log level should developers choose for a new logging statement? *Empirical Software Engineering* 22 (2017), 1684–1716.

[22] Lingwei Li, Li Yang, Huaxi Jiang, Jun Yan, Tiejian Luo, Zihan Hua, Geng Liang, and Chun Zuo. 2022. AUGER: automatically generating review comments with pre-training models. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1009–1021.

[23] Zhenhao Li. 2020. Towards providing automated supports to developers on writing logging statements. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 198–201.

[24] Zhenhao Li, Tse-Hsun Chen, and Weiyi Shang. 2020. Where shall we log? studying and suggesting logging locations in code blocks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 361–372.

[25] Zhenhao Li, Heng Li, Tse-Hsun Chen, and Weiyi Shang. 2021. Deeplv: Suggesting log levels using ordinal based neural networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1461–1472.

[26] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log clustering based problem identification for online service systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 102–111.

[27] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2021. What Makes Good In-Context Examples for GPT-3? *arXiv preprint arXiv:2101.06804* (2021).

[28] Jiahao Liu, Jun Zeng, Xiang Wang, Kaihang Ji, and Zhenkai Liang. 2022. Tell: log level suggestions via modeling multi-level code block information. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 27–38.

[29] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.

[30] Jian-Guang Lou, Qiang Fu, Shenqi Yang, Ye Xu, and Jiang Li. 2010. Mining invariants from console logs for system problem detection. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*.

[31] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2021. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. *arXiv preprint arXiv:2104.08786* (2021).

[32] Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. 2022. Using deep learning to generate complete log statements. In *Proceedings of the 44th International Conference on Software Engineering*. 2279–2290.

[33] Sewon Min, Mike Lewis, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2021. Metaicl: Learning to learn in context. *arXiv preprint arXiv:2110.15943* (2021).

[34] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*.

[35] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[36] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[37] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21, 1, Article 140 (jan 2020), 67 pages.

[38] Ohad Rubin, Jonathan Herzig, and Jonathan Berant. 2021. Learning to retrieve prompts for in-context learning. *arXiv preprint arXiv:2112.08633* (2021).

[39] Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Brain Adams, Ahmed E Hassan, and Patrick Martin. 2013. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 402–411.

[40] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the evaluation of neural code summarization. In *Proceedings of the 44th International Conference on Software Engineering*. 1597–1608.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[42] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R Lyu. 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 382–394.

[43] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[44] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903* (2022).

[45] Wikipedia. 2023. Large Language Model. https://en.wikipedia.org/wiki/Large_language_model. [Online; accessed 19-March-2023].

[46] W Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham. 2011. Effective software fault localization using an RBF neural network. *IEEE Transactions on Reliability* 61, 1 (2011), 149–169.

[47] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. Revisiting the Plastic Surgery Hypothesis via Large Language Models. *arXiv preprint arXiv:2303.10494* (2023).

[48] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. *arXiv preprint arXiv:2301.13246* (2023).

[49] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*. 143–154.

[50] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be conservative: Enhancing failure diagnosis with proactive logging. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 293–306.

[51] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 102–112.

[52] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1385–1397.

[53] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. 2019. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 807–817.

[54] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate before use: Improving few-shot performance of language models. In *International Conference on Machine Learning*. PMLR, 12697–12706.

[55] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. 2015. Learning to log: Helping developers make informed logging decisions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 415–425.

[56] De-Qing Zou, Hao Qin, and Hai Jin. 2016. Uilog: Improving log-based fault diagnosis by log analysis. *Journal of computer science and technology* 31, 5 (2016), 1038–1052.