Go Static: Contextualized Logging Statement Generation

YICHEN LI, The Chinese University of Hong Kong, China
YINTONG HUO[*], The Chinese University of Hong Kong, China
RENYI ZHONG, The Chinese University of Hong Kong, China
ZHIHAN JIANG, The Chinese University of Hong Kong, China
JINYANG LIU, The Chinese University of Hong Kong, China
JUNJIE HUANG, The Chinese University of Hong Kong, China
JIAZHEN GU, The Chinese University of Hong Kong, China
PINJIA HE, The Chinese University of Hong Kong, China
MICHAEL R. LYU, The Chinese University of Hong Kong, China

Logging practices have been extensively investigated to assist developers in writing appropriate logging statements for documenting software behaviors. Although numerous automatic logging approaches have been proposed, their performance remains unsatisfactory due to the constraint of the single-method input, without informative programming context outside the method. Specifically, we identify three inherent limitations with single-method context: limited static scope of logging statements, inconsistent logging styles, and missing type information of logging variables.

To tackle these limitations, we propose SCLogger, the first contextualized logging statement generation approach with inter-method static contexts. First, SCLogger extracts inter-method contexts with static analysis to construct the *contextualized prompt* for language models to generate a tentative logging statement. The contextualized prompt consists of an extended static scope and sampled similar methods, ordered by the chain-of-thought (COT) strategy. Second, SCLogger refines the access of logging variables by formulating a new *refinement prompt* for language models, which incorporates detailed type information of variables in the tentative logging statement.

The evaluation results show that SCLogger surpasses the state-of-the-art approach by 8.7% in logging position accuracy, 32.1% in level accuracy, 19.6% in variable precision, and 138.4% in text BLEU-4 score. Furthermore, SCLogger consistently boosts the performance of logging statement generation across a range of large language models, thereby showcasing the generalizability of this approach.

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

Additional Key Words and Phrases: code generation, software maintenance, large language models

[*]Yintong Huo is the corresponding author.

Authors' addresses: Yichen Li, The Chinese University of Hong Kong, Hong Kong, China, ycli21@cse.cuhk.edu.hk; Yintong Huo, The Chinese University of Hong Kong, Hong Kong, China, ythuo@cse.cuhk.edu.hk; Renyi Zhong, The Chinese University of Hong Kong, Hong Kong, China, ryzhong22@cse.cuhk.edu.hk; Zhihan Jiang, The Chinese University of Hong Kong, Hong Kong, China, zhjiang22@cse.cuhk.edu.hk; Jinyang Liu, The Chinese University of Hong Kong, Hong Kong, China, jyliu@cse.cuhk.edu.hk; Junjie Huang, The Chinese University of Hong Kong, Hong Kong, China, junjayhuang@outlook.com; Jiazhen Gu, The Chinese University of Hong Kong, Hong Kong, China, jiazhengu@cuhk.edu.hk; Pinjia He, The Chinese University of Hong Kong, Shenzhen, China, hepinjia@cuhk.edu.cn; Michael R. Lyu, The Chinese University of Hong Kong, Hong Kong, China, lyu@cse.cuhk.edu.hk.

ACM 2994-970X/2024/7-ART28
https://doi.org/10.1145/3643754

ACM Reference Format:
Yichen Li, Yintong Huo, Renyi Zhong, Zhihan Jiang, Jinyang Liu, Junjie Huang, Jiazhen Gu, Pinjia He, and Michael R. Lyu. 2024. Go Static: Contextualized Logging Statement Generation. *Proc. ACM Softw. Eng.* 1, FSE, Article 28 (July 2024), 22 pages. https://doi.org/10.1145/3643754

1 INTRODUCTION

Logging practices have been widely studied since logs provide rich resources for software debugging and maintenance [Du et al. 2017; Liu et al. 2023; Yuan et al. 2012c; Zhang et al. 2017]. A logging statement is typically comprised of three components [Jiang et al. 2023a,b]: logging level, logging variables and logging text. The following shows an example of logging statement, where the terms *"info"*, *"service"*, and *"Entry to state for"* represent the logging level, logging variable, and logging text, respectively. Furthermore, the logging variable *"service"* is further utilized through the invocations of its member functions *getServiceState()* and *getName()*.

```
LOG.info("Entry to state " + service.getServiceState() + " for " + service.getName());
```

To facilitate developers writing logging statements, a number of works are proposed to build models for automated logging statement generation. These works focus on two aspects of logging, including generating the logging contents (i.e., *what-to-log*) and suggesting the logging positions (i.e., *where-to-log*) in the code context. A preliminary work [He et al. 2018] pointed out that logging statements in one project usually share similar patterns so that the logging history can be learned to generate new logging statements. Motivated by such patterns, existing logging generation approaches are mostly neural-based and supervised, learning logging patterns from historical data. These approaches can be further categorized into two types: discriminative and generative.

The idea of the discriminative logging approach is to develop deep-learning models for determining *single* component in a logging statement or predicting *whether* a logging statement should be contained in a piece of code. For example, DeepLV [Liu et al. 2022] suggests log levels by building a Bi-LSTM network to learn syntactic code features and log message features.

However, these discriminative logging models are restrained by their classification nature, which relies on a pre-defined set of classes and cannot generate complete logging statements. Inspired by trending language models, recent generative logging models overcome these limitations by considering the task of logging statement generation as the problem of text generation, which accepts code snippets and outputs the entire logging statement and its corresponding logging place. The pioneer logging statement generative approach, LANCE [Mastropaolo et al. 2022], employs a Text-to-Text Transfer Transformer (T5) mode [Raffel et al. 2020] to inject complete logging statements given a code snippet. The modern large language models (e.g., GPT-3.5) also show promising results in this task [Li et al. 2023b].

While generative models are investigated for better performance, their analysis scopes for logging statement generation are still outstretched: *When recommending logging statements for a specific method, existing approaches solely look into this method (or even a code block inside) while ignoring the programming context from other methods, not to mention from other files.* In particular, these models simply choose the method-level context (i.e., single-method) while ignoring the critical context outside the target method for inferring the logging position and corresponding logging statement. In the following, we present three significant inherent limitations of the previous single-method context and further discuss them with real-world examples in Sec. 2.

(1) *Limited static scope of logging statements.* Complete software integrates numerous interconnected methods, each of which serves as a function being called by others. The flow of execution across various methods offers a comprehensive overview of code functionalities in the entire system for developers. Relying on a limited single-method context, it is challenging to infer the

logging purpose and thus decrease the logging quality. Apart from execution flow, the available variables that are beyond the scope of the target method (e.g., attributes in the current class) are also indispensable. Without knowing the available variables of given method-level context, choosing logging variables beyond the method scope becomes nearly impossible.

(2) *Inconsistent logging styles.* In well-maintained software projects, consistent logging styles are crucial [Rong et al. 2018, 2020, 2017; Yuan et al. 2012a] for ensuring log readability [Li et al. 2023a] and the coherence of logs. Such consistency encompasses maintaining coherent logging levels for component lifecycles [Liu et al. 2022], choosing appropriate words [Li et al. 2023a], and using accordant log text separators. Previous works [Ding et al. 2022, 2023a] have shown similar code can provide additional information on the syntactic structure of logging text and logging pattern. However, in the method-level context, learning the logging style of a specific project becomes challenging without in-project adaptation. Relying solely on the general knowledge that pre-trained models acquired during the large-scale pre-training phase poses a challenge in providing project-specific, consistent logging text and appropriate logging levels for a given logging statement. This could lead to inconsistencies in the logging style, potentially impacting the readability [Li et al. 2023a] and maintainability of the software.

(3) *Missing type information of logging variables.* Suggesting proper logging variables not only requires predicting the object variables (i.e., *service* in the first paragraph) themselves but also predicting their attributes and member functions (i.e., the *getServiceState()* member function). However, variable attributes and member function declaration are usually defined within the class, which stays out of the target method context. Existing work [Liu et al. 2019; Mastropaolo et al. 2022] only focus on the inside content of a method while never covering the detailed variable type information from the outside. This type information, if present, could provide explicit definitions for the attributes and member functions of the variables. Without such information, the logging models may mistakenly invoke non-existent member functions and misuse variables, which will further lead to compilation errors and software bugs.

Our Work. To tackle these limitations, we propose SCLogger, the first logging statement generation approach powered by inter-method **S**tatic **C**ontexts. SCLogger analyzes inter-method programming contexts for logging statement generation with four phases, including static scope extension, logging style adaption, contextualized prompt construction, and logging variable refinement. In the static scope extension phase, SCLogger extends the static scope of the given method by constructing the function invocation relationships, deriving the execution paths containing logging statements, as well as collecting available variables for the current method, such as class member variables and inherited variables, as the logging variable candidates. During this phase, the limitation of **limited static scope** is mitigated. In the logging style adaption phase, SCLogger adapts the idea of the in-context learning (ICL) strategy to select intra-project similar examples demonstrating the logging styles and logging patterns of the current project, which address the issue of **inconsistent logging style**. In the contextualized prompt construction phase, SCLogger translates the inference steps of logging statement generation into a chain-of-thought (COT) prompt. The context-aware code knowledge generated by the previous two steps, along with the COT prompt, is put together as a *contextualized prompt*. With the constructed contextualized prompt, SCLogger then invokes a large language model to generate the required logging statement for the given target method. In the final phase, i.e., logging variable refinement, SCLogger further refines the usage of logging variables. It provides comprehensive type definitions for logging variables that were generated in the third phase. This allows SCLogger to self-refine and correct the variable usage, ensuring syntactical correctness of the generated logging statement. This phase tackles the limitation of **missing type information**.

Following the previous works [Ding et al. 2022, 2023a], we conduct a comprehensive evaluation on ten open-source Java projects from different domains. The results show that SCLogger achieves the best performance over all metrics in both deciding logging location (i.e., *where-to-log*) and generating logging statement content (i.e., *what-to-log*). More specifically, SCLogger outperforms the state-of-the-art approach by 8.7% in logging location accuracy, 32.1% in logging level accuracy, 19.6% in logging variable precision, and 138.4% in logging text *BLEU-4* score, respectively.

Moreover, SCLogger consistently enhances the performance of logging statement generation models with various backbone large language models, thus demonstrating its generalizability. Besides, we explore the individual contribution of each phase and provide the reasoning for the improvements brought by our approach.

This paper's contributions are summarized as follows:

- To the best of our knowledge, we propose the first contextualized logging statement generation approach named SCLogger. With analyzed static context, SCLogger addresses the limitations of current method-level approaches with limited context.
- We propose a novel prompt structure to incorporate static context of code into large language models, which can be generalized to various language models for future improvement.
- We conduct the comprehensive evaluation of SCLogger on public logging datasets. The results demonstrate the effectiveness of SCLogger and the adaptability of SCLogger with different backbone models.
- The source code of SCLogger is publicly available at https://github.com/YichenLi00/SCLogger to benefit both developers and researchers.

2 MOTIVATING STUDY

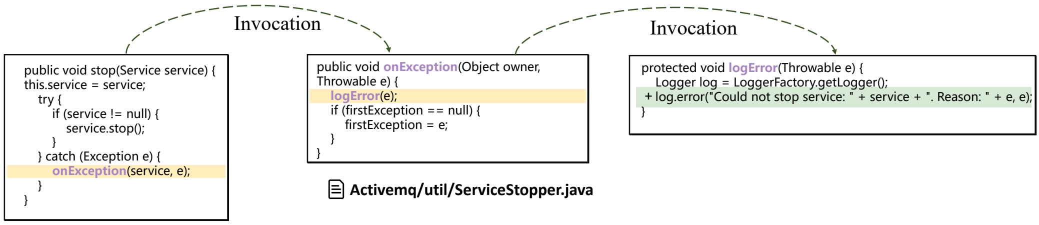
Logging statement automation has been a longstanding area in the domain of software development and maintenance [He et al. 2021], since high-quality logging statements can precisely describe system activities and ease the burden for maintainers to diagnose system anomaly behaviors. A wide range of approaches have been proposed to automatically recommend proper logging points [Li et al. 2020; Mastropaolo et al. 2022; Zhao et al. 2017] or generate effective logging statements [Ding et al. 2022, 2023a; He et al. 2018; Liu et al. 2022; Mastropaolo et al. 2022] for developers.

However, we discover that existing studies contain the same limitation of only exploiting single-method information for automatic logging, missing the inter-method contexts. In fact, since methods in software are interrelated, the programming context across different methods plays a vital role in understanding logging purposes and making logging suggestions accordingly. Intuitively, if the variable is defined outside a certain method, the developer can only log this variable properly once he/she reads the outside context. In this section, we perform a motivating study to illuminate the need for a more context-aware approach. In particular, we present real-world cases to illustrate limitations introduced by using a single-method context, that is, *limited static scope*, *inconsistent logging style*, and *missing type information of variables.*

2.1 Limited Static Scope

Modern software consists of a great number of methods, each of which is responsible for small functionalities. Single methods are often short, and their content is insufficiently informative for logging generation models to grasp the logging purpose. The static profile from other methods, including invocations, subsequent logs in execution order, and available variables, should be expanded within the current static scope. This additional context can be instrumental in understanding system behavior for logging purposes.

We exhibit Fig. 1 as an example from the ActiveMQ project to illustrate the significance of static scope. In this example, the `logError` method, which consists of two lines, is expected to log the



```
public void stop(Service service) {
this.service = service;
    try {
        if (service != null) {
            service.stop();
        }
    } catch (Exception e) {
        onException(service, e);
    }
}
```

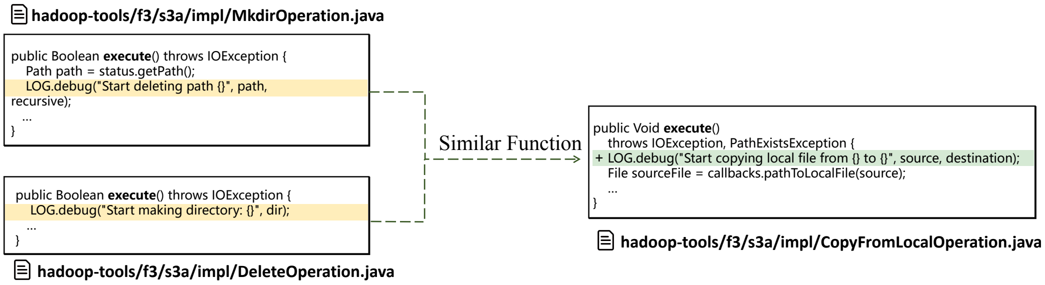
```
public void onException(Object owner,
Throwable e) {
    logError(e);
    if (firstException == null) {
        firstException = e;
    }
}
```

```
protected void logError(Throwable e) {
    Logger log = LoggerFactory.getLogger();
  + log.error("Could not stop service: " + service + ". Reason: " + e, e);
}
```

Fig. 1. Motivating example 1. The origin logging statement is highlighted in the green area while the invocation points are highlighted in the orange area.

error information for a certain error. Given only the method name and a single line of logger registration, it becomes relatively challenging to predict what kind of error should be logged [Yuan et al. 2012c]. However, by tracing the method's invocation sequence, we discover that `logError` is called within the method `OnException`, which in turn is invoked by the `stop` method of the `ServiceStopper` class. Consequently, such invocations reveal that the purpose of `logError` method is to record the state of service stop from `stop` method. Without the invocation information of the `logError` method, it would be nearly impossible to infer the appropriate logging statements for recording such service `stop` behavior and record this error. Moreover, the fact that *service* is also beyond the scope of the *logError* method further underscores the limitations of method-level static scope.

Furthermore, many invocation relationships span across different files, and some logs need to be interpreted within a sequence of logs [Chen et al. 2021; Du et al. 2017; Huo et al. 2023; Li et al. 2022]. Therefore, directly extending context to the file level may also be insufficient to address the issue of lack of information. To thoroughly understand the relative functionality of target method in program execution, it is essential to acquire a more comprehensive and specific context through inter-procedural analysis.


hadoop-tools/f3/s3a/impl/MkdirOperation.java
Similar Function
hadoop-tools/f3/s3a/impl/DeleteOperation.java
hadoop-tools/f3/s3a/impl/CopyFromLocalOperation.java

```
public Boolean execute() throws IOException {
    Path path = status.getPath();
    LOG.debug("Start deleting path {}", path,
recursive);
    ...
}
```

```
public Boolean execute() throws IOException {
    LOG.debug("Start making directory: {}", dir);
    ...
}
```

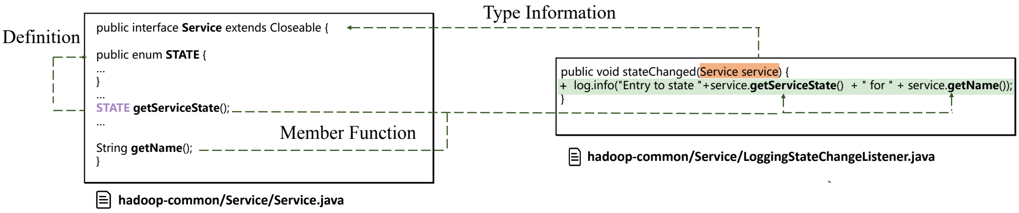
```
public Void execute()
    throws IOException, PathExistsException {
+ LOG.debug("Start copying local file from {} to {}", source, destination);
    File sourceFile = callbacks.pathToLocalFile(source);
    ...
}
```

Fig. 2. Motivating example 2. The origin logging statement is highlighted in the green area while the logging statements in the similar methods are highlighted in the orange area.

2.2 Inconsistent Logging Style

Logging style in software development and maintenance is maintained with relative consistency [Rong et al. 2018, 2020, 2017; Yuan et al. 2012a] in a mature project. Examples of this consistency include maintaining coherent logging levels for the lifecycle of certain components [Liu et al. 2022], writing logging statements with similar words [Li et al. 2023a], applying the same separators in logging text, and so on. To understand the logging style of the current project, developers naturally learn from similar logging examples within the project.

Fig. 2 showcases an instance from the Hadoop-AWS toolset within the Hadoop project. The three `execute` methods, extracted from `MkdirOperation`, `CopyFromLocalOperation`, and `DeleteOperation` files, are the primary execution methods that enable various file operations in the S3A filesystem. These methods serve as similar methods, exhibiting similar logging styles, particularly in terms of levels and wording. Without similar methods as a reference of logging style, it would be challenging to infer that such file operations should choose the `debug` level and use the `Start doing` text structure. Furthermore, given only the target function `execute`, the model has no way of knowing that similar file operation methods in the project will log `Start doing..` at the beginning of the methods before performing file operations. Hence, missing additional methods as references might cause logging style inconsistency, which can be mitigated by providing samilar methods from the same project.



```
public interface Service extends Closeable {

public enum STATE {
...
}
...
STATE getServiceState();
...
String getName();
}
```

hadoop-common/Service/Service.java

```
public void stateChanged(Service service) {
+ log.info("Entry to state "+service.getServiceState() + " for " + service.getName());
}
```

hadoop-common/Service/LoggingStateChangeListener.java

Fig. 3. Motivating example 3. The origin logging statement is highlighted in the green area while the corresponding logging variable is highlighted in theorange area.

2.3 Missing Type Information of Logging Variables

Since variables are often defined outside the method (*e.g.*, class attributes for object-oriented programming languages [Yuan et al. 2012c]), the third limitation of the intra-method context for logging is the missing type information of logging variables. Ignoring such information obstructs logging models from determining the proper usage of a variable (e.g., properties from a class) even though they understand the logging purpose.

Fig. 3 showcases an example from the Hadoop project, emphasizing the critical role of variable type information context within the automated logging process. The primary objective of this logging statement is to record the state change status of the given service. To this end, it is essential to invoke the two member functions (i.e., `getServiceState()` and `getName()`) from other classes defined outside the method, which retrieve the defined service state and name, respectively. Without integrating the `Service` interface information at `Service.java` into logging context, logging models are asked to guess the member functions of `service`, inevitably impairing their performance and practicality. The incorrect variable predictions (e.g., devoid invocation) can further lead to program compilation errors and software bugs.

Insights. The motivating study demonstrates the limitations of method-level context in understanding the semantics of the target method, maintaining consistency in the project-specific logging style, and selecting appropriate logging variables. Hence, we should devise models equipped with more context-aware code knowledge that does not exist in the target method for effective logging.

3 METHODOLOGY

3.1 Overview

First of all, we describe the problem of logging statement generation as follows. Given a method code as input (i.e., target method), the goal of this task is to predict the logging location and its corresponding logging statement content. Specifically, the purpose of SCLogger is to predict the code line number (i.e., location) and generate a complete logging statement accordingly.

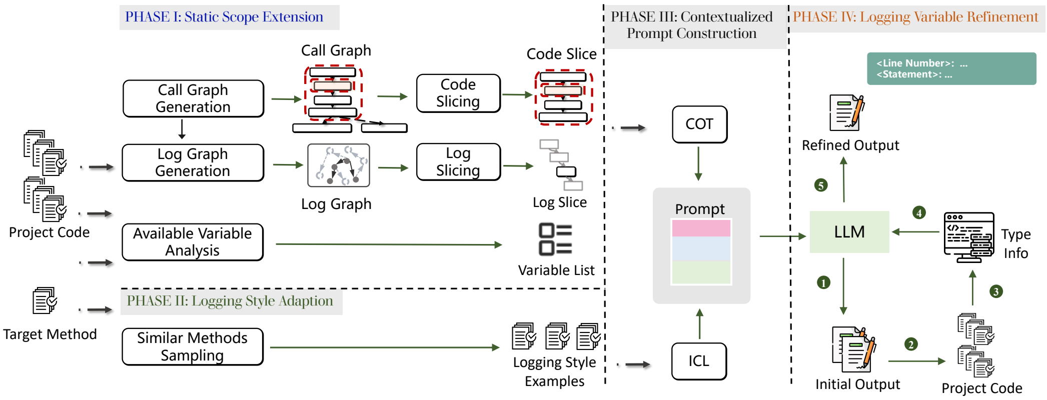


Fig. 4. The overview workflow of SCLogger.

We propose SCLogger, a static analysis enhanced contextualized logging statement generation framework via large language models. Intuitively, simply putting the entire project code into language models should work, however, such long input can be out of the model's input limit size and make the model get lost. As a result, we need a more advanced approach to extract useful information from the entire project. To this end, SCLogger extracts the logging-related context surrounding the target method and constructs contextualized prompts, which will be fed into language models for predicting logging position and generating logging statements. We present Fig. 4 to illustrate the workflow of SCLogger.

SCLogger takes the target method and its corresponding project code as input. The static scope extension phase derives inter-method information including code slice, log slice, and variable list. Code slice is a chain of methods code reflecting the method calling relationship associated with target methods. Log slice recording the potential subsequent and precedent logs during execution. The variable list contains all available variables for the target method. Afterwards, the logging style adaption phase utilizes the in-context learning (ICL) strategy by sampling a small set of similar methods from the project as logging style examples. Then, the third phase applies the chain-of-thought (COT) strategy [Wei et al. 2022] to translate logging inference into a few steps, then combines it with the context-aware knowledge coming from the two previous phases, to form a *contextualized prompt*. This combined prompt is then fed to the large language model (LLM) to get a tentative logging statement with the corresponding position (i.e., line number). During the final phase, logging variable refinement, SCLogger constructs a new *refinement prompt* that contains the detailed type information of the variable extracted from the tentative result. SCLogger eventually feeds the new prompt into the LLM and generates the final logging statement with rectified variables. The example with the contextualized prompt and the refinement prompt is illustrated in Fig. 6.

3.2 Static Scope Extension

The static scope extension phase aims to extract the static context associated with logging that surrounds the target method. Ultimately, this phase will generate three types of context: the code slice, the log slice, and the list of available variables.

3.2.1 Code Slice Generation. To enhance the model's understanding of the target method's relative position and functionality within the project, we designed the code-slicing step to extract the invocation context. The code slicing phase of SCLogger is designed to extract the relevant invocation methods of target method m_t from the statically generated call graph, which describes the invocation relationships among methods. To construct a relatively accurate call graph, our

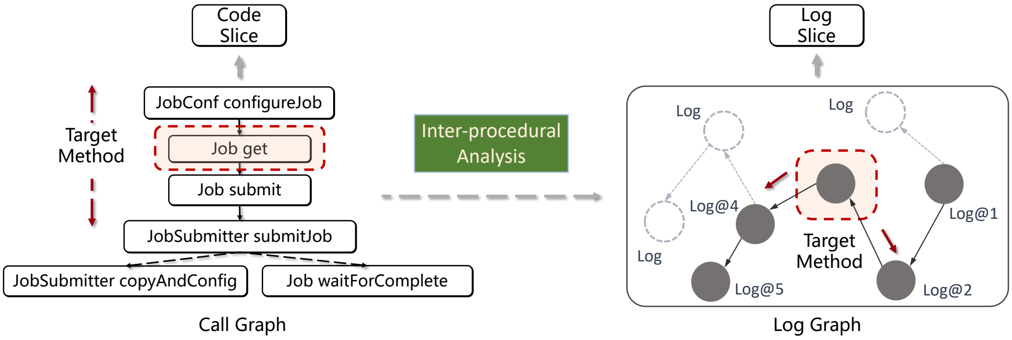


Fig. 5. The log slice and code slice example of SCLogger. The target method is highlighted in the red area.

model utilizes a context-sensitive pointer analysis [Li et al. 2018] to increase the precision of the call graph, especially in handling virtual method calls and similar situations.

In particular, the code slicing process identifies the methods that either invoke the target method or are invoked by the target method within two hops. Here, a single invocation can be considered as one hop. The position in the call graph corresponds to the node linked with the target method m_t, as per the invocation details. The graph traversal process follows the graph's directed edges, either forward or backward, to extract the preceding and succeeding methods. If there are too many paths within two hops, we only randomly select one of these invocation paths. As shown in Fig. 5, for method *get*, its method invocations of within two-hop are *configureJob*, *submit* and *submitJob* The output of code slicing is text descriptions of invocation relationships between the identified methods, as well as the chain of these method codes (i.e., *get*, *configureJob*, *submit*, and *submitJob*).

3.2.2 *Log Slicing.*

A single log event, being part of the log sequence, cannot be fully understood in isolation. The log slicing phase aims at identifying preceding logs and subsequent logs of a given target method based on a log graph which indicates the log dependency relationships.

(1) *Log Graph Construction*: To extract both preceding and succeeding logging statements for the target method, SCLogger constructs log graphs for a given project p. This construction process is guided by the frameworks and ideas presented in previous works [Chen et al. 2018; Huo et al. 2023; Zhao et al. 2023]. A log graph is characterized as a directed graph (L, E) (as shown in Fig. 5), where L represents the set of logging statements which is the node set in graph and E represents the edge set, which is composed of program execution paths. Each logging statement $l \in L$ is obtained through static analysis of the source code of project p with its belonging method m. Consequently, the connected logging statements in the log graph are causally related with the possible execution order, as they are derived from the execution paths within the same source code of project p.

Specifically, SCLogger extends the framework of previous works [Huo et al. 2023; Zhao et al. 2023]. It begins by analyzing and identifying the set of methods $M' \subseteq M$ that contain any logging statement. SCLogger then uses the call graph for the project p described above. For the call graph of project p, SCLogger prunes the methods that do not directly or indirectly invoke any method $m \in M'$ from the complement of $\overline{M'}$ in M. These are referred to as log-irrelevant methods [Huo et al. 2023]. For the remaining methods, SCLogger conducts the static execution path analysis [Huo et al. 2023] for each method. All relationships, including control flow and method calls, which are also part of the Interprocedural Control Flow Graph (ICFG), are integrated into the edge set E. Thus, each edge $e \in E$ represents a potential execution path from one logging statement to another, which aids in understanding and handling the dependency relationships among logging statements.

While the log graph provides a comprehensive set of potential executable paths, it includes certain paths that remain infeasible regardless of the constrains. To refine the log graph and reduce these infeasible paths, we undertake a preliminary intra-procedural constraint analysis. This process involves the collation of constraints in a method and the initial filtering out of any paths that contain unsatisfiable constraints. If a path within a method is determined to be infeasible, all paths reliant on this infeasible path are subsequently removed. Note that due to the inherent limitations, some potentially infeasible paths may still remain.

(2) *Log Slice Generation*: With the log graph, SCLogger generates a log slice that consists of preceding and succeeding logging statements for the target method m_t. This is accomplished by traversing the graph both backward and forward, beginning from the position of the target method in the log graph. As shown in Fig. 5, log-specific slicing allows SCLogger to capture long-distance log dependency information within two log hops (which will be beyond two method hops, as some methods may not contain logs) for the target method, within a greatly reduced and focused context. This is more targeted than directly incorporating all relevant code into the context according to the invocation sequence. The position in the log graph refers to the node associated with the target method m_t according to invoke information and execution information. The process of traversing the graph involves following the directed edges in the graph, either forward or backward, to identify the preceding and succeeding logging statements. If an excessive number of nodes exist within two hops, we simply choose one of these paths from the methods available in the **training set**.

3.2.3 *Scope Variable Analysis.*

The scope of accessible logging variables for a given method is not limited to its parameters and local variables. It also encompasses variables that are beyond the class level, including those inherited from a parent class. Consequently, merely capturing all member variables in the current file does not yield a comprehensive overview of available variables [Yuan et al. 2012c]. We illustrate the details of this phase as follows:

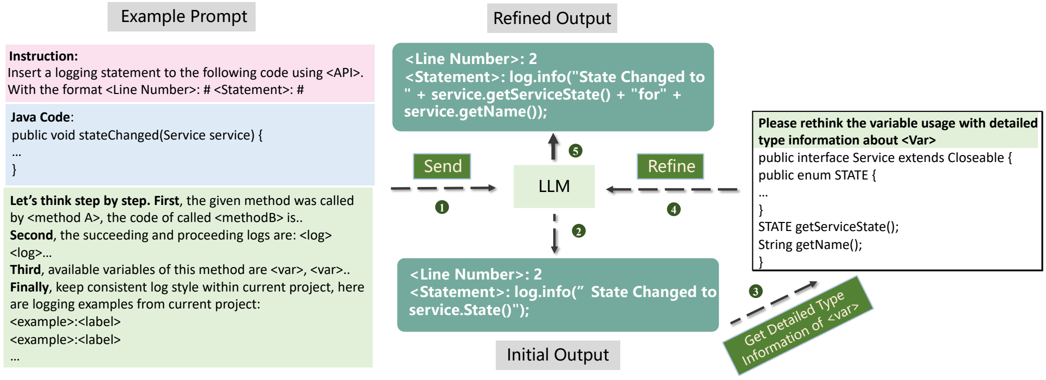
In the context of a target method m_t, we define several sets of variables. V_p represents the set of parameters, which are the inputs to the method. V_m stands for the set of local variables, which are defined and used only within the method. V_c is the set of class member variables, which belong to the class that the method is a part of. V_s is the set of static variables, which belong to the class as a whole rather than any specific instance. Lastly, V_i denotes the set of inherited variables, which are class member variables that come from the parent class.

The logging process should focus on any variable v that belongs to one or more of these sets ($v \in V_p \cup V_m \cup V_c \cup V_s \cup V_i$) during the execution of m_t [Yuan et al. 2012c]. As a result, all these variables need to be included in the context for selection as potential logging variables and subsequently form the available variable list.

Note that here, we are not giving the model the detailed type definition for each variable. The list of available variables primarily includes the variable name and roughly inferred type name. We will further discuss how we address the further type issue of logging variables in Sec.3.5.

3.3 Logging Practice Adaptation

To align with the logging style of the current project, SCLogger employs the in-context learning (ICL) strategy, which has proved its effectiveness in code-related tasks [Gao et al. 2023; Peng et al. 2023], to adapt to the project's logging style. The in-context learning strategy [Dong et al. 2022], as its name suggests, allows the model to learn and adapt to the specific examples of the project, ensuring that the generated logging statements are in line with the project's existing style. Specifically, this strategy provides a few examples sampled from the inter-project training set (with labels, detailed in Sec. 4.1) as demonstrations of logging style so that SCLogger can learn from these examples to generate consistent logging statements.



Instruction:
Insert a logging statement to the following code using <API>.
With the format <Line Number>: # <Statement>: #

Java Code:

```
public void stateChanged(Service service) {
...
}
```

Let's think step by step. First, the given method was called by <method A>, the code of called <methodB> is..
Second, the succeeding and proceeding logs are: <log> <log>...
Third, available variables of this method are <var>, <var>..
Finally, keep consistent log style within current project, here are logging examples from current project:
<example>:<label>
<example>:<label>
...

<Line Number>: 2
<Statement>: log.info("State Changed to " + service.getServiceState() + "for" + service.getName());

Please rethink the variable usage with detailed type information about <Var>

```
public interface Service extends Closeable {
public enum STATE {
...
}
STATE getServiceState();
String getName();
}
```

<Line Number>: 2
<Statement>: log.info(" State Changed to service.State()");

Fig. 6. Contextualized prompt example with logging variable refinement.

Following previous works [Gao et al. 2023; Peng et al. 2023], we use the BM25 [Robertson et al. 2009] similarity function to select these examples. The BM25 function is based on the TF-IDF (Term Frequency-Inverse Document Frequency) method. With the code of a target method as input, the BM25 function calculates the term frequency of each keyword in the query within the examples. It then multiplies this frequency by the inverse document frequency of the given term. The BM25 similarity score will be higher if there is a greater relevance between the query and the examples. This score helps SCLogger to select the most relevant examples from the training set. Specifically, we select the top five examples from the training set with the highest BM25 similarity scores. The example prompts are combined with the other prompt to form the complete contextualized prompt, as shown in Fig. 6.

3.4 Contextualized Prompt Construction

As shown in Fig 6, SCLogger converts all the context information gathered from static analysis during the first two phases into chain-of-thought (COT) [Wei et al. 2022] prompts, which incorporate static domain knowledge regarding the requirements of a logging statement. The innovative COT approach, which employs sequential reasoning, guides language models towards generating complex and specific outputs [Peng et al. 2023]. This approach allows the model to focus on one aspect of the task at the time, potentially enhancing the quality of the generated outputs.

Additionally, SCLogger integrates the sampled logging style examples from the second phase with the In-Context Learning (ICL) strategy into the input prompt with reasoning prompt. It suggests the model to maintain a style consistent with the logging samples derived from the current project. By fusing these two components, the initial contextualized prompt is ultimately constructed.

3.5 Logging Variable Refinement

To address the issue of variable usage within selected variables, an intuitive idea might be to provide the detailed type information for every variable alongside the available variable list during the first phase. However, offering the detailed type definition of every available variable within the current scope is unrealistic and would lead the model into a wild-goose chase.

To tackle this, we employed a two-staged variable type refinement mechanism to determine the proper usage of logging variables, as shown in Fig. 6. In the first stage, after providing the model with the contextualized prompt with an available variable list, we let the model conduct the inference. Then, we extract the logging variable chosen by the model and conduct a thorough type analysis of that variable to generate the detailed type information extracted from the project, which is then fed back to the model for reconsideration and self-refinement.

To obtain the detailed type information of a selected logging variable, we performed static analysis with class definition resolution. Given a variable v and target method code m_t within a project p, we first find the type t of v using with variable type resolving, then acquire its detailed information of t. Resolving the variable's type within method m_t involves checking the parameter scope of m_t, local scope within m_t, and the class c_t where m_t is defined if the variable is a class-level variable.

After obtaining type t, we denote the set of acquired referring class definitions in the project p as Def by analyzing imported intra-project packages and the current package. A class definition $def \in Def$ is a tuple (t, M, A), where t is the type, M is the set of member functions, and A is the set of attributes. The class definition resolving function $R : t \times p \to (M, A)$ is then defined as follows:

$$R(t, p) = (M, A) \quad \text{if} \quad \exists (t', M, A) \in Def \quad \text{such that} \quad t = t' \tag{1}$$

The function R takes a type t and a project p as input and returns the member functions M and attributes A of the type if there exists a definition (t', M, A) in Def where $t = t'$.

In the case of inheritance and polymorphism, we traverse the class hierarchy to collect all relevant member functions and attributes. If the class implements any interfaces, we also consider the methods declared in these interfaces. For generic classes, we consider all possible concrete types the generic type can take. This process is repeated recursively until we have a complete picture of the class's definition, including its inherited and overridden member functions and attributes.

In conclusion, we extract detailed type information of a variable, including its member functions and attributes, through variable type resolution and class definition resolution, for refining the generated logging statement.

The resolved detailed type information will be fed into SCLogger, if the model realizes it has used the variable type incorrectly (e.g., not using *var.getinfo()*), it will take this opportunity to carefully read the logging variable's information and correct the type error and further method usage. This allows SCLogger to self-refine and correct the variable usage, ensuring syntactical correctness of the generated logging statement.

4 EXPERIMENT SETUP

4.1 Subject Projects

Following previous works [Ding et al. 2022, 2023a; He et al. 2018], we evaluate SCLogger on ten open-source Java projects that span various domains, such as storage, cloud platforms, computation engines. Detailed information of these projects can be found in Table. 1. The source lines of code (SLOC) for the investigated projects range from 330K to 2.12M. Each project contains between 1,978 and 15,744 logging statements within 901 to 7,365 methods that contain these statements. Notably, every project boasts a development history exceeding ten years, which highlights the progression and evolution of each software system. Note that our decision not to select another generic logging dataset [Mastropaolo et al. 2022] is based on the fact that it is comprised solely of sampled methods without any supplementary information such as the associated path or project. This absence of information proves insufficient for conducting static analysis for obtaining context.

Next, we analyze the invocation of popular logging APIs (i.e., Log4j [Apache 2023] and Slf4j [Gulcu 2023] at the Abstract Syntactic Tree (AST) level to extract all log statements from the original samples to complete the datasets: The extracted logging statements were marked with a line number tag (i.e., <Line Number#>) with corresponding logging statement (i.e., <Statement>: *log.info(msg)*) to indicate their position in the initial method and the complete logging statement. These served as the ground truth labels for their respective methods.

In line with works [Ding et al. 2022, 2023a; He et al. 2018], for each subject project, we randomly split all the methods containing logging statements into the ratio of train:test=8:2. All sampling and static analysis processes of SCLogger will not involve any methods in the test set.

Table 1. Details of the studied projects.

Project	Version	SLOC	# of logging statements	# of methods contain logging statements
ActiveMQ	5.16.0	415k	5,352	2,876
Ambari	2.7.5	490k	4150	1,689
Brooklyn	1.0.0	339K	2,937	1,374
Camel	4.0.0	2.12M	9,603	4,460
CloudStack	4.16.11	782k	11,261	3,994
Hadoop	3.3.0	1.7M	15,744	7,365
HBase	2.4.0	912K	8,677	3,526
Hive	3.1.2	1.7M	7,415	2,650
Ignite	2.8.1	1.1M	4,319	2,335
Synapse	3.0.1	330k	1,978	901

4.2 Baselines

We choose LANCE [Mastropaolo et al. 2022], the first and only one-stop logging statement approach based on T5 [Raffel et al. 2020] and and its updated version, LANCE2.0[Mastropaolo 2023], as our primary baselines, since other appraoches only focus on certain subtask (*i.e.*, logging level prediction [Li et al. 2021; Liu et al. 2022]). Code completion models are also beyond the scope of our baselines, due to their inability to infer the logging position. Given the progress in the development of Large Language Models (LLMs) and their potential use in similar development tasks, we also consider several prominent LLMs as baselines, including GPT-3.5 [OpenAI. 2022], Davinci [OpenAI 2023], GPT-4 [OpenAI 2023] and Llama-2-70b [Touvron et al. 2023]. We further choose these models as the backbone models to demonstrate the generalizability of our approach. For the LLM-based baselines, we provide five fixed examples for task demonstration. Implementation details can be seen in Section. 4.4.

We intentionally did not compare SCLogger with approaches that focus on a specific aspect of *what to log* (e.g., logging text generation) or current code completion models. This is because they are unable to locate the logging position in a one-stop manner and generate the corresponding logging statement. Our experimental results have demonstrated that our approach can be generalized to various backbone models, emphasizing its effectiveness as a generalized strategy instead of a specific trained model.

4.3 Metrics

Following the previous work[Mastropaolo et al. 2022], we evaluate the effectiveness of SCLogger with respect to two primary dimensions: *where to log* and *what to log*.

4.3.1 Where to log. In line with previous work [Mastropaolo et al. 2022], we employ the metric of Position Accuracy (PA) to assess the performance of logging position prediction. We argue that the block level might be overly coarse. In this scenario, we calculate PA as 1 (indicating a successful prediction) if the distance between the predicted line number and the actual line number is less than or equal to one line and both predicted and actual line numbers must be within the same block. Otherwise, PA is calculated as 0 (indicating an unsuccessful prediction).

4.3.2 What to log. Under the *what to log* category, we evaluate SCLogger in terms of its *logging levels*, *logging variables*, and *logging texts* following the previous work [Li et al. 2023b].

(1) Logging levels. We adopt level accuracy *(L-ACC)* and Average Ordinal Distance Score *(AOD)* from previous studies [Li et al. 2023b, 2021; Liu et al. 2022] to evaluate logging level predictions. L-ACC represents the percentage of correctly predicted log levels, while AOD calculates the distance between logging levels. since different levels are not independent of each other. For example, the `error` is closer to `warn` compared with `trace`. The formula for AOD is $AOD = \frac{\sum_{i=1}^{N}(1-Dis(a_i,s_i)/MaxDis(a_i))}{N}$, 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. We employ *Precision*, *Recall*, and *F1* to evaluate the predicted set of logging variables. For each generated logging statement, we denote the variables in the model's prediction as S_p and the variables in the actual logging statement of ground truth as S_g. We calculate the precision ($\frac{S_p \cap S_{gt}}{S_p}$), recall ($\frac{S_p \cap S_g}{S_g}$), and their harmonic mean (F1=2 $* \frac{Precision*Recall}{Precision+Recall}$), and report these metrics. **Note that** predictions that use the same variable as the ground truth with different member function usages are considered incorrect predictions.

(3) Logging texts. We use *BLEU* [Papineni et al. 2002] and *ROUGE* [Lin 2004] metrics, consistent with previous research [Ding et al. 2022; Mastropaolo et al. 2022], to evaluate the quality of the generated logging texts. These metrics compute the similarity between generated and actual log messages, ranging from *0* to *1*. A higher score indicates better quality. Specifically, we use BLEU-K ($K = \{1, 4\}$) and ROUGE-K ($K = \{1, L\}$) to compare the overlap of K-grams.

Note that for practicality, we **only calculate** the metrics for *what-to-log* when the logging position(*where-to-log*), is **predicted correctly**. This is because if the logging position is not right, the purpose and meaning of the log would be incorrect, thus lacking value for further evaluation.

4.4 Implementation

The static analysis part of SCLogger has been implemented using 4,738 lines of Java code, leveraging both Soot [Vallée-Rai et al. 2010] and Eclipse JDT Core [Foundation 2023] for comprehensive Java bytecode and source code analysis. The experiments of SCLogger and all baselines were conducted on a Linux machine (Ubuntu LTS 18.04) equipped with an Intel Xeon Platinum 8255C Processor (2.50GHz), four NVIDIA A100-80GB GPUs, and 1TB of RAM.

For GPT-3.5, Davinci and GPT4o, we use the public APIs provided by OpenAI[OpenAI 2023] with *gpt-3.5-turbo-0301*, *text-davinci-003* and *gpt-4-0314*, respectively. We run the Llama2-70b model on our machine using the Llama version *Llama2-70b-chat-hf* to infer the results. By default, we set the hops of log slice and code slice to 2 and give 5 in-project examples in logging style adaption phase. For baselines, we use 5 fixed examples for task demonstration.

5 EVALUATION RESULTS

5.1 Research Questions

For the evaluation, we consider the following research questions:

- **RQ1**: How effective is SCLogger compared with existing approaches?
- **RQ2**: What is the impact of different phases of SCLogger?
- **RQ3**: How generalizable is SCLogger for different backbone models?
- **RQ4**: What is the impact of different logging examples?

5.2 RQ1: How Effective is SCLogger Compared with Existing Approaches?

To evaluate the effectiveness of SCLogger in logging statement generation task, we conduct a comprehensive evaluation with comparison to other baselines on the datasets. The evaluation results are illustrated in Table. 2, where the best results for each metric are marked in **bold face**. We analyze the evaluation results from two dimensions: *where-to-log* and *what-to-log*.

Table 2. Logging statements generation results from both *where-to-log* and *what-to-log* dimensions.

Model	Posistion	Logging Levels		Logging Variables			Logging Texts			
	PA	L-ACC	AOD	Precision	Recall	F1	BLEU-1	BLEU-4	ROUGE-1	ROUGE-L
LANCE	0.501	0.574	0.763	0.657	0.414	0.508	0.207	0.110	0.179	0.175
LANCE2.0	0.563	0.601	0.807	0.632	0.508	0.563	0.219	0.113	0.275	0.266
Davinci-003	0.307	0.470	0.714	0.626	0.544	0.582	0.267	0.128	0.288	0.295
Llama-2-70b	0.248	0.442	0.682	0.506	0.477	0.490	0.209	0.070	0.218	0.219
GPT-3.5	0.395	0.495	0.727	0.618	0.496	0.550	0.164	0.064	0.176	0.174
GPT-4	0.518	0.562	0.779	0.634	0.611	0.622	0.285	0.138	0.317	0.321
SCLogger	**0.612**	**0.794**	**0.914**	**0.758**	**0.735**	**0.746**	**0.493**	**0.329**	**0.517**	**0.509**

Table 3. Ablation Study of SCLogger.

Ablation	Posistion	Logging Levels		Logging Variables			Logging Texts			
	PA	L-ACC	AOD	Precision	Recall	F1	BLEU-1	BLEU-4	ROUGE-1	ROUGE-L
SCLogger	0.612	0.794	0.914	0.758	0.735	0.746	0.493	0.329	0.517	0.509
w/o Loging Scope Extension	0.579	0.702	0.858	0.720	0.711	0.716	0.430	0.278	0.468	0.469
w/o Logging Style Adaption	0.549	0.679	0.869	0.752	0.696	0.723	0.354	0.191	0.393	0.386
w/o Logging Variable Refinement	0.614	0.791	0.912	0.708	0.654	0.680	0.483	0.348	0.507	0.503

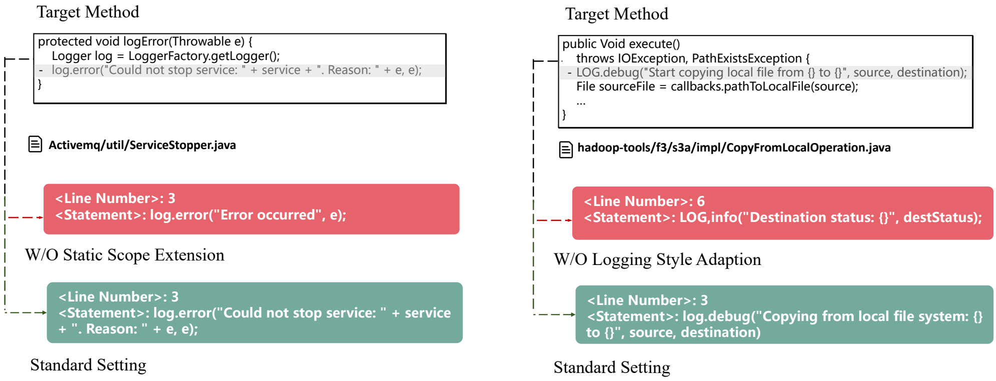
Where-to-log. According to the evaluation results, it is clear that SCLogger outperforms all baselines in logging position prediction. Specifically, SCLogger outperforms the best performing baseline, LANCE2.0, by 8.7%. Furthermore, despite having five fixed examples as task demonstrations for Large Language Models (LLMs), only GPT-4 manages to exceed the performance of the domain-specific baseline, LANCE, which is based on T5 with the considerably smaller size. The performance of the remaining LLMs on this task underscores the current limitations of these models. Notably, under the line-level metric, PA, the performance of SCLogger further underscores the effectiveness in the task of determining *where to log*.

What-to-log. We compare SCLogger with all baselines in terms of *logging levels*, *logging variables*, and *logging texts*. Regarding the logging levels, we observe that SCLogger outperforms the best performing approach by 32.1% and 13.3% for level accuracy and AOD. When considering logging variable prediction, SCLogger achieves a consistent improvement of 19.6% to 20.3% on *prediction* and *recall* than the best performing baseline. These results demonstrate that that SCLogger, equipped with domain knowledge of available variables and detailed type information, significantly outperforms existing methods in predicting logging variables, along with their attributes and member functions. In terms of logging text generation, SCLogger shows a marked improvement compared to all baselines. It achieves a BLEU-4 score of 0.386 and a ROUGE-L score of 0.599, outperforming GPT-4 by 138.4% and 58.6%, respectively. By adopting logging style adaptation and contextualized strategies, SCLogger can generate logging text that aligns with the current project's logging style, both in text structure and wording. This not only improves the overall quality of the logging text but also highlights the its potential practicality for real-world software development.

Answer to RQ1. By introducing the context information in the prompt design, SCLogger demonstrates superior performance in both dimensions of *where to log* and *what to log*, significantly outperforms the best baseline.

5.3 RQ2: What is the Impact of Different Phases of SCLogger?

We conduct an ablation study to investigate the impact of different components within the framework of SCLogger. Specifically, we design three variants of SCLogger by removing the proposed


Target Method
Activemq/util/ServiceStopper.java
W/O Static Scope Extension
Standard Setting
Target Method
hadoop-tools/f3/s3a/impl/CopyFromLocalOperation.java
W/O Logging Style Adaption
Standard Setting

```
protected void logError(Throwable e) {
    Logger log = LoggerFactory.getLogger();
-   log.error("Could not stop service: " + service + ". Reason: " + e, e);
}
```

```
<Line Number>: 3
<Statement>: log.error("Error occurred", e);
```

```
<Line Number>: 3
<Statement>: log.error("Could not stop service: " + service
+ ". Reason: " + e, e);
```

```
public Void execute()
    throws IOException, PathExistsException {
 -  LOG.debug("Start copying local file from {} to {}", source, destination);
    File sourceFile = callbacks.pathToLocalFile(source);
    ...
}
```

```
<Line Number>: 6
<Statement>: LOG,info("Destination status: {}", destStatus);
```

```
<Line Number>: 3
<Statement>: log.debug("Copying from local file system: {}
to {}", source, destination)
```

(a) Removing the phase of static scope extension.

(b) Removing the phase of logging style adaption.

Fig. 7. Case study of the ablation study about phase static scope extension and logging static adaption.

phases *i.e.*, logging scope extension, logging style adaption and logging variable refinement in comparison with SCLogger.

Table 3 demonstrates the experimental results. The results indicate that without conducting the static scope extension, the overall performance of SCLogger generally declines across all metrics. Specifically, there is a decrease of 5.4% in position accuracy (*PA*), while level accuracy (*L-ACC*) and variable precision experience drops of 11.6% and 5.0% respectively. This decline is primarily due to the misunderstandings of the target method semantics and corresponding logging purpose, given the limited information available at the method level code. When logging style adaptation is not conducted, the performance of SCLogger on logging text aspect experiences the most significant decrease, indicating the importance of providing logging style examples for formulating text structure and wording. Specifically, for logging text, the performance achieves the *BLEU* score from 0.354 to 0.191 and the *ROUGE* score from 0.393 to 0.386 for the studied projects, which are 28.2% to 41.9% and 24.0% to 24.2% lower than the standard setting, respectively. Additionally, the *PA* decreases by 10.3%, indicating that the demonstration of logging style examples can significantly enhance the performance of identifying logging positions. This also suggests that models can effectively learn logging patterns from these demonstrated examples. Furthermore, when the phase of logging variable refinement is omitted, the performance of SCLogger for logging variables drops obviously (8.8%, reflected by *F1*), while the performance of other dimensions almost remains relatively stable (i.e., logging level). This decrease demonstrates the effectiveness of the variable refinement phase with detailed type information.

Fig. 7 presents two cases (details described in Sec. 2.1 and Sec. 2.2) to illustrate how SCLogger can be benefited from each phase. The gray line represents the original logging statements. For instance, in Fig. 7a, without the extended method static scope information, SCLogger failed to understand the functionality of this method. As a result, SCLogger conservatively inferred a general error log without realizing that the current method's purpose is to report the error when the service cannot be stopped. With the help of extended static scope context (detailed in Sec. 2.1), SCLogger understand the functionality and available variables of current method, therefore can generate a more appropriate log recording the certain error. For the example shown in Fig. 7b, SCLogger cannot locate the logging location for such a method without understanding the logging pattern of current project. With the knowledge gaining from similar methods (detailed in Sec. 2.2), SCLogger can pinpoint the appreciate logging position (before the file operation) and generate a similar structure logging statement. The case presented in Fig. 6 demonstrates the effectiveness of

Table 4. The performance of SCLogger with different backbone models.

Model	Approach	Posistion	Logging Levels		Logging Variables			Logging Texts			
		PA	L-ACC	AOD	Precision	Recall	F1	BLEU-1	BLEU-4	ROUGE-1	ROUGE-L
LLaMa-2-70b	Base	0.248	0.442	0.682	0.506	0.477	0.490	0.209	0.070	0.218	0.219
	SCLogger	0.282	0.486	0.743	0.618	0.467	0.532	0.283	0.177	0.299	0.292
	Δ	↑ 13.7%	↑ 10.0%	↑ 8.8%	↑ 22.1%	↓ 2.1%	↑ 8.6%	↑ 35.4%	↑ 152.9%	↑ 37.2%	↑ 33.3%
GPT-3.5	Base	0.395	0.452	0.713	0.618	0.496	0.550	0.164	0.091	0.176	0.174
	SCLogger	0.478	0.559	0.766	0.712	0.548	0.619	0.324	0.213	0.330	0.329
	Δ	↑ 21.0%	↑ 23.7%	↑ 7.4%	↑ 15.2%	↑ 10.5%	↑ 12.5%	↑ 97.6%	↑ 134.1%	↑ 87.5%	↑ 89.1%
GPT-4	Base	0.518	0.562	0.779	0.634	0.611	0.622	0.285	0.138	0.317	0.321
	SCLogger	0.612	0.794	0.914	0.758	0.735	0.746	0.493	0.329	0.517	0.509
	Δ	↑ 18.1%	↑ 41.3%	↑ 17.3%	↑ 19.6%	↑ 20.3%	↑ 20.3%	↑ 73.0%	↑ 138.4%	↑ 63.1%	↑ 58.6%

logging variable refinement. After taking the type information of the variable *service*, SCLogger can retrieve the state information and service name by calling the relevant member functions *getServiceState()* and *gerName()*. As a result, a more suitable logging statement is generated, and the incorrect invocation of *getState()* is corrected.

Answer to RQ2. While evaluating individual contributions of each phase of SCLogger, the ablation study reveals that removing any component significantly decreases the overall performance in terms of all the metrics. Thus, each phase individually contributes significantly to the overall effectiveness of the SCLogger framework.

5.4 RQ3: How Generalizable is SCLogger for Different Backbone Models?

In this RQ, we evaluate the performance of SCLogger by utilizing various LLMs in conjunction with our contextualized strategy. We have selected three representative and popular LLMs that are frequently used in research, specifically GPT-4, GPT-3.5, and Llama-2-70b-chat. It should be noted that by default, SCLogger employs GPT-4 as the backbone model. Additionally, the size of the LLMs must be sufficiently large to ensure the capability of both ICL and COT [Gao et al. 2023; Xu et al. 2023].

The experimental results are shown in Table. 4. We observe that our contextualized strategy can consistently enhance the performance of the utilized base models in terms of all metrics by a large margin. On average, all models have improved by 17.6% in determining the logging position. From the perspective of *what to log*, models have improved their performance in selecting the logging level, predicting the logging variable, and generating the logging text by an average of 25% (reflected by *L-ACC*), 13.8% (reflected by *F1*), and 60.3% (reflected by *ROUGE-L*) respectively. Meanwhile, with stronger abilities to understand our designed prompt, larger language models benefit more from the contextualized strategy associated with SCLogger.

The results not only demonstrate the advantage of SCLogger's design but also demonstrate the generalizability for different backbone models of our contextualized strategy. We believe that the performance of SCLogger can be further improved with the development of code-specific large language models.

Answer to RQ3. SCLogger demonstrates the ability to consistently improve models' performance of logging statement generation, even when utilizing relatively smaller and not code-specific language models. This demonstrates the generalizability of the proposed contextualized strategy.

5.5 RQ4: What is the Impact of Different Logging Examples?

In this RQ, we evaluate the effects of the number of logging examples and example sampling similarity calculation approaches in the prompt design of SCLogger. Following previous works [Gao

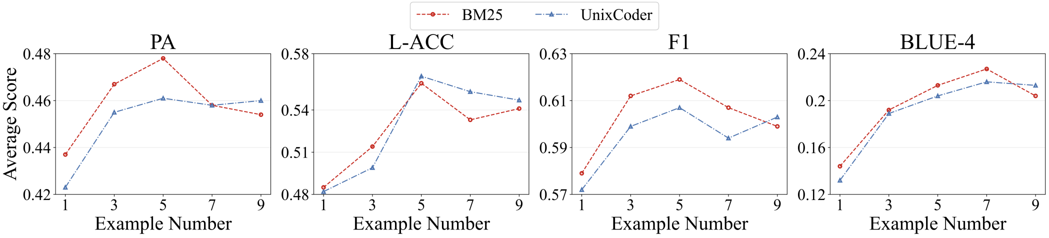


Fig. 8. The selected metrics of SCLogger with different numbers of examples and different sampling methods

et al. 2023; Peng et al. 2023; Xu et al. 2023], we change the number of examples from one to nine and compare two similarity calculation approaches: Unixcoder [Xu et al. 2023] (searching on the embedded space with Unixcoder [Guo et al. 2022], a unified cross-modal pre-trained code large language models) and BM25. Due to the expense and limited experiment resource, the experiment is conducted using the GPT-3.5 as the backbone model.

As shown in Fig. 8, we observe that the performance of SCLogger is affected by the number of logging examples, while is less affected by the sampling approaches. The performance drops significantly when the number of examples is relative small. Moreover, the performance of SCLogger using both sampling approaches either plateaus or starts to decrease after the example number is large than five. This proves that dissimilar logging examples and overly long prompts will result in performance loss, which stays consistent with previous works [Gao et al. 2023; Peng et al. 2023] on ICL with code tasks. While comparing BM25 to UnixCoder, we observe that BM25-similarity slightly outperforms UnixCoder in improving the performance of SCLogger. One possible explanation is that BM25, as a text retrieval algorithm, is more capable in capturing the textual similarity instead of code semantic similarity between the logging examples and target method, thus providing more relevant examples with same syntactic structure instead of semantics.

Answer to RQ4. SCLogger achieves the best performance with five examples, which contributes to maintaining a relatively short prompt length. The performance difference between the two sampling approaches is not significant, facilitating the use of SCLogger in various situations.

6 DISCUSSION

6.1 Practicality of SCLogger

SCLogger is designed to help developers write logging statements during software development and maintenance. We discuss the practicality of SCLogger from the following two aspects.

Cost reduction. For large language models, the cost is proportional to the length of the prompt. To reduce the cost, SCLogger only extracts and isolates the context related to logging to form the prompt, rather than taking the file-level content as input like existing programming assistants (i.e., Copilot [GitHub 2023]). Moreover, for the refinement phase, SCLogger only takes type information of chosen logging variables instead of all available variables, which also help with shorten prompt length. Our experiments show that in 84.3% of cases, our prompt is shorter than the length of the current file of the target method, which demonstrates the relative low cost.

IDE integration. SCLogger can be easily integrated into well-established Integrated Development Environments (IDEs), such as Eclipse[Foundation 2023], for practical applications. In particular, Eclipse JDT[Foundation 2023], the built-in static analysis tool of Eclipse, has the capability to automate the majority of the static analysis procedures of SCLogger. Compared with exisiting LLM-based code completion tools, such as Copilot [GitHub 2023] or Tabnine [Tabnine 2023], SCLogger offers more comprehensive static features beyond method-level to improve the model's logging performance. Furthermore, experimental results (as detailed in Sec. 5.4) demonstrate that SCLogger is compatible with a variety of large language models, thereby continuously benefiting from development of LLMs.

6.2 Threats to Validity

Potential data leakage. A primary concern in this work is the potential data leakage issue arising from the use of public code. Specifically, there is a possibility that the model has been trained on the test set, resulting in memorization of the results rather than conducting inference [Huang et al. 2023; Li et al. 2023b; Rabin et al. 2023; Yang et al. 2023]. To address this concern, instead of directly providing the model with the file-level contexts (which might exist in the training corpus), SCLogger receives a complex prompt composed of code snippets with logical reasoning relationships. This type of **data format** is unlikely to have been encountered by the model during training. Furthermore, our experimental results reveal that the model's performance in directly generating logging text is significantly below that of practical use-cases, indicating a minimal probability of direct memorization of the test set.
The selection of models. In this study, we employ three popular instruction-taken and practically coding-capable LLMs for experimentation, aiming to demonstrate the effectiveness of our proposed methodology. While a multitude of LLMs exist that could potentially be employed for experimentation, we have discovered that smaller parameter models fail to satisfy our requirements for understanding such complex prompt. Some models that we have experimented with either lack the capability to understand instructions or have not yet attained the level of practical application for instruction-taken coding. In future work, we plan to extend our experimentation to other emerging models, thereby evaluating the further generalizability of our method.
The selection of language. One potential external concern may be that the datasets primarily rely on the Java language, which could raise questions about the generalizability of SCLogger to other programming languages. However, Java is among the most prevalent programming languages for logging research purposes, in accordance with previous works [Li et al. 2021; Liu et al. 2022; Mastropaolo et al. 2022]. The core idea of the contextualized prompt construction and the process of static analysis can be generalized to other language with appropriate adaption.

7 RELATED WORK

In this section, we review the related work on empirical studies of logging practices and approaches of automatic logging.

7.1 Studies on Logging Practices

In order to enhance the observability and maintainability of systems, logging practices have been a subject of study [Chen and Jiang 2017, 2021; Ding et al. 2015; Yuan et al. 2012c], which aids developers in adopting more suitable logging strategies. Fu et al. [2014] examined the logging practices in two large-scale online service systems involving experienced industry developers and provided six key findings about logging code categories, decision-making factors, and the feasibility of auto-logging. Furthermore, another industrial study [Pecchia et al. 2015] revealed that logging processes are developer-dependent, highlighting the need for standardizing event logging activities across a company. Researchers have also explored the evolution of logging statements in open software projects [Chen and Jiang 2017; Kabinna et al. 2018; Shang et al. 2014]. These studies found that paraphrasing, inserting, and deleting logging statement operations are widespread during software development. Zhao et al. [2023] investigated the IDs in logging statements and introduced LTID for automatic ID injection based on a log dependency graph. Ding et al. [2023b] delved into the temporal relationships between logging and corresponding source code, leading to the detection of logging-code temporal inconsistencies through logical and semantic temporal relation rules.

Despite the comprehensive studies of logging practices, offering general and automated strategies for effective logging remains challenging. The general experience obtained from above studies is neither automatic nor consistent with the logging style of each project. To bridge the gap, this work is the first automatic one-stop logging statement generation approach with adapted logging style, benefiting further research and real-world application.

7.2 Logging Statement Automation

Traditionally, the logging statement automaton can be divided into two steps based on stages [Chen and Jiang 2021; He et al. 2021]: the selection of logging locations and the generation of logging statements, which we summarize as *where-to-log* and *what-to-log*.

To solve the problem of *where to log*, researchers have tried many approaches [Jia et al. 2018; Li et al. 2020; Yao et al. 2018; Zhao et al. 2017; Zhu et al. 2015] to find the appropriate logging location in the source code. Prior studies [Lal et al. 2016; Yuan et al. 2012b] have tackled the log placement problem within specific code constructs such as *catch* and *if* statements. However, such logging placement can lead to an excess of logging statements, bringing additional system overhead. Log20 [Zhao et al. 2017] was proposed to identify an almost optimal placement of logging statements, guided by information theory and under the constraints of performance overhead. It determines the logging position by evaluating the effectiveness of each logging statement in distinguishing execution paths. With the development of machine learning, data-driven approaches [Li et al. 2020; Zhu et al. 2015] have brought about more possibilities. LogAdvisor [Zhu et al. 2015] will suggest logging positions by learning structural features, textual features, and syntactic features from systems. By introducing deep learning to learn the features from source code, Li et al. [2020] has elevated performance to a new level.

For *what to log*, the process of generating logging statements is generally divided into three subtasks: logging level prediction [Li et al. 2017, 2021; Liu et al. 2022], logging variables selection [Dai et al. 2022; Liu et al. 2019; Yuan et al. 2012c], and logging text generation [Ding et al. 2022; Mastropaolo et al. 2022]. Ordinal-based neural networks [Li et al. 2021] and graph neural networks [Liu et al. 2022] have been utilized to learn syntactic code features and semantic text features to recommend for logging level. LogEnhancer [Yuan et al. 2012c], from a programming analysis viewpoint, aspires to alleviate the complexity of failure diagnosis by incorporating causally-related variables into a logging statement. Meanwhile, LoGenText [Ding et al. 2022] and LoGenText-Plus [Ding et al. 2023a] translates the related source code into short textual descriptions and then generate the logging text using neural machine translation models.

The most recent approach LANCE [Mastropaolo et al. 2022], provides a one-stop logging statements solution of deciding logging points and logging statements for method level java code. Nevertheless, owing to its end-to-end design and limited method-level context, the approach suffers from insufficient context. As a result, it fails to satisfy the practical needs of real-world development scenarios. To address this, our approach for context-aware logging statement generation simultaneously addresses the issues of *where-to-log* and *what-to-log*, providing a practical solution for logging statement automation.

8 CONCLUSION

In this paper, we propose SCLogger, the first contextualized logging statement generation approach with static contexts. SCLogger incorporates static domain knowledge into language models via a context-aware prompt structure and further self-refinement. Experimental results show that SCLogger outperforms all baselines and can be generalized various LLMs. We believe that SCLogger would benefit both developers and researchers in the field of logging statement generation.

ACKNOWLEDGMENT

The work described in this paper was supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14206921 of the General Research Fund). We thank all reviewers for their valuable comments.

REFERENCES

Apache. 2023. log4j. https://logging.apache.org/log4j/2.x/

Boyuan Chen and Zhen Ming Jiang. 2017. Characterizing and detecting anti-patterns in the logging code. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 71–81.

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

Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming Jiang. 2018. An automated approach to estimating code coverage measures via execution logs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 305–316.

Zhuangbin Chen, Jinyang Liu, Wenwei Gu, Yuxin Su, and Michael R Lyu. 2021. Experience report: Deep learning-based system log analysis for anomaly detection. *arXiv preprint arXiv:2107.05908* (2021).

Shaozhi Dai, Zhongzhi Luan, Shaohan Huang, Carol Fung, He Wang, Hailong Yang, and Depei Qian. 2022. REVAL: REcommend which VAriables to Log with pre-trained model and graph neural network. *IEEE Transactions on Network and Service Management (TNSM)* (2022).

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)*. 139–150.

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.

Zishuo Ding, Yiming Tang, Xiaoyu Cheng, Heng Li, and Weiyi Shang. 2023a. LoGenText-Plus: Improving Neural Machine Translation-based Logging Texts Generation with Syntactic Templates. *ACM Transactions on Software Engineering and Methodology* (2023).

Zishuo Ding, Yiming Tang, Yang Li, Heng Li, and Weiyi Shang. 2023b. On the Temporal Relations between Logging and Code. (2023).

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).

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.

Eclipse Foundation. 2023. Eclipse Java Development Tools (JDT) Core. https://www.eclipse.org/jdt/core/

Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE)*. 24–33.

Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, and Michael R Lyu. 2023. Constructing Effective In-Context Demonstration for Code Intelligence Tasks: An Empirical Study. *arXiv preprint arXiv:2304.07575* (2023).

GitHub. 2023. GitHub Copilot: Your AI pair programmer. https://github.com/features/copilot

Ceki Gulcu. 2023. slf4j. https://www.slf4j.org

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).

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 (ASE)*. 178–189.

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.

Yizhan Huang, Yichen Li, Weibin Wu, Jianping Zhang, and Michael R Lyu. 2023. Do Not Give Away My Secrets: Uncovering the Privacy Issue of Neural Code Completion Tools. *arXiv preprint arXiv:2309.07639* (2023).

Yintong Huo, Yichen Li, Yuxin Su, Pinjia He, Zifan Xie, and Michael R Lyu. 2023. AutoLog: A Log Sequence Synthesis Framework for Anomaly Detection. *arXiv preprint arXiv:2308.09324* (2023).

Zhouyang Jia, Shanshan Li, Xiaodong Liu, Xiangke Liao, and Yunhuai Liu. 2018. SMARTLOG: Place error log statement by deep understanding of log intention. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 61–71.

Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R Lyu. 2023a. LLMParser: A LLM-based Log Parsing Framework. *arXiv preprint arXiv:2310.01796* (2023).

Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jiazhen Gu, Zhuangbin Chen, Jieming Zhu, and Michael R Lyu. 2023b. A Large-scale Benchmark for Log Parsing. *arXiv preprint arXiv:2308.10828* (2023).

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

Sangeeta Lal, Neetu Sardana, and Ashish Sureka. 2016. 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, 215–220.

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

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

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29.

Yichen Li, Xu Zhang, Shilin He, Zhuangbin Chen, Yu Kang, Jinyang Liu, Liqun Li, Yingnong Dang, Feng Gao, Zhangwei Xu, et al. 2022. An Intelligent Framework for Timely, Accurate, and Comprehensive Cloud Incident Detection. *ACM SIGOPS Operating Systems Review* 56, 1 (2022), 1–7.

Zhenhao Li, An Ran Chen, Xing Hu, Xin Xia, Tse-Hsun Chen, and Weiyi Shang. 2023a. Are They All Good? Studying Practitioners' Expectations on the Readability of Log Messages. *arXiv preprint arXiv:2308.08836* (2023).

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 (ASE)*. 361–372.

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.

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

Jinyang Liu, Junjie Huang, Yintong Huo, Zhihan Jiang, Jiazhen Gu, Zhuangbin Chen, Cong Feng, Minzhi Yan, and Michael R Lyu. 2023. Scalable and Adaptive Log-based Anomaly Detection with Expert in the Loop. *arXiv preprint arXiv:2306.05032* (2023).

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

Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2019. Which variables should i log? *IEEE Transactions on Software Engineering (TSE)* 47, 9 (2019), 2012–2031.

Antonio Mastropaolo. 2023. LANCE2.0. https://github.com/antonio-mastropaolo/automating-logging-acitivities

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.

OpenAI. 2022. GPT-3.5. https://platform.openai.com/docs/models/gpt-3-5

OpenAI. 2023. ChatGPT. https://openai.com/blog/chatgpt/

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

Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. 2015. Industry practices and event logging: Assessment of a critical software development process. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, Vol. 2. IEEE, 169–178.

Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R Lyu. 2023. Generative Type Inference for Python. *arXiv preprint arXiv:2307.09163* (2023).

Md Rafiqul Islam Rabin, Aftab Hussain, Mohammad Amin Alipour, and Vincent J Hellendoorn. 2023. Memorization and generalization in neural code intelligence models. *Information and Software Technology (Inf. Softw. Technol.)* 153 (2023), 107066.

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. *The Journal of Machine Learning Research (JMLR)* 21, 1 (2020), 5485–5551.

Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

Guoping Rong, Shenghui Gu, He Zhang, Dong Shao, and Wanggen Liu. 2018. How is logging practice implemented in open source software projects? a preliminary exploration. In *2018 25th Australasian Software Engineering Conference (ASWEC)*. IEEE, 171–180.

Guoping Rong, Yangchen Xu, Shenghui Gu, He Zhang, and Dong Shao. 2020. Can you capture information as you intend to? A case study on logging practice in industry. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 12–22.

Guoping Rong, Qiuping Zhang, Xinbei Liu, and Shenghiu Gu. 2017. A systematic review of logging practice in software engineering. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 534–539.

Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E Hassan, Michael W Godfrey, Mohamed Nasser, and Parminder Flora. 2014. 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)* 26, 1 (2014), 3–26.

Tabnine. 2023. Tabnine. https://www.tabnine.com/

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.

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).

Junjielong Xu, Ruichun Yang, Yintong Huo, Chengyu Zhang, and Pinjia He. 2023. Prompting for Automatic Log Template Extraction. *arXiv preprint arXiv:2307.09950* (2023).

Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, DongGyun Han, and David Lo. 2023. What Do Code Models Memorize? An Empirical Study on Large Language Models of Code. *arXiv preprint arXiv:2308.09932* (2023).

Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Steve Sporea, Andrei Toma, and Sarah Sajedi. 2018. Log4perf: Suggesting logging locations for web-based systems' performance monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. 127–138.

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

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

Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012c. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)* 30, 1 (2012), 1–28.

Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. 2017. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 19–33.

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

Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. 565–581.

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

Received 2023-09-29; accepted 2024-01-23