# Automated File-Level Logging Generation for Machine Learning Applications using LLMs: A Case Study using GPT-4o Mini

**Mayra Sofia Ruiz Rodriguez**

**A Thesis**

**In the Department**

**of**

**Computer Science and Software Engineering**

**Presented in Partial Fulfillment of the Requirements**

**For the Degree of**

**Master of Applied Science (Software Engineering)**

**at Concordia University**

**Montréal, Québec, Canada**

**June 2025**

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Mayra Sofia Ruiz Rodriguez

Entitled: Automated File-Level Logging Generation for Machine Learning Applications using LLMs: A Case Study using GPT-4o Mini

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________ Chair
Dr. Nikolaos Tsantalis

______________________ Examiner
Dr. Tse-Hsun (Peter) Chen

______________________ Examiner

______________________ Thesis Supervisor(s)
Dr. Emad Shihab

______________________ Thesis Supervisor(s)

Approved by ______________________
Dr. Denis Pankratov Chair of Department or Graduate Program Director

______________________

Dr. Mourad Debbabi Dean of Gina Cody School of Engine

# Abstract

## Automated File-Level Logging Generation for Machine Learning Applications using LLMs: A Case Study using GPT-4o Mini

**Mayra Sofia Ruiz Rodriguez**

**Concordia University, 2025**

Logging is essential in software development, helping developers monitor system behavior and aiding in debugging applications. Given the ability of large language models (LLMs) to generate natural language and code, researchers are exploring their potential to generate log statements. However, prior work focuses on evaluating logs introduced in code functions, leaving file-level log generation underexplored—especially in machine learning (ML) applications, where comprehensive logging can enhance reliability.

In this study, we evaluate the capacity of GPT-4o mini as a case study to generate log statements for ML projects at file level. We gathered a set of 171 ML repositories containing 4,073 Python files with at least one log statement. We identified and removed the original logs from the files, prompted the LLM to generate logs for them, and evaluated both the position of the logs and log level, variables, and text quality of the generated logs compared to human-written logs. In addition, we manually analyzed a representative sample of generated logs to identify common patterns and challenges.

We find that the LLM introduces logs in the same place as humans in 63.91% of cases, but at the cost of a high overlogging rate of 82.66%. Furthermore, our manual analysis reveals challenges for file-level logging, which shows overlogging at the beginning or end of a function, difficulty logging within large code blocks, and misalignment with project-specific logging conventions. While the LLM shows promise for generating logs for complete files, these limitations remain to be addressed for practical implementation.

# Acknowledgments

I would like to express my sincere gratitude my supervisor, Dr. Emad Shihab, for his continuous support, patient mentorship, and invaluable guidance. His expertise and insightful feedback has been transformative in my professional development and have fundamentally shaped my approach to tackling complex challenges.

I am also thankful to our postdoc, Dr. SayedHassan Khatoonabadi who has been a exceptional collaborator and mentor throughout this journey. I would also like to acknowledge Dr. Ahmad Abdelatiff for his mentoring the first months of my master's program and for his continued guidance on our Sandoz chatbot project.

My journey would not have been the same without my colleagues in the DAS lab. Your support through both academic and personal challenges has truly made all the difference. A special thank you to Chaima and Gideon for your invaluable advice and encouragement every step of the way.

I must thank my family and closest friends who supported me throughout this process. To Poncho and Aeon; even though we live in different countries, I always feel your support, and I hope you know you can count on mine just the same. Thank you to my parents, Alberto and Carmen, and to my brother, Jose Alberto, for their unwavering love and support. Thanks to the music that got me through this: Magdalena Bay's "Imaginal Disk", Mort Garson's "Plantasia", Charli xcx's "Brat", Brutalismus 3000 raves, and the soundtracks of Metroid and Persona 5.

Finally, I am deeply grateful to my boyfriend, Johnny, who has supported me in every conceivable way throughout this process. Thank you for your words, your time, and all the ways you showed up for me. I couldn't have done this without you.

# Dedication

To my sister *Diana*, in loving memory.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Thesis Introduction

Logging plays a critical role in modern software development. It enables monitoring in production environments [42], facilitates performance analysis [62], and supports failure diagnosis and debugging efforts [65, 63, 67]. Effective logging requieres developers to make two critical decisions: what-to-log and where-to-log.

The what-to-log problem refers to deciding the appropiate log ingredients. As illustrated in listing 1.1, a log statement includes the log level (e.g., *info*), which indicates the severity of the log; the log text (e.g., *Evaluation metrics*, which describes the activity being recorded; and the log variables (e.g., *eval_metric*), which capture dynamic runtime information. On the other side, where-to-log concerns the location in the code where the log should be introduced.

Listing 1.1: An example of log ingredients. Log level is info, log text is “Evaluation metrics: ” and the log variable is “eval_metric”

```
log.info(f'Evaluation metrics: {eval_metric}')
```

The complexity of these logging decisions has been a subject of study of researchers, which have created different tools aiming to help developers make better log decisions [31, 9, 12]. However, the introduction of large language models has opened new possibilities for automating software engineering tasks, showing promising results in code generation [26], testing [20], and bug resolution [59].

As a result, the application of LLMs to automate log generation has gained traction in the research community. For instance, Mastropaolo et al. [35] introduced LANCE and later LEONID [37], both models trained on millions of methods to generate log statements at the method level. However, the fine-tuning process for these models requires significant computational resources. To reduce training demands, Xu et al. [61] proposed a framework that uses an in-context learning prompt paradigm with a few carefully selected examples to guide the model to generate logs. Building on this direction, Li et al. [28] evaluated different LLMs' performance on log generation. Their study involves removing log statements from functions and using placeholders where logs should be introduced before prompting the LLM.

Despite these advances, existing research focus exclusively on inserting a single log within a method, failing to provide LLMs with the context of an entire file and overlooking the realistic scenario where multiple logging statements are needed in the file. We focus on file-level logging since many machine learning (ML) files define key variables, configurations, and processes outside functions or classes, making function-level approaches insufficient to capture the full logging context. Furthermore, compared to traditional software, ML applications introduce unique challenges compared to traditional data: there is an inherent dependency on data [2], and their non-deterministic behavior makes it difficult to debug [41]. ML applications show lower log statements than traditional software, averaging one log statement per 1,150 lines of code in ML applications using Python [14]. As a result, using LLMs for this purpose is not systematically explored.

In this study, we aim to investigate how GPT-4o mini performs in the task of logging complete ML files, without specifying any placeholder in the code. To this end, we first curate a list of 371 ML projects on GitHub. We extract all Python files from each project and determine which Python files have at least one log, reducing our dataset to 171 projects. We collect a dataset consisting of 4,073 Python files, then we remove the logs in the files and prompt GPT-4o mini to generate logs for these files. Based on this dataset, we aim to answer the following three research questions:

**RQ1: To what extent does GPT-4o mini log in the same position as human written logs?** We aim to understand human and LLM logging patterns because, while logging is essential for debugging, LLMs' tendency toward verbose outputs [52] might result in excessive logging. We compared the position of logs for both human-written and GPT-4o mini generated logs. We find that the LLM prioritizes comprehensive logging coverage, as shown by its low underlogging rate (4.75%) and moderately high coverage (63.91%). However, that results in generating a large number of logs: 5.15 times more than those written by humans. Even in cases where both humans and the LLM logged, the LLM tends to include more logs overall, with a rate of 68.03%.

**RQ2: How does GPT-4o mini perform at logging ML applications compared to human developers?** We aim to understand the quality of LLM-generated logs compared to human-written logs. To this end, we analyze the generated logs in terms of ingredients: level, variables, and text; and compare them to the corresponding human-generated log. We find that GPT-4o mini shows moderate quality but significant limitations when generating logs compared to human developers. While it matches the log levels with 59.19% exact match and 84.34% average ordinal distance, it identifies only 40.58% of variables that the human logged. Its log texts are similar in meaning to human logs (ROUGE-L: 0.316), but uses different vocabulary and phrasing (BLEU-4: 0.050), and require substantial editing to match human logs (Levenshtein: 0.735).

**RQ3: What are the challenges of using GPT-4o mini for automated log generation?** We aim to identify GPT-4o mini's limitations in file-level logging. For this purpose, we categorize all developer-LLM pairs of logs into four categories: overlogging, underlogging, different levels, and different variables. We get a stratified sample of 384 pairs of logs, and manually analyze the surrounding code in which they were introduced. We find that GPT4-o mini's main challenge is overlogging (85.8%), with most logs introduced at the start or end of a function. Moreover, the LLM underlogs (4.7%), particularly in large code blocks. In cases with different log level (5.3%), the LLM does not align with project-specific logging conventions. Finally, in case of different variables captured between humans and the LLM (4.3%), the LLM sometimes misses important variables that are present in the GT, despite the larger available context.

Based on our findings, we discuss key considerations for using GPT-4o mini in logging complete ML files. While LLMs can generate useful log statements, they often overlog, potentially cluttering code and reducing its effectiveness for debugging. Developers should critically review LLM-generated logs to ensure relevance and avoid excessive logging. Additionally, repository-specific logging configurations, such as custom log levels, are generally overlooked but crucial for log generation in real-world projects. Our results suggest that future tools might benefit from this information to generate higher-quality logs that aligns better with previous developers' preferences.

## 1.2 Thesis Overview

This thesis is organized as follows:

- Chapter 2 describes the literature review. We start by reviewing studies on the use of LLMs in software engineering tasks. Then, we examine machine learning studies related to maintenance and monitoring, followed by empirical studies on logging practices in traditional software engineering code. Finally, we review existing approaches for automating logging statement generation. This final section is organized into three areas: research on *what-to-log* (log level, variables, and text), *where-to-log* (the position of the log in the code), and research on generating complete log statements.
- We detail our study design in Chapter 3. We first describe the characteristics of the GitHub repositories we mined and the process we followed to select files for our study. We then explain how we determine the position of each log in our dataset, define the prompt used in our study, and describe how we paired human-written logs to LLM generated ones.
- We present the results of our three research questions in Chapter 4.
- We discuss the implications of our findings and provide recommendations in Chapter 5.
- We outline the limitations of our study in Chapter 6 where we examine the internal and external validity of our thesis.
- Finally, Chapter 7 presents our conclusion and discusses the key directions for future work.

## 1.3 Thesis Contributions

The main contributions of this thesis are as follows:

- We provide empirical evidence on the effectiveness of GPT-4o mini for automated file-logging generation.
- We discuss the challenges of using GPT-4o mini for file-level log generation based on our manual analysis.

- To promote the reproducibility of our study and facilitate future research on this topic, we publicly share our scripts and dataset.[1]

## 1.4 Related Publications

The work presented in this thesis has been submitted to the following venues:

- **Mayra Sofia Ruiz Rodriguez**, SayedHassan Khatoonabadi, and Emad Shihab. Automated File-Level Logging Generation for Machine Learning Applications using LLMs: A Case Study using GPT-4o Mini. Submitted to IEEE/ACM International Conference on Automated Software Engineering (ASE).

[1]https://zenodo.org/records/15558239

# Chapter 2

# Literature review

In this chapter, we provide an overview of the relevant work to our thesis. We first review studies on the use of large language models in software engineering tasks. Then, we examine studies on machine learning, followed by empirical studies on logging practices. Finally, we review existing approaches for automating logging statement generation.

## 2.1 Large language models for software engineering

Large language models (LLMs), such as GPT-4o mini [44] and Claude Sonnet 4 [3], are deep learning models pre-trained on large quantities of unlabeled data. LLMs are typically based on the transformer architecture [58], that consists of a set of neural networks called an encoder and a decoder.

LLMs have demonstrated string performance in a wide range of natural language processing tasks, as well as coding and software engineering tasks. Some software engineering applications include code generation [26], requirement classification [17], root cause analysis [1], code translation [45] and code review automation [57, 27]. Since most LLMs come already trained in general data and are not really trained in a specific task, fine-tuning has been gaining adoption, which is further training an LLM model on a smaller dataset. For example, Chen et al. [7] introduced Codex, which is a GPT model fine-tuned on code, and they also introduced the evaluation set HumanEval to measure code correctness. Similarly, Hey et al. [17] fine-tuned a BERT model [10] on a set of functional and non-functional requirements for requirements classification. Furthermore, Mastropaolo et al. [36] fine-tuned a T5 (Text-to-Text Transfer Transformer) model [50] to generate code summaries.

Specific to logging, Li et al. [28] evaluated LLMs for automated log generation. However, their study focused on function-level generation and did not explore LLM behavior when provided with a broader context. *Our study addresses this gap by investigating how GPT-4o mini generates log statements for a complete file, and sheds light on how the LLM logs machine learning files.*

## 2.2 Machine learning applications and maintenance challenges

In recent years, machine learning (ML) applications have been surging in popularity, introducing unique challenges that distinguish them from traditional software development. ML applications inherently depend on the training data, which typically evolves and expands alongside the systems they support [47]. This data centric characteristic makes data managing and versioning significantly more complex than traditional software [2].

Maintaining ML systems in production is particularly complex. These systems face unique monitoring challenges to ensure reliability [41] and explainability [22]. Practitioners often struggle to detect issues such as data drift, bias, and failures [41]. ML applications also introduce different forms of technical debt, including data dependencies, and sensitivity to changes in the external world [53]. To be able to detect such changes in data requires close monitoring [4]. Shankar et al. [55] highlight that developers stress the need to log information at all stages of the ML pipeline (e.g. model training) to support future debugging.

Foalem et al. [14] conducted an empirical study of logging practices in ML applications. Their study revealed that ML-based applications use both general logging libraries (e.g., logging) and ML-specific libraries such as Wandb [8] and MLFlow [39]. Their results show that the most frequently used logging levels in ML applications are *info* and *warning*, and the model training stage contains the highest number of logs across the ML pipeline. Importantly, their study revealed that ML applications have a significantly lower log density than traditional software, with one log statement per 1150 lines of code.

*Our thesis builds on this line of work by exploring how LLMs can introduce logs in ML files. We want to explore how GPT-4o mini would perform logging a complete ML file to aid in the low log density seen in ML software.*

## 2.3 Empirical studies on logging practices

Early work in this area includes Yuan et al. [65] empirical study on logging practices in C/C++ open-source projects. Their analysis revealed that developers extensively utilize log statements and regularly modify them to enhance both debugging capabilities and system maintenance processes. Following this research direction, Fu et al. [15] analyzed logging practices across two Microsoft industrial projects, specifically examining which types of code blocks developers typically choose to log. Their analysis showed that developers choose to log unexpected situations where the system would throw an error, and they also log critical execution points (i.e., execution paths).

Performance studies in logging practices have received considerable attention from researchers. Ding et al. [11] developed a cost-aware logging system that makes runtime decisions about introducing a log or not based on a predefined budget. Li et al. [25] investigated the developers' perspective on logging practice. They found that logging increases developers' maintenance efforts, and that log decisions involve a trade-off between the value of introducing a log and the cost of adding it.

The evolution of log statements as part of the software evolution has been extensively studied. Kabinna et al. [19] found that 20% to 45% of introduced logging statements are changed at least once in the project's lifetime. Shang et al. [54] found that the evolution of log statements impacts the traceability of the log monitoring at run-time.

These foundational studies establish our understanding of logging practices as an integral part of software development. Although logging practices have been studied, there is a lack of approaches for evaluating how LLMs handle logging tasks at the file-level. *Our work bridges this gap by providing the first analysis of GPT-4o mini's performance on logging complete ML files and identifying its limitations.*

## 2.4 Automating logging statements

Studies in logging statement automation are divided into addressing the problems of *what-to-log* and *where-to-log*. *What-to-log* studies focus on predicting a single logging ingredient (level, variables or text). In contrast, *where-to-log* studies focus on suggesting the most appropriate location in the code to insert a log statement.

### 2.4.1 What-to-log

A log statement consists of a log level, variables, and text. Research in this area supports developers in selecting an individual component of a log statement.

#### Log Level

Log levels provides information about the importance or severity of the log. While the exact list may vary by programming language and logging library, the list of log levels typically include *info*, *debug*, *warning*, and *error*. For example, *info* level indicates that code is executing as expected, whereas the *error* level informs about a failure in the system. Choosing an appropriate log level is crucial: assigning an *info* level to critical events may cause them to be overlooked, while logging trivial events using the *error* level can generate false alarms and lead to confusion for developers and end users. Therefore, selecting a proper log level is not a trivial task. To address this issue, Li et al. [23] created a technique to suggest a log level based on ordinal regression model. Building on this, Li et al. [31] proposed *DeepLV*, a deep-learning model to recommend logging levels for existing logging statements based on location information of a logging statement. More recently, Liu et al. [33] proposed *Tell* which extended number of code block information to suggest better log levels.

#### Log Variables

Logging variables can provide critical run-time context, aiding in debugging and root-cause analysis. Yuan et al. [66] introduced *LogEnhancer*, which identifies relevant variables into log statements using information related to failure diagnosis. However, their approach adds as many variables as possible and is intended to be used after software release, limiting its applicability during development. To address these limitations, Liu et al. [34] propose a neural network model that generates log variables based on the code preceding the log statement. In contrast to *LogEnhancer*, their approach allows developers select variables during development while prioritizing certain variables, thus improving performance. Building on this work, Dai et al. [9] proposed *REVAL*, a tool that combines the pre-trained model BERT [10] and a graph convolutional network to analyze the semantic context of the code and recommend a set of variables to log.

**Log Text**

The log text gives the developers information about the process being executed at run-time. Research in this area focuses on generating relevant log texts for developers. For example, Ding et al. [12] proposed *LoGenText*, which uses a neural machine translation model to generate log texts based on the source code and context information such as the location of the log statement. Later, Ding et al. [13] improves on their approach introducing *LoGenText-Plus*, which extends *LoGenText* by introducing an intermediate step using syntactic template information. The neural machine translation model first generates a template, then that template is used to generate a logging text.

### 2.4.2 Where-to-log

This research area focuses on determining the optimal locations for inserting log statements. This decision is critical in terms of balancing cost-benefit from logs: inserting a small amount of log statements may result in missing critical events, while adding a high amount of log statements may cause important information to be overlooked besides increasing storage and resources cost [69].

Initial studies primarily focused on identifying code where exceptions might occur, but lacked a corresponding log. The absence of logging statements in such cases left developers with little insight about the failures encountered. To address this issue, Yuan et al. [64] proposed Errlog which searched for potential exceptions in the code while avoiding performance overhead. Then, Zhu et al. [69] proposed *LogAdvisor*, a machine learning model to suggest developers where to introduce a log, which was trained on logs introduced by developers.

Later, Jia et al. [18] proposed SmartLog, a tool that detects where to insert error log statements based on the intention (i.e., semantics) of existing logs. Zhao et al. [68] proposed *Log20* which determines the placement of log statements by considering all possible execution paths. Li et al. [30] analyzed the locations where developers introduced logs and proposed a deep learning model to suggest log locations. Cândido et al. [6] demonstrated that ML models trained on open source code can be effectively applied in industrial settings, achieving 79% accuracy, 81% precision, and 60% recall.

Additional studies have investigated the characteristics of code snippets that tend to contain log statements. For example, Li et al. [24] identified a strong correlation between the topics of a code snippet (i.e., thread interruption) and the likelihood that it includes a log statement.

### 2.4.3 Generating a complete log statement

Recently, there have been studies that tackle the problem of generating complete logs by deciding level, variables, text, and where the log should be introduced. Mastropaolo et al. [35] introduced LANCE which is a tool that automatically generates a proper log level, text, variables, and location. Mastropaolo et al. [38] then improved on their approach with LEONID, which also generated a complete log as LANCE, but it also has the capability of deciding if a method needs a log or not. Xu et al. [61] proposed a warmup and in-context learning (ICL) approach to enhance log generation. Since this is a prompt approach, it can be used with any LLM as backbone. Xie et al. [60] proposed FastLog, which uses a token classification model to locate where a log statement should be added, and then uses a Seq2Seq model to generate a complete logging statement for that position. Li et al. [29] proposed SCLogger, which introduces domain knowledge to extend the context of the prompt to include logging-related context. Tan et al. [56] introduced AL-Bench, which introduces a dynamic evaluation of logs at run-time. Their study revealed that 20.1% to 83.6% of the generated log statements by *FastLog*, *UniLog*, *LANCE* and *LEONID* are unable to compile.

*Current tools target function-level log generation for general applications, our work complements existing work by examining file-level log generation in ML projects using GPT-4o mini. This larger context introduces unique challenges, such as overlogging, adapting to project-specific logging conventions, and managing variables available in external classes.*

# Chapter 3

# Methodology

In this chapter, we present the study design of our thesis, including the selection of GitHub repositories, and the methodology to match human-written logs to LLM-generated ones.

## 3.1 Study Design

The objective of our research is to evaluate the effectiveness of GPT-4o mini generated logs for ML applications at the file-level. Figure 3.1 provides an overview of our approach, which includes data collection, preprocessing, generating GPT-4o mini logs, and evaluation phases. We explain the details of each phase in the following sections.

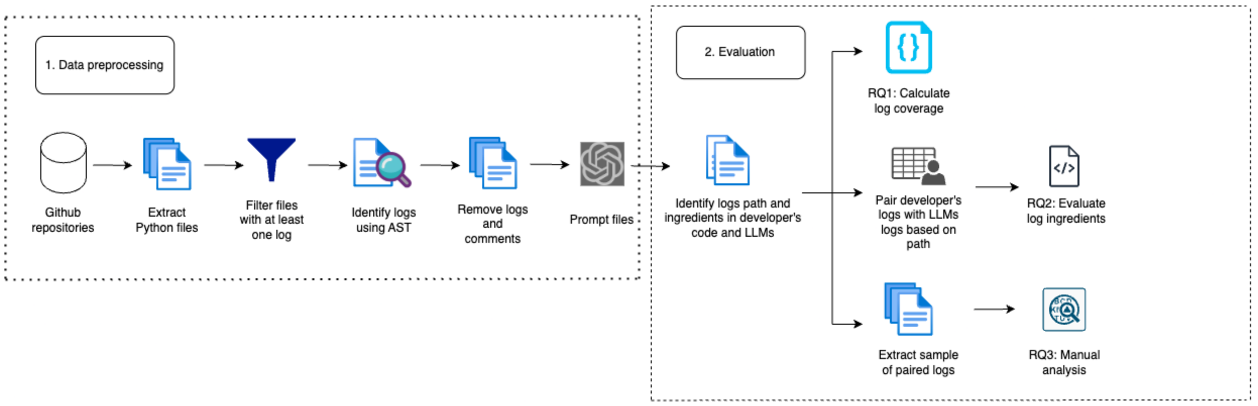
1. Data preprocessing
Github repositories
Extract Python files
Filter files with at least one log
Identify logs using AST
Remove logs and comments
Prompt files
2. Evaluation
Identify logs path and ingredients in developer's code and LLMs
RQ1: Calculate log coverage
Pair developer's logs with LLMs logs based on path
RQ2: Evaluate log ingredients
Extract sample of paired logs
RQ3: Manual analysis

Figure 3.1: Research workflow of this study

Table 3.1: Descriptive Statistics of the Selected Projects

| | **Mean** | **Median** | **Min** | **Max** |
|---|---|---|---|---|
| # Stars | 5,393 | 886 | 54 | 133,592 |
| # Commits | 4,000 | 1,159 | 12 | 79,758 |
| # Contributors | 78 | 25 | 3 | 455 |
| # Days since last push | 78 | 27 | 0 | 364 |
| Age (years) | 7.5 | 7 | 3.2 | 15.1 |

### 3.1.1 Dataset

**Project Selection.** To analyze logging in ML projects, we need a comprehensive collection of ML repositories. For this purpose, we use the dataset originally developed by Gonzalez et al. [16] and later revised by Rzig et al. [51]. This dataset represents one of the most comprehensive collections of ML projects for research purposes.

The original dataset was built by identifying GitHub topic labels specifically related to ML and collecting matching projects via the GitHub API. Rzig et al. further improved the dataset through manual analysis, removing non-ML and trivial projects, adding an additional layer of human verification. The final dataset consists of 4,031 ML projects including both ML frameworks such as TensorFlow and ML applications such as Faceswap.

To select projects appropiate for our thesis, we employed the GitHub API to mine the metadata for each project from the above dataset as of October 2024. We filter our dataset by choosing repositories that meet the following criteria:

- We chose Python as the target language for our study, due to its popularity in ML tools and applications [16].
- We focus on projects with at least 50 number of stars and at least three contributors to filter out toy projects.
- We select projects that are active by filtering the projects that have their last push operation within the last 365 days. This ensures that the selected projects capture the most current logging activity from developers.

This filtering process concludes with a dataset of 371 ML projects. With this dataset, we create a shallow clone of each repository to extract its Python files, resulting in a collection of 105,332 Python files. To focus our analysis specifically on files that contain logs, we applied an additional filter to identify Python files using the Python *logging* library and containing at least one log statement. This filtering process results in a total of 4,073 files across 171 projects. Table 3.1 contains descriptive statistics of the selected projects. To prepare the files for our analysis, we removed all existing log statements and comments.

## 3.2 Methodology

### 3.2.1 AST Visitor

We utilize the Abstract Syntax Tree (AST) Python module [49] to precisely locate logging positions within the source code. The AST provides a tree representation of the abstract syntax tree of source code, where each node corresponds to different syntactic elements such as class definitions, function declarations, control flow constructs (i.e., *if*, *for*, and *while* statements), exception handling blocks (i.e., *try*, and *except* statements), and other node types defined in the AST module documentation. This allows us to analyze logging placement in terms of their position in the AST tree, rather than the line of code. For each file in our dataset, we generate an AST and utilize the visitor pattern to traverse and examine each node within the tree structure.

**Define path.** In this study, we define ”path” as the sequence of consecutive code blocks where a log is positioned. While traversing the AST, we maintain a record of the visited code blocks (i.e., classes, functions, for statements). First, we initialize the path as ”global” to refer to the log statements located outside functions and classes. As we navigate through each code block type, we employ a stack data structure to track the nodes we enter. For each node, we examine whether it contains a log statement. When we exit the node, we remove the code block name from our stack, thus updating the current path definition.

We address cases involving multiple code-blocks with identical names, such as multiple *if* statements at the same hierarchical level within a function. For such cases, we track the nodes: *if, for, else, while, try, except,* and *with*. When the AST visitor encounters these nodes, we extend our path definition. Instead of simply appending the node name, we also append a sequential number to distinguish between repeated instances of the same node type within the same code block. In Figure 3.2, the log statement on line 3 will have the path ”global/Analysis/__init__”. The log statement on line 6 will have the path ”global/Analysis/__init__/if1”, while the log statement on line 10 will have the path ”global/Analysis/__init__/if2”.

```
1  class Analysis():
2      def __init__(self, data, message):
3          logger.info("Initializing")
4  
5          if data is None:
6              logger.debug("Data not yet available")
7          else:
8              logger.debug("Data available")
9          if message is None:
10             logger.debug("No message received")
```

Figure 3.2: Example Python code

Nodes like *else* are inherently dependent on their corresponding *if* statements. In our approach, we prioritize the order of node entry and deliberately do not associate *if*/*else* or *try*/*except* pairs with a shared numerical identifier. For example, in line 8 of Figure 3.2, since we visit the *if* node, then exit and enter the *else* node, we report the path as ”global/Analysis/__init__/else1”.

Our approach allows us to assign each log a position based on its location in the AST, rather than its line number in the source code. Finally, after identifying all log statements in our dataset with their corresponding paths, we drop them from the files and prompt GPT-4o mini to generate the logs for the cleaned code files.

### 3.2.2 Prompt Crafting

After preprocessing our files, we use the OpenAI API to prompt GPT-4o mini (*gpt-4o-mini-2024-07-18*) to generate logs for each file. We use this LLM because, at the time our study, it was a state-of-the-art model with a large context window of 128,000 tokens and a max output of 16,384 tokens, which is sufficient to process and output complete files. In addition to its context window, GPT-4o mini is a cost-efficient model [44], making it practical for our use case.

Following OpenAI's prompt engineering guidelines [43], we construct a prompt illustrated in Figure 3.3, where the ”$SOURCE_CODE” variable is dynamically replaced with the file's content. We do not specify the exact number or position of logs, as our goal is to observe how GPT-4o mini chooses to log when asked to introduce logs for an entire file, and compare this behavior to human logs. We also set the temperature to 0 to ensure a more deterministic output [28].

The prompt is structured into five key components:

❶ **Persona and context.** We start the interaction by requesting the LLM to adopt the persona “expert machine learning developer”, and we inform that it will receive a Python file as input.

❷ **Review.** We instruct the LLM to review the provided file.

❸ **Log statement generation.** We ask the LLM to generate log statements using Python's *logging* library. We use this library due to its prevalence in ML applications [14]. While ML-specific logging libraries such as Wandb and MLflow exist, their usage was limited in our pre-filtered dataset of 371 projects (before filtering for files that contain at least one *logging* library log). Only 68 files imported Wandb and 777 imported MLflow, out of 105,332 files. Moreover, we found just 55 instances of wandb.log and 529 instances of mlflow.log (including mlflow.log_metrics, mlflow.log_param, etc.). Focusing on *logging* allowed us to work with a larger number of examples for comparison.

❹ **Log quality instructions.** This section defines the expected log quality through precise guidelines. Each point refers to a specific log quality instruction: where-to-log and what-to-log.

❺ **Task specification.** The task the LLM is instructed to do is to return the complete file, ensuring that the only modifications are the newly introduced log statements.

### 3.2.3 Mapping Ground Truth and LLM logs

Our study involves submitting complete source files to GPT-4o mini for logging, without specifying where or how many logs the LLM should insert. The LLM is free to add as many logs as it finds necessary, often exceeding the number of logs in the ground truth (GT). Due to this scenario, it can become complex to evaluate the quality of the generated logs. Additionally, when both GT and LLM logs include multiple logs in the same code block, it becomes difficult to determine which GT log corresponds to which LLM log.

To facilitate evaluation, we establish pairings between GT and LLM-generated logs. This also helps evaluate pairs of logs that serve similar purposes or capture comparable information. We define four mapping possibilities:

- **1:1 pairing:** A scenario where exactly one log exists in the same path for both GT and LLM files. This represents the ideal case for direct comparison, where both the human and the LLM decided to log the same location and frequency.
- **1:n pairing:** A path contains one GT log and multiple LLM-generated logs.
- **n:1 pairing:** A path contains multiple GT logs and a single LLM-generated log.

❶ You are an expert machine learning developer. You will receive a Python file. Follow these instructions:

❷ 1. Review the provided Python file

❸ 2. Add any missing log statements using the logging library.

❹ 3. Verify that each logging statement is in an appropriate position within the code.
4. Check the logging level of each logging statement to ensure it aligns with its importance.
5. Evaluate the quality of the log texts, ensuring they cover important details and follow best practices.

❺ 6. Return only the complete code snippet with all necessary log statements added. Do not modify the rest of the code.

❻ This is the file: $SOURCE_CODE

Figure 3.3: Prompt template for automated logging

- **n:n pairing:** Multiple logs are present in both GT and LLM within the same path.

Figure 3.4 illustrates these four pairing scenarios and the matching process. We directly pair the 1:1 cases, since these logs were added in the same path and in the same quantity by both humans and the LLM. For the next three cases (1:n, n:1, and n:n), we use cosine similarity on the complete log statement to pair each GT log with its most similar LLM-generated log. Our goal is to match each GT log to a corresponding LLM-generated log, so we can evaluate the quality of each LLM-generated log based on its code positioning. Our matching process ensures that all GT logs are matched with their semantically closest LLM-generated log. In case of 1:n pairing and n:n pairing, some LLM logs may remain unmatched if they do not represent the closest semantic match to any GT log within the path they are located in. For instance, if one GT log corresponds to two LLM logs on the same path, we only pair it with the LLM log showing higher cosine similarity, effectively excluding the other LLM log from evaluation. To summarize, all GT logs are retained in our analysis, and each one of them will be matched to an LLM log unless the LLM produced no logs for that particular path.

## 3.3 Evaluation

Using AST, we simultaneously identify log statements and their corresponding paths. We utilize two techniques to extract the log ingredients: regular expressions are used to extract variables within the log statements, while AST nodes provide information about the log level and the log text. We also remove format specifiers (i.e., *%s*, *%d*) from log texts to ensure a fair comparison. Following the path-based pairing of GT and LLM-generated logs, we proceed to evaluate the effectiveness of GPT-4o mini for generating log statements.

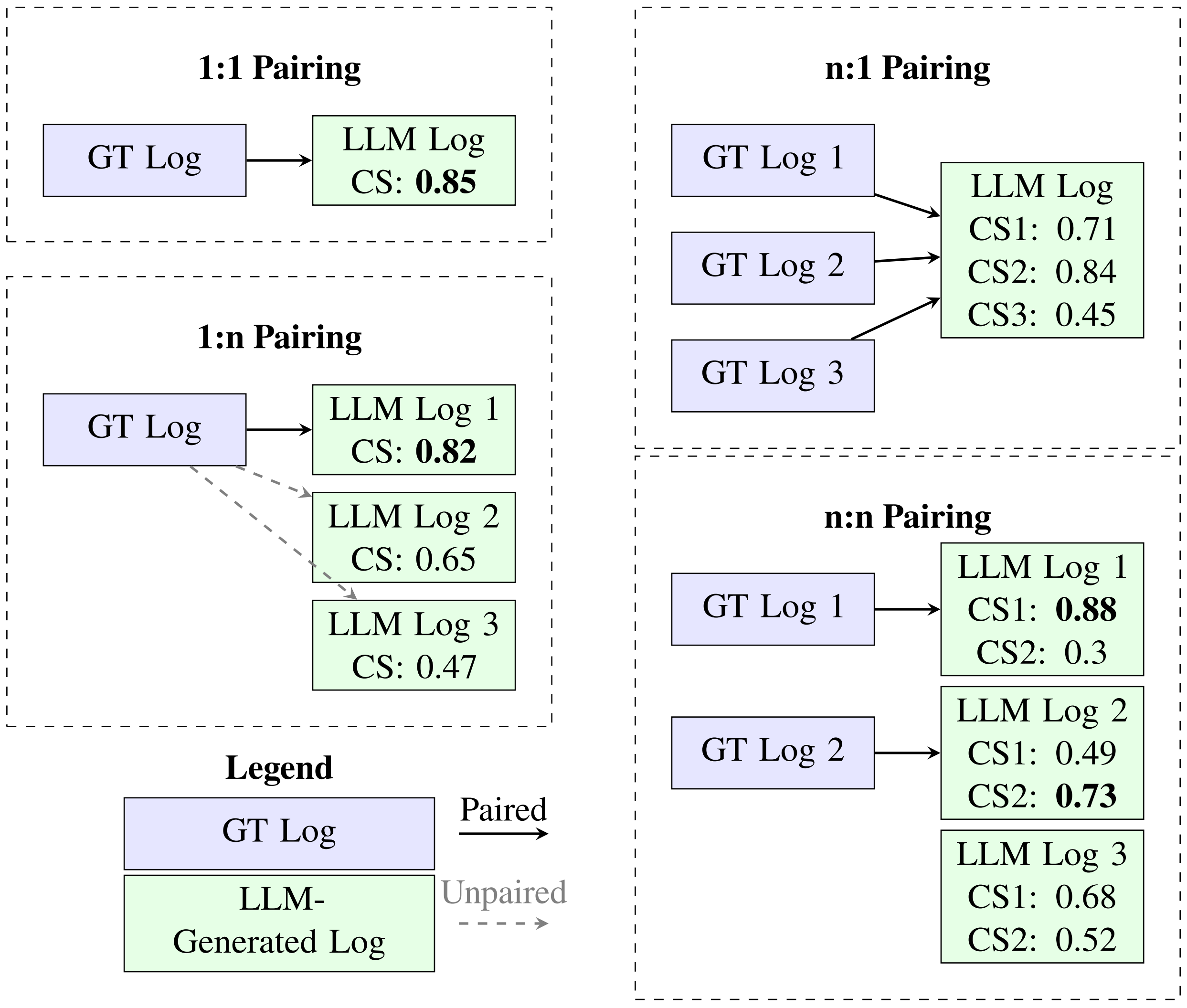

Figure 3.4: Log pairing process between GT and LLM-generated logs. Each GT log is matched with its most similar LLM log based on cosine similarity (CS).

# Chapter 4

# Results

In this chapter, we present the results of our thesis. For each research question, we include the motivation behind it, describe our approach and report the corresponding findings.

## 4.1 RQ1: To what extent does GPT-4o mini log in the same position as human written logs?

### 4.1.1 Motivation.

Logging is critical for recording run-time information and aids developers in debugging. However, effective logging requires a balance: logging too little can result in missing critical events, while excessive logging can cause important information to be overlooked, besides increasing storage and resources cost [69].

Since LLMs tend to generate verbose answers [52], they may introduce more logs than a human would when prompted. This research question explores how the quantity and placement of logs compare between human-written and GPT-4o mini generated logs. Specifically, we want to determine if GPT-4o mini places logs in the same code blocks as humans.

### 4.1.2 Approach.

To analyze how GPT-4o mini generates logs compared to our GT, we begin by extracting all log statements from each file in our dataset. For each log, we collect its ingredients: the log level, variables, text, and its location in the code. We determine location information using the AST blocks the log was introduced in, as described in Section 3, and we call this the log's code path.

Next, we calculate the number of logs in each log path for both the GT and LLM files by summing the logs per path. We then merge these counts based on file name and log path, resulting in a dataset of the number of logs between GT and LLM per path for easier comparison. Using this merged dataset, we calculate the average of the following metrics: **Overlogging** refers to cases where the LLM inserted logs but the GT did not, while **Underlogging** captures the instances where the LLM failed to insert logs in places where the GT did.

To further analyze human and LLM logging practices, for the next two metrics, we filter our merged dataset to include only paths that contain at least one GT log. For these paths, we calculate **Coverage**, as a binary metric indicating whether an LLM-generated log appears in a position where a GT log exists, which is either 0 (Not Covered) or 1 (Covered), and **Quantified Coverage (QC)** as the ratio of LLM logs to GT logs in each path.

In summary, these are our four key metrics:

- **Overlogging:** The proportion of paths where the LLM added logs but the GT did not. If a path contains zero GT logs and at least one LLM log, we mark that path as overlogged, and we report the average of these cases across the dataset.
- **Underlogging:** The proportion of paths where the GT added logs but the LLM did not. If a path contains one or more GT logs but the LLM had zero logs, we mark that path as underlogged, and we report the average of these cases across the dataset.
- **Coverage:** A binary indicator of whether the LLM inserted at least one log in a path where the GT logged. For each path with at least one GT log, if the LLM also includes one or more logs, we mark it as covered (1); otherwise, it is marked as not covered (0).
- **Quantified Coverage (QC):** The ratio of LLM logs to GT logs in each path where the GT logged. For example, if a path has 3 GT logs and 2 LLM logs, the QC is 2/3 since the LLM missed one log in that position. A QC value of 1 indicates an exact match in log count between the LLM and the GT in that path.

Table 4.1: Coverage and verbosity metrics based on log placement across code paths

| **Metric** | **Result** |
|---|---:|
| Coverage (binary) | 63.91% |
| Quantified Coverage (ratio) | 68.03% |
| Overlogging | 82.66% |
| Underlogging | 4.75% |

### 4.1.3 Results.

Table 4.1 presents our results based on log placement. GPT-4o mini achieved a Coverage rate of 63.91%, which indicates a moderate level of similarity in adding logs in the same code paths as GT. This reflects that the LLM generally captures human behavior about which particular points in the code need to introduce a log. However, Coverage does not take into consideration the amount of logs added in those blocks. To address this, we examine the QC, which resulted in 68.03%. This result shows that even in locations where the GT and LLM agree that a log is needed, the LLM tends to insert slightly more quantity of logs than human developers would. This reflects a tendency toward verbose logging at those instances.

This tendency becomes clearer when we examine the Overlogging rate of 88.66%, which strongly confirms previous observations about LLM verbosity [52]. This indicates that GPT-4o mini frequently introduces logs in code blocks where human developers determined logging was unnecesary. Such behavior may reflect GPT-4o mini's preference to be on the side of caution, trying to log important information, but at the risk of adding excessive logs in production environments.

In contrast, the low Underlogging rate of 4.75% suggests that GPT-4o mini is unlikely to miss logging positions that humans consider important. This difference between Overlogging and Underlogging rates suggests that the LLM prioritizes comprehensive coverage, rather than missing logging critical paths in the code.

Our results show that GPT4-o mini prioritizes comprehensive logging coverage, as shown by its low underlogging rate (4.75%) and moderately high coverage (63.91%). However, that results in generating a large number of logs, which is 5.15 times more than those written by humans. Even in cases where both humans and GPT-4o mini logged, the LLM tends to include more logs overall, with a rate of 68.03%.

## 4.2 RQ2: How does GPT-4o mini perform at logging ML applications compared to human developers?

### 4.2.1 Motivation.

In this research question, we aim to understand the quality of logs generated by GPT-4o mini in comparison to human-written logs. Specifically, we focus on instances where GPT-4o mini placed a log in the same position as the human developer. We evaluate GPT-4o mini performance based on three logging ingredients: levels, variables, and text. Understanding how well GPT-4o mini performs at generating these elements is crucial for evaluating its potential as a logging generator tool in software development, particularly in the context of ML applications where logging plays a critical role in monitoring model pipelines.

### 4.2.2 Approach.

We evaluate log quality by first matching each LLM-generated log with its corresponding GT based on their paths (see Section 3). To ensure a fair comparison, we calculate the evaluation metrics described below only for log pairs that share the same code path. Once matched, we assess the quality of the logging ingredients as follows:

**1. Logging Levels**

Following prior work [28], [31], we use two key metrics:

- **Level Accuracy (L-ACC)** measures the percentage of log statements where GPT-4o mini correctly generates the exact same logging level as the GT.
- **Average Ordinal Distance (AOD)** quantifies how far off the predicted logging level is from the correct one. A higher AOD indicates closer alignment, even if the exact level was not predicted. We follow the Python logging library's standard logging levels in ascending severity: *debug*, *info*, *warning*, *error*, and *critical* [48]. The formula is:

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

where $N$ represents the total number of log statements across the dataset, while $Dis(a_i, s_i)$ denotes the distance between the actual logging level $a_i$ and the generated logging level $s_i$. $MaxDis(a_i)$ represents the maximum possible distance from the actual log level $a_i$ to the most distant level in the severity scale.

## 2. Logging Variables

Each log may include zero or more variables. We define Variable Coverage as the proportion of GT variables correctly captured by GPT-4o mini. For example if the GT contains {*var1, var2, var3*} and GPT-4o mini outputs {*var1, var3*}, then Variable Coverage = 2/3.

## 3. Logging Text

We evaluate the generated texts based on the following metrics used in Natural Language Processing (NLP) to evaluate automatically generated text:

- **BLEU** evaluates the quality of automatically generated text by measuring how many n-grams (short word sequences) it shares with the reference text [46]. A higher score indicates that more of the original phrasing is preserved. The score ranges from 0, meaning the texts are completely different, to 1, meaning they are identical. BLEU can be calculated using four different values of n-grams, such as BLEU-1 through BLEU-4.
- **ROUGE** is a set of metrics used to evaluate summarization tasks [32]. We use the ROUGE-L variant, which is based on the Longest Common Subsequence (LCS). It identifies the longest sequence of words that appears in both texts in the same order, though not necessarily consecutively. The values range from 0 to 1, with higher values indicating better similarity to the reference text.
- **METEOR** is a metric based on the harmonic mean of unigram precision and recall, with recall weighted higher than precision [5]. METEOR considers synonyms, word stems, and sentence structure. It is designed to better align with human judgment and penalizes cases where the correct words are present but arranged incorrectly. The score ranges from 0 (completely different) to 1 (identical).
- **Levenshtein distance** is a metric for measuring the difference between two texts by calculating the minimum number of single-character edits required to change one text into the other [21]. This metric provides an estimate of how much effort would be needed to modify the generated log so that it matches

Table 4.2: Evaluation metrics on log ingredients generated by GPT-4o mini

| **Ingredient** | **Metric** | **Results** |
|---|---|---|
| Level | Level Accuracy (L-ACC) | 59.19% |
| | Average Ordinal Distance (AOD) | 84.34% |
| Variables | Variable Coverage | 40.58% |
| Text | BLEU-1 | 0.178 |
| | BLEU-2 | 0.095 |
| | BLEU-4 | 0.050 |
| | METEOR | 0.328 |
| | ROUGE-1 | 0.341 |
| | ROUGE-2 | 0.121 |
| | ROUGE-L | 0.316 |
| | Levenshtein | 0.735 |

the GT. Following prior work [38], we utilize the normalized token-level Levenshtein distance (NTLev) between the generated log text and the GT. Normalization involves dividing the computed distance by the length of the text, resulting in a score between 0 (identical) and 1 (completely different).

### 4.2.3 Results.

Table 4.2 presents the performance of GPT-4o mini in generating each log ingredient at the file level.

#### 1. Logging Levels

GPT-4o mini exactly matches the level defined by developers in about 59% of the cases across all the dataset. However, the AOD score of 84% indicates that, even when the LLM fails to generate the exact level, it often selects a level close in severity.

#### 2. Logging Variables

GPT-4o mini achieves a Variable Coverage of 40%, indicating that GPT-4o mini frequently misses the variables included in GT logs. This suggests that GPT-4o mini finds it difficult to identify the most relevant run-time data to include in the log, despite the file-level context giving access to a broader set of declared variables.

**3. Logging Text**

GPT-4o mini's generated log texts demonstrate moderate alignment with the GT. The BLEU scores (BLEU-1: 0.178, BLEU-2: 0.095, BLEU-4: 0.050) suggest that the model produces texts with relatively few exact n-gram overlaps with the GT logs. However, a METEOR score of 0.328 and ROUGE scores (ROUGE-1: 0.341, ROUGE-2: 0.121, ROUGE-L: 0.316) indicate that the generated texts capture the general meaning of the GT logs.

Overall, these results indicate that GPT-4o mini's generated log texts capture the meaning of the GT log texts (moderate ROUGE score), but it is using a different vocabulary or phrases the texts differently (low BLEU score). This is also supported by the high Levenshtein distance of 0.735, which reveals that substantial editing would be required to transform the GPT-4o mini generated texts into their human-written counterparts.

GPT-4o mini shows moderate quality but significant limitations when generating logs compared to human developers. While it matches the log levels with 59.19% exact accuracy and 84.34% AOD, it identifies only 40.58% of variables that human logged. Its log texts are similar in meaning to human logs (ROUGE-L: 0.316) but uses different vocabulary and phrasing (BLEU-4: 0.050), and require substantial editing to match human logs (Levenshtein: 0.735).

## 4.3 RQ3: What are the challenges of using GPT-4o mini for automated log generation?

### 4.3.1 Motivation.

Previous research has mostly evaluated the generation of logs at the function level, either by providing a specific line of code where to log [28] or by asking the LLM for the specific line where the log should be added [61]. In contrast, our study focuses on evaluating LLMs in the task of generating logs for a complete file. In this research question, we aim to understand the specific challenges that arise when using an LLM to log complete files. Our goal is to identify the limitations and opportunities.

### 4.3.2 Approach.

We classify all the LLM-generated logs paired with human logs into four categories:

Table 4.3: Categories of challenges in LLM-generated logs

| | **# Paired logs** | **# Sampled logs** | **Proportion** |
|---|---:|---:|---:|
| Underlogging | 5,382 | 18 | 4.7% |
| Overlogging | 98,208 | 328 | 85.9% |
| Different Level | 6,018 | 20 | 5.2% |
| Different Variables | 4,914 | 16 | 4.2% |
| Total | 114,522 | 382 | 100% |

- **Underlogging**: The LLM did not introduce a log in a path where the GT did.
- **Overlogging**: The LLM introduced a log in a path where the GT did not.
- **Different Level**: Both LLM and GT introduced a log in the same position, but used different log levels.
- **Different Variables**: Both LLM and GT introduced a log in the same position, but logged different variables. For this category, we consider pairs of logs where both GT and LLM have at least one variable.

Using a 95% confidence level and a 5% margin of error, we obtain a stratified sample of 384 paired logs for manual analysis. Table 4.3 shows the distribution of logs within the stratified sample. For each pair of logs, we manually analyze the GT and LLM files where the log was introduced. This analysis involves examining the surrounding code to better understand the context in which the log statement was introduced. In case of log level, we also analyze the repository's logging level preferences based of the quantity of different levels of severity used in the repository.

### 4.3.3 Results.

We examine each category of logging differences between GPT-4o mini and GT below.

**Overlogging** was the most frequent issue found in our dataset (85.8%). After manual analysis, we found that most of these instances reflected that the LLM often inserts logs at the start or end of functions (179/328) or blocks (120/328). In most examples, the LLM interpreted the beginning of a function or a block as an important process that was about to begin, even when the function was not implemented, as shown in Listing 4.1.

Table 4.4: Results of manual labeling based on source code analysis where a log was introduced

| **Category** | **Label** | **# Instances** |
|---|---|---|
| Overlogging | Log at the beginning/end of a function | 179/328 |
| | Log at the beginning/end of a block | 120/328 |
| | Logging about start/completion of a process, with variables | 151/328 |
| | Logging about start/completion of a process rather than variables/state | 112/328 |
| | Log before an exception | 40/328 |
| | Log reflects state of the code | 29/328 |
| Underlogging | LLM not logging variable changes in long block | 5/18 |
| | Missing process status in condition | 10/18 |
| | Others | 3/18 |
| Different Level | Same intent (mismatch between info/debug) | 8/20 |
| | LLM level is more appropriate | 5/20 |
| | GT level is more appropriate | 2/20 |
| | In level mismatch, the level used by developers was the most used log level in that project | 13/20 |
| Different Variables | The goal/focus of the log is completely different. This shows that the logs have completely different purposes | 7/16 |
| | The goal/focus of the log is similar to GT, but the variables logged are completely different (missing relevant details/information) | 3/16 |
| | The goal/focus of the log is similar to GT, but the variables logged are partially different (partial missing relevant details/information) | 5/16 |

Listing 4.1: An example of overlogging: Not implemented function with introduced log

```
def sync_optimizer(self, optimizer: Optimizer):
    logger.info("Syncing optimizer across processes.")
    pass
```

Moreover, when logging about the start or completion of a process, the LLM tends to include variables (151/328). However, it also frequently omits them (112/328). For example, in Listing 4.1 the log message references the optimizer process, but fails to include details about the optimizer object itself despite it being a function parameter. The LLM exhibited a pattern of adding logs immediately before raising exceptions (40/328), as demonstrated in line 5 of Listing 4.2. Line 2 of listing 4.2 shows an example of the type of variables the LLM would include in logs, which are generally variables part of that block and it can infer variables that are part of an object if they were used later in the code.

Listing 4.2: An example of overlogging: Line 2 shows a log added at the start of a function, while log 5 represents a log introduced just before raising an exception

```
1  def _intersect(self, dataloader):
2      logger.debug("Calculating intersection with dataloader of
           type: %s", type(dataloader.backend))
3      if not isinstance(dataloader.backend, NoBackend):
4          msg = 'Intersection can only be calculated between same
               backends (NoBackend), instead get {}'.format(type(
               dataloader.backend))
5          logger.error(msg)
6          raise Exception(msg)
```

**Underlogging** occurred less often (4.7%), but typically involved missing variable changes in long blocks of code (5/18), or failing to log process state within conditions (10/18). While nearly half of the overlogging cases were logs placed at the beginning of code blocks, in underlogging we observed instances where GPT-4o mini failed to insert logs within long code blocks (i.e. a long main function) even when introducing new inner blocks (i.e. *for* loop). This pattern is demostrated in listing 4.3, where the GT logs before evaluation on line 112, in contrast, the LLM does not log until the ending of the main function. This suggests that GPT-4o mini has trouble understanding what is important in large chunks of code. Also, the LLM seems to infer what the code is doing by heavily relying on the function name where the log is added.

Listing 4.3: An example of underlogging: The LLM miss logging in long blocks of code

```
def main():
    # 153 lines before this one
    # GT log
    logger.info("Evaluate the following checkpoints: %s",
        checkpoints)
    for checkpoint in checkpoints:
        global_step = checkpoint.split('-')[-1] if len(
            checkpoints) > 1 else ''
        model = load_model(args, checkpoint=checkpoint)
        result = evaluate(args, model, tokenizer, prefix=
            global_step)
        result = dict(((k + ('_{}'.format(global_step) if
            global_step else ''), v) for (k, v) in result.items()
            ))
        results.update(result)
    # LLM log
    logging.info('Main function completed.')
    return results
```

**Different levels** (5.3%) frequently involved level mismatches between *info* and *debug* (8/20). In cases where the mismatch involved other levels, such as *debug* vs. *warning*, the log level chosen by the LLM (5/20) or by the developer (2/20) was more appropiate based on the severity level defined by the Python logging library [48]. A key insight was that in the majority of level mismatches (13/20), the developers chose levels that matched the most commonly used level in that particular project, suggesting that project-specific logging preferences influence log level selection. Listing 4.4 shows an example of logs where GT and LLM exhibit same logging intent, but use different levels.

Listing 4.4: An example of logs with different level: Log with the same intent but different log level

```
for i, point in enumerate(datapoints):
    if i % log_every_n == 0:
        # GT log
        logger.info("Featurizing datapoint %i" % i)
        # LLM log
        logger.debug(f'Featurizing datapoint {i}/{len(datapoints)
            }.')
        try:
            features.append(self._featurize(point, **kwargs))
        except:
            features.append(np.array([]))
```

**Different variables** (4.3%) often reflected different log intents (7/18), suggesting that different log purposes led to different logged variables. GPT-4o mini captured some but not all of the variables that developers did (5/16), indicating a partial understanding of which variables are important in each code block. In a few cases, the LLM missed all the variables that developers considered important to log (3/18).

One of the main challenges is that GPT-4o mini seems to not have enough context of external class definitions, therefore, it is unable to suggest relevant variables external to the file. Listing 4.5 shows an instance where the LLM lacked context about the objects defined in imported classes.

Listing 4.5: An example of logs with different variables: LLM does not have context of Topology's object variables

```python
def _debug_dump_topology(topology: Topology, resource_manager:
    ResourceManager) -> None:
    for i, (op, state) in enumerate(topology.items()):
        # GT log
        logger.debug(
            f"{i}: {state.summary_str(resource_manager)}, "
            f"Blocks Outputted: {state.num_completed_tasks}/{op.
                num_outputs_total()}"
        )
        # LLM log
        logger.debug(f'Debug dump for operator {op}: {state}')
```

Our manual analysis of GPT-4o mini generated logs shows that the main challenge is overlogging (85.8%), with most logs introduced at the start or end of a function. Second, the LLM underlogs (4.7%), particularly in large code blocks. Third, in cases with different log level (5.3%) the LLM does not align with project-specific logging conventions. Finally, in case of different variables captured between humans and GPT-4o mini (4.3%), the variables introduced by the LLM sometimes miss important variables that are present in the GT, particularly when those variables were from imported classes outside the file's context.

# Chapter 5

# Implications

This chapter discusses the implications of our findings, providing practical recommendations for developers and researchers.

## 5.1 Review LLM output to reduce overlogging.

Excessive logging can make useful information to be overlooked under a large quantity of logs. Our findings highlight that while GPT-4o mini can generate logging statements that align with general developer intentions, the LLM often overcompensates by adding logs at the beginning or end of functions. This results in the inclusion of logs that trace the application control flow, but do not offer much insights into internal state changes or key variable states, which are more helpful during debugging.

This behavior suggests that LLMs may clutter the code with excessive logs, which reduces the usefulness of logging for debugging and monitoring. Thus, is essential that developers review and filter the logs generated by the LLM. An ideal workflow would include an additional tool to filter out generated logs that are redundant, irrelevant, or do not contribute to failure analysis and program understanding. This helps ensure the logs to be introduced in the code base are informative and useful for developers.

## 5.2 Add context related to imported classes

Modern software development have often modular codebases, with functionality divided across multiple files (i.e., an imported class). Understanding the imported classes behaviour is critical to ensure the variables of the generated log are relevant for debugging.

We observed that GPT-4o mini frequently omits logging variables that are part of imported classes. This limitation is part of the LLMs lack of context about the project structure, since it is not included in the prompt. Without understanding what these imported methods do or how they behave, the LLM can not generate logs that reflect this context.

When prompting an LLM to generate logs, developers should consider adding context about imported classes. This may include class definitions, function names, and input/output variables. Providing this context can help the LLM generate more relevant logs.

## 5.3 Incorporate repository specific logging configurations.

Most projects use standardized logging configurations (e.g., Python's *logging* module default configuration), but others define their own log levels or adjust the severity of existing levels to better align with internal development and repository configuration. For instance, adding a new log level like *TRACE* in Python, or redefining the severity of certain log levels is not uncommon.

Since the LLM's context is the code file, the LLM is unaware of such configurations. As a result, the LLM generates logs based on the standard Python logging library defaults. This may lead to misaligned log recommendations in projects with customized logging.

Developers should incorporate information about their custom logging setups in the prompt, or post-process the result of the LLM to validate that the generated logs align to repository specific configuration. This ensures the generated logs are consistent with the present repository logs and avoids confusion during debugging.

## 5.4 Refine prompt with pipeline step context.

In ML projects, the steps to train a model are called pipeline stages. These include data pre-processing, model training, evaluation and deployment. Each of these states has its own log needs. For instance, during model training it is useful to log metrics like accuracy, while data pre-processing might log the transformations applied to the data.

Our analysis suggests that the LLM tends to produce generic logging statements that do not consider the nature of each pipeline step. This reduces the value of the generated logs in ML workflows.

To generate relevant logs for ML applications, the LLM context can be improved by including a tool that identifies the pipeline stage of a code snippet. For example, if a tool identifies a code snippet related to model training, the prompt could include sample logs from other model training code snippets. This context may help the LLM to focus on important ML-specific events.

## 5.5 File-level benchmarks for automated logging.

Prior evaluations of automated logging tools focus on code snippets. However, in practice, developers work with files, and the utility of the introduced logs depends on the context that a complete file provides.

Our file-level evaluation of GPT-4o mini reveals insights that could not be possible in method-level studies, especially because most studies focus on generating only one log per code snippet. While our thesis provides a useful baseline, it does not directly evaluate the utility of the generated logs during debugging or maintenance tasks. Furthermore, there is little research on how LLM-generated logs perform at runtime. Addressing these challenges is essential for the success of a log automation tool.

# Chapter 6

# Threats to validity

In this chapter, we discuss threats to the validity of our research, and discuss how we addressed potential limitations in our study design. We categorize these threats as internal and external validity.

## 6.1 Internal Validity.

The internal validity of our study is affected by three main factors. First, we use GPT-4o mini for our study, which is pre-trained with information up until October 2023, while our data was collected from GitHub in October 2024. As a result, there is a potential risk of data leakage for repositories that remained unchanged during that period. However, we note that the outputs generated by the LLM are not closely aligned with the GT. This suggests that, even if the LLM had seen parts of the repositories during training, it does not appear to result in memorization or reproduction of the original logs. Therefore, while the theoretical risk of data leakage exists, its practical impact on our findings is likely minimal.

Second, the effects of using different prompts changes the outcome of the results. To minimize this threat, we crafted a prompt that explicitly asks for log quality instructions related to logging placement and logging ingredients.

Third, we made prompt design choices to manage the token limitations of GPT-4o mini that may have affected result quality. To prevent exceeding GPT-4o mini's context window, we removed documentation and existing log comments to avoid token cutoff, while still providing the complete source file to maintain code context. Previous studies [28] have found that prompting LLMs and removing documentation actually decreases the quality of the logging ingredients. In contrast, the same study found that prompting the complete file for logging helps the LLMs get more context about the functions in the code, which translates in better logging quality output from the LLM. Future work should focus on refining prompts to include documentation while incorporating file context, and limiting the prompt to output only the necessary log statements to address the LLM output token limitations.

## 6.2 External Validity.

First, we used GPT-4o mini because, at the time of our study, it was a state-of-the-art model with a large context window. While other LLMs may show different performance, we believe that the challenges identified in the paper would remain important, albeit with different frequencies. Second, we focused our research on open-source projects hosted on GitHub, which may not fully represent the wider spectrum of machine learning projects, especially proprietary ones. To mitigate this, we utilized established best practices in mining software repositories, which are selecting active, popular, and mature ML projects [40]. Furthermore, our dataset includes a diverse range of ML projects, which also have been studied before in the literature. Second, our study was made for Python repositories, which may affect the generalizability of our findings to other languages such as Java. However, previous research found that Python is the most utilized language for machine learning applications [16]. Future work can explore how our findings extend to other languages. Third, we focus on generating logs using Python's *logger* library, and did not explore generating logs for ML-specific logging libraries such as MLFlow or Weights & Biases (Wandb). However, in our pre-filtered dataset of 105,332 Python files, we found only 55 instances of wandb.log and 529 instances of mlflow.log (including mlflow.log_metrics, mlflow.log_param, etc.). Future work could evaluate the quality of LLM generated logs for ML-specific logging libraries using a larger dataset.

# Chapter 7

# Conclusion and Future Work

In this chapter, we present a summary of the thesis and contributions to the field of logging statement generation. We also discuss directions for future research at the end of the chapter.

## 7.1 Conclusion

Developers introduce logs in their code to support failure diagnosis, debugging and monitoring production environments. Researchers have analyzed the developers' logging practices, and have introduced tools which help developers choose proper log levels, variables, texts, and position of the log. Recently, studies have proposed tools that are a one stop tool to generate a log including position, variable, level and text. However, previous studies focus on generating logs for a code snippet, in this thesis we tackle the problem of generating logs for a complete file and focus our analysis for ML applications.

Our findings highlight both the potential and limitations of GPT-4o mini for file-level automated log generation in ML applications. The LLM showed moderate effectiveness in identifying where to place logs. At the same time, this high coverage comes at the cost of generating 5.15 times more log statements than human developers, which could limit its practical usefulness. In terms of evaluating logging ingredients, the LLM achieves a 59.19% exact log match. However, GPT-4o mini for file-level logging struggles with selecting relevant variables, covering only 40.58% of variables in the GT. Log texts are similar in meaning to human logs but use different vocabulary and phrasing. Our manual analysis further revealed particular issues of logging complete files, such as logging at the start or end of functions, skipping the introduction of logs in large code blocks, and the lack of adherance to project-specific logging preference.

Overall, our results indicate areas of improvement for file-level automated logging. We suggest adding relevant context related to each file in the prompt to generate better logs, as well to extract some samples related to ML pipeline to improve significant variables or messages specific to ML logs.

## 7.2 Future Work

This thesis is a first step into the generation of logging statements at the file level and, to our knowledge, the first to specifically target ML applications. Our findings open several promising paths for future research.

First, the issue of overlogging suggests the need of post-processing techniques that filter or prioritize logs based on relevance, developer intent, or potential points of failure. Automatically identifying the most relevant logs could improve the practicality of LLM-generated logs.

Second, we find that the LLM lacks awareness of variables in imported classes, as well as repository-specific logging configurations. This limitation is expected, since such information is not included in the prompt context. To ensure adoption, future work could investigate strategies for incorporating such context into the prompt. For example, the prompt may include information about relevant documentation and log configuration files.

Third, ML applications are often divided into pipeline stages to deploy a model in production. These stages include data pre-processing and model training. In-context learning techniques could be used to guide log generation based on the pipeline stage, which may improve quality of the generated logs for ML workflows.

Additionally, future work could investigate how existing baselines perform when applied to complete file logging. One approach would be to run a function-level baseline on every function within a file, enabling a direct comparison between file-level and function-level methods.

Finally, while our thesis evaluates the quality of generated logs, it does not assess their utility for developers. Future research could conduct user studies to evaluate the LLM-generated logs on different development tasks. This includes evaluating whether the inserted logs help developers detect bugs, monitor performance, or better understand system behavior.

# Bibliography

[1] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. Recommending root-cause and mitigation steps for cloud incidents using large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1737–1749. IEEE, 2023.

[2] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019.

[3] Anthropic. Introducing claude 4. https://www.anthropic.com/news/claude-4, 2025.

[4] Anders Arpteg, Björn Brinne, Luka Crnkovic-Friis, and Jan Bosch. Software engineering challenges of deep learning. In *2018 44th euromicro conference on software engineering and advanced applications (SEAA)*, pages 50–59. IEEE, 2018.

[5] Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72, 2005.

[6] Jeanderson Cândido, Jan Haesen, Maurício Aniche, and Arie van Deursen. An exploratory study of log placement recommendation in an enterprise system. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 143–154. IEEE, 2021.

[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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[8] Coreweave. Weightsbiases. https://wandb.ai/site.

[9] Shaozhi Dai, Zhongzhi Luan, Shaohan Huang, Carol Fung, He Wang, Hailong Yang, and Depei Qian. Reval: Recommend which variables to log with pretrained model and graph neural network. *IEEE Transactions on Network and Service Management*, 19(4):4045–4057, 2022.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.

[11] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. Log2: A {Cost-Aware} logging mechanism for performance diagnosis. In *2015 USENIX annual technical conference (USENIX ATC 15)*, pages 139–150, 2015.

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

[13] Zishuo Ding, Yiming Tang, Xiaoyu Cheng, Heng Li, and Weiyi Shang. Logentext-plus: Improving neural machine translation based logging texts generation with syntactic templates. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–45, 2023.

[14] Patrick Loic Foalem, Foutse Khomh, and Heng Li. Studying logging practice in machine learning-based applications. *Information and Software Technology*, 170:107450, 2024.

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

[16] Danielle Gonzalez, Thomas Zimmermann, and Nachiappan Nagappan. The state of the ml-universe: 10 years of artificial intelligence & machine learning software development on github. In *Proceedings of the 17th International conference on mining software repositories*, pages 431–442, 2020.

[17] Tobias Hey, Jan Keim, Anne Koziolek, and Walter F Tichy. Norbert: Transfer learning for requirements classification. In *2020 IEEE 28th international requirements engineering conference (RE)*, pages 169–179. IEEE, 2020.

[18] Zhouyang Jia, Shanshan Li, Xiaodong Liu, Xiangke Liao, and Yunhuai Liu. Smartlog: Place error log statement by deep understanding of log intention. In *2018 IEEE 25th international conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 61–71. IEEE, 2018.

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

[20] Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2312–2323. IEEE, 2023.

[21] VI Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Proceedings of the Soviet physics doklady*, 1966.

[22] Grace A Lewis, Ipek Ozkaya, and Xiwei Xu. Software architecture challenges for ml systems. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 634–638. IEEE, 2021.

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

[24] Heng Li, Tse-Hsun Chen, Weiyi Shang, and Ahmed E Hassan. Studying software logging using topic models. *Empirical Software Engineering*, 23:2655–2694, 2018.

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

[26] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–23, 2025.

[27] Lingwei Li, Li Yang, Huaxi Jiang, Jun Yan, Tiejian Luo, Zihan Hua, Geng Liang, and Chun Zuo. 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*, pages 1009–1021, 2022.

[28] Yichen Li, Yintong Huo, Zhihan Jiang, Renyi Zhong, Pinjia He, Yuxin Su, Lionel Briand, and Michael R Lyu. Exploring the effectiveness of llms in automated logging generation: An empirical study. *arXiv preprint arXiv:2307.05950*, 2023.

[29] Yichen Li, Yintong Huo, Renyi Zhong, Zhihan Jiang, Jinyang Liu, Junjie Huang, Jiazhen Gu, Pinjia He, and Michael R Lyu. Go static: Contextualized logging statement generation. *Proceedings of the ACM on Software Engineering*, 1(FSE):609–630, 2024.

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

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

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

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

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

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

[36] Antonio Mastropaolo, Matteo Ciniselli, Luca Pascarella, Rosalia Tufano, Emad Aghajani, and Gabriele Bavota. Towards summarizing code snippets using pre-trained transformers. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, pages 1–12, 2024.

[37] Antonio Mastropaolo, Valentina Ferrari, Luca Pascarella, and Gabriele Bavota. Log statements generation via deep learning: Widening the support provided to developers. *Journal of Systems and Software*, 210:111947, 2024.

[38] Antonio Mastropaolo, Valentina Ferrari, Luca Pascarella, and Gabriele Bavota. Log statements generation via deep learning: Widening the support provided to developers. *Journal of Systems and Software*, 210:111947, 2024.

[39] MLFlow. Mlflow. https://mlflow.org/.

[40] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22:3219–3253, 2017.

[41] Nadia Nahar, Haoran Zhang, Grace Lewis, Shurui Zhou, and Christian Kästner. A meta-summary of challenges in building products with ml components–collecting experiences from 4758+ practitioners. In *2023 IEEE/ACM 2nd International Conference on AI Engineering–Software Engineering for AI (CAIN)*, pages 171–183. IEEE, 2023.

[42] Adam Oliner, Archana Ganapathi, and Wei Xu. Advances and challenges in log analysis. *Communications of the ACM*, 55(2):55–61, 2012.

[43] OpenAI. Prompt engineering. URL https://platform.openai.com/docs/guides/prompt-engineering/tactic-ask-the-model-to-adopt-a-persona.

[44] OpenAI. Gpt-4o mini: advancing cost-efficient intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/, 2024.

[45] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

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

[47] Neoklis Polyzotis, Martin Zinkevich, Sudip Roy, Eric Breck, and Steven Whang. Data validation for machine learning. *Proceedings of machine learning and systems*, 1:334–347, 2019.

[48] Python. Logging. https://docs.python.org/3/library/logging.html.

[49] Python Software Foundation. ast - abstract syntax trees. https://docs.python.org/3/library/ast.html.

[50] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

[51] Dhia Elhaq Rzig, Foyzul Hassan, Chetan Bansal, and Nachiappan Nagappan. Characterizing the usage of ci tools in ml projects. In *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 69–79, 2022.

[52] Keita Saito, Akifumi Wachi, Koki Wataoka, and Youhei Akimoto. Verbosity bias in preference labeling by large language models. *arXiv preprint arXiv:2310.10076*, 2023.

[53] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. *Advances in neural information processing systems*, 28, 2015.

[54] Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E Hassan, Michael W Godfrey, Mohamed Nasser, and Parminder Flora. An exploratory study of the evolution of communicated information about the execution of large software systems. *Journal of Software: Evolution and Process*, 26(1):3–26, 2014.

[55] Shreya Shankar, Rolando Garcia, Joseph M Hellerstein, and Aditya G Parameswaran. " we have no idea how models will behave in production until production": How engineers operationalize machine learning. *Proceedings of the ACM on Human-Computer Interaction*, 8(CSCW1):1–34, 2024.

[56] Boyin Tan, Junjielong Xu, Zhouruixing Zhu, and Pinjia He. Al-bench: A benchmark for automatic logging. *arXiv preprint arXiv:2502.03160*, 2025.

[57] Rosalia Tufano, Ozren Dabić, Antonio Mastropaolo, Matteo Ciniselli, and Gabriele Bavota. Code review automation: strengths and weaknesses of the state of the art. *IEEE Transactions on Software Engineering*, 50(2):338–353, 2024.

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

[59] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494. IEEE, 2023.

[60] Xiaoyuan Xie, Zhipeng Cai, Songqiang Chen, and Jifeng Xuan. Fastlog: An end-to-end method to efficiently generate and insert logging statements. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 26–37, 2024.

[61] Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liqun Li, Yu Kang, Qingwei Lin, Yingnong Dang, et al. Unilog: Automatic logging via llm and in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

[62] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132, 2009.

[63] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 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*, pages 143–154, 2010.

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

[65] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *2012 34th international conference on software engineering (ICSE)*, pages 102–112. IEEE, 2012.

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

[67] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. 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*, pages 807–817, 2019.

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

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