An Empirical Study of Logging Practice in CUDA-based Deep Learning Systems

An Chen[1], Kundi Yao[1,*], Haonan Zhang[1,*], Yiming Tang[2], and Weiyi Shang[1]
[1]University of Waterloo, Waterloo, Ontraio, Canada
[2]Rochester Institute of Technology, Rochester, New York, United States
{a429chen, kundi.yao, haonan.zhang, wshang}@uwaterloo.ca[1], yxtvse@rit.edu[2]
*corresponding author

Abstract—Although logging practices have been extensively explored in conventional software systems, there remains a lack of understanding of how logging is applied in CUDA-based deep learning (DL) systems, despite their growing adoption in practice. In this paper, we conduct an empirical study to examine the characteristics and rationales of logging practices in these systems. We analyze logging statements from 33 CUDA-based open-source DL projects, covering both general-purpose logging libraries and DL-specific logging frameworks. For each type, we identify the development or execution phases in which the logs are used and investigate the reasoning behind their usage. Our quantitative analysis reveals that the majority of logging statements occur during the model training phase, with significant usage also in the model loading phase and model evaluation/validation phase. Furthermore, we observe that logging is predominantly used for monitoring purposes and tracking model-related information. Our findings not only shed light on current logging practices in CUDA-based DL development but also provide practical guidance on when to use DL-specific versus general-purpose logging, helping practitioners make more informed decisions and guiding the evolution of DL-focused logging tools to better support developer needs.

Keywords–logging practices; deep learning systems; mining software repositories

1. Introduction

Deep learning (DL) systems, with their exceptional ability to solve complex problems and process big data efficiently, are pivotal in advancing modern technology and society. These systems support major advancements across various domains, including computer vision, where they enable real-time object detection in autonomous vehicles, and natural language processing, where they support context-aware machine translation and text generation. Compared to traditional software systems, DL systems exhibit greater complexity in both software design and execution. For example, they rely on data-driven neural networks and process large volumes of unstructured data with the support of GPU/TPU acceleration. This complexity makes ensuring their quality, performance, and maintainability challenging.

Software logging is an important and widely used practice for capturing runtime information in software systems, which facilitates various critical development and maintenance activities, including assessing software quality [1, 2], spotting anomalies [3, 4], reporting errors [5], diagnosing performance issues [6], understanding system behaviors [7, 8], as well as estimating code coverage [9].

The importance of software logging in ensuring software quality has been supported by many existing studies. For example, Yuan et al. [10], Chen and Jiang [11] analyze the logging practice in open-source C/C++ and Java projects, respectively, Fu et al. [12] present the logging practice in the Microsoft industrial systems, and Zeng et al. [13] explore the logging practice in mobile applications. However, these studies focus on traditional software systems, while logging practice in DL systems still remains unclear.

The findings from existing logging studies may not be directly applicable to DL systems, particularly those involving GPU programming, where logs are typically generated on the CPU side. Due to the architectural separation between the CPU and GPU, and the asynchronous nature of GPU execution, traditional logging practices face limitations in capturing runtime behavior within GPU kernels. In DL systems, CUDA has emerged as the dominant platform for GPU programming. CUDA, developed by NVIDIA, provides a parallel computing platform and programming model that helps developers accelerate training and inference tasks by harnessing the power of GPU accelerators [14]. It is widely adopted in popular frameworks such as TensorFlow, PyTorch, MXNet, etc. By studying CUDA-based DL systems, this study aims to bridge the current research gap and provide a deeper understanding of logging practices in GPU-accelerated DL systems.

Logging plays an even more important role in projects that leverage DL techniques due to the unique characteristics of these projects. The DL model often functions as a black box with complex internal states unobservable, making their behavior difficult to understand and debug without proper observability [15]. Therefore, logging becomes an important practice when using DL models as the log messages generated often serve as the only source of information about the model. Logging statements in these projects can help capture crucial information about model training processes (e.g., loss values, learning rates), inference behaviors (e.g., prediction distributions), and resource utilization (e.g., GPU memory consumption). This information is vital for reproducing experiments, tracking model convergence, and ensuring model reliability in production environments.

To study logging practice in CUDA-based DL systems, we systematically collect a set of open-source projects that use CUDA-related DL libraries and study the logging practice

2693-9177/25/$31.00 ©2025 IEEE
DOI 10.1109/QRS65678.2025.00027

in these projects, hoping to shed light on the research about logging in deep learning-based projects. Specifically, our study investigates the usage of both general-purpose and DL-specific logging libraries in the studied DL systems. Our findings indicate that the majority of the studied DL projects leverage both types of logging, highlighting a complementary relationship rather than a substitutional one. Furthermore, our analysis reveals that within the DL pipeline, the phases of model training and model loading are the most frequently logged, underscoring their importance in model lifecycle management. We also observe that the majority of logging content focuses on monitoring and model-related information, with such an observation consistently present in both logging types. These insights suggest that current DL logging tools should be optimized to address the specific needs of DL developers, particularly in reducing the fragmentation caused by routing logs to separate destinations through different frameworks. By aligning logging capabilities more closely with common usage scenarios, future DL logging frameworks can better support transparency, traceability, and debugging in DL workflows.
The contributions of this paper are as follows:

- To the best of our knowledge, this is the first study that analyzes the logging practice in CUDA-based deep learning systems.
- We report five findings concerning the characteristics of both general-purpose and DL-specific logging, as well as their interrelationships, providing valuable insights for future research on logging in CUDA-based deep learning systems.
- To facilitate the reproducibility of our work, we have made our dataset and code publicly available [1].

Paper organization. The paper is structured as follows: Section 2 introduces the background in Deep Learning pipelines, general and DL-specific logging libraries. Section 3 describes the subjects we have studied and how we extract the related data. Section 4 explains the motivation, approach, and results of each research question. Section 5 presents the threats to the validity of the study. Section 6 discusses the related work. Finally, Section 7 concludes this paper.

2. Background

In this section, we will introduce the general architecture of a DL system: the DL pipelines, and compare DL-specific logging libraries with general-purpose logging libraries.

2.1 DL Pipelines

DL pipelines are a series of steps that are used to build, train, and deploy deep neural network models.
They enable the automation of DL workflows, making it easier for data scientists and engineers to work with large datasets and complex algorithms. DL pipelines typically consist of several stages. Each stage of the pipeline is designed to perform a specific task, and the output of one stage is often used as the input for the next stage. This allows for a seamless flow of data and information throughout the entire pipeline.

[1]https://github.com/A1ex-Chen/DL_project_log

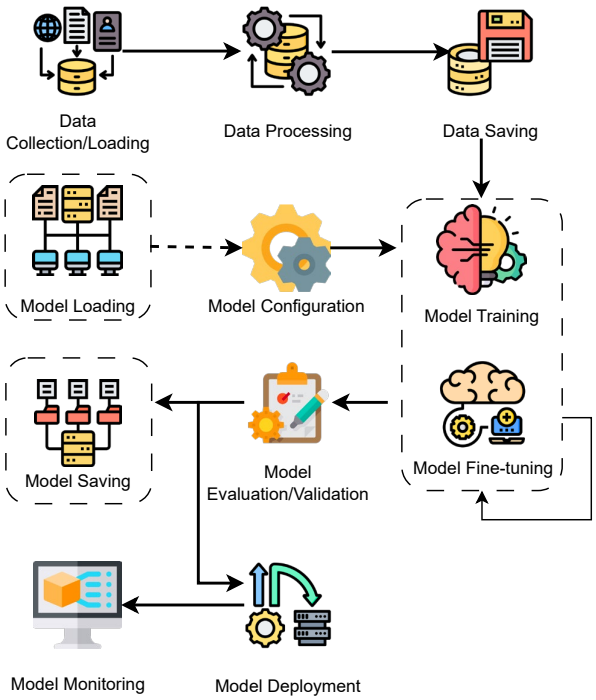


Figure 1: Overview of DL pipeline

Figure 1 illustrates a general DL pipeline along with its key stages, according to prior research [16, 17]. The pipeline begins with *data collection* and *preprocessing*, followed by *model development* where the architecture is defined and hyperparameters are tuned. It then proceeds to *model training*, where the model learns from the training data, and *validation*, where performance is evaluated on unseen data. After the model reaches the desired performance, it is deployed to production environments. Throughout this process, monitoring and logging are essential to track the model's performance, debug issues, and ensure the system operates as intended. Each stage has distinct logging needs, from tracking data distributions in preprocessing to monitoring inference latency in deployment.

2.2 General and DL specific logging libraries

In the context of DL systems, logging libraries can be categorized into two types:

- **General-purpose logging libraries:** General-purpose logging libraries, such as Python's built-in `logging` module [18], provide basic functionalities for recording system runtime information. These libraries offer essential features for logging, such as log levels for filtering logs, customizable formatting, and flexible output destinations (console, files, or remote services). For instance, the logging statement `logging.info("Model training started")` records the initiation of deep learning model training, with the log message directed to a predefined

output destination (e.g., console or file) in plain text format, as specified by the configuration of the logging framework. However, in the context of DL systems, general-purpose logging lacks built-in support for tracking key artifacts such as model weights, gradients, loss curves, or evaluation metrics over time. As a result, while such libraries remain useful for operational and debugging purposes, they often fall short in supporting model development and experimentation workflows.

- **DL-specific logging libraries:** To address the specialized needs of DL practitioners, DL-specific tools such as *Tensorboard* and *Weights & Biases (WandB)* have emerged as a DL-specific logging solution. These tools are designed for the DL development lifecycle, focusing on experiment tracking, model performance, and visualizing experiments. For instance, Figure 2 is used to log an image for visualization in distributed training and record training information. Figure 3 shows the visualization produced by this logging statement. Unlike general logging which primarily records events after they happen, DL logging is actively incorporated into the experimentation workflow.

```
import torch.distributed as dist
from torch.utils.tensorboard import SummaryWriter
# Initialize distributed training
dist.init_process_group(backend="nccl", init_method="env://")
local_rank = torch.distributed.get_rank()
torch.cuda.set_device(local_rank)
writer = SummaryWriter()
for epoch in range(5):
    train_loss = 0.02 * (10 - epoch)
    writer.add_image("Loss", train_loss, epoch)
    # Log an image
    writer.add_image("MNIST Digits", figure, epoch)
writer.close()
```

Figure 2: Example of DL-specific logging

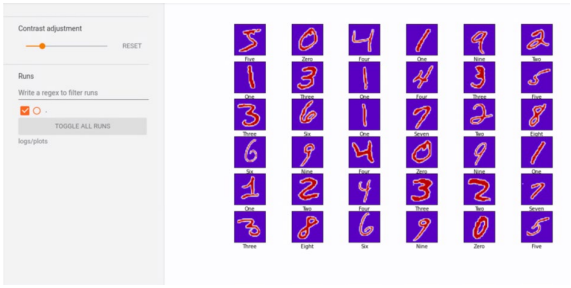


Figure 3: Example of visualization capabilities

3. Case Study Setup

In this section, we describe how we collect the subjects under study and extract data from them for quantitative and qualitative analysis [19].

a) Identifying CUDA-related Libraries

To identify CUDA-related libraries commonly used in DL frameworks, we conduct a gray and white literature review using Google search and Google Scholar to build a comprehensive list of documented CUDA-related libraries. This approach allows us to identify both standard NVIDIA CUDA libraries and other specialized CUDA-related libraries that leverage GPU acceleration. We summarize in total five CUDA-related libraries that are widely used in DL projects: *PyTorch*[2], *TensorFlow*[3], *CUDA*[4], *Numba*[5], and *PyCUDA*[6].

b) Extracting GitHub projects using CUDA-related Libraries

We collect the study subjects from Github, the largest open-source software platform to date. To find repositories that utilize CUDA-related libraries, we leverage the GitHub Search API [7] to search for projects that import or use our identified CUDA-related libraries. Following the prior work [16], our approach includes these steps: (1) We search repositories that import any of the five CUDA-related libraries in Python files, as these libraries are predominantly used within Python-based DL frameworks. For instance, we search for keywords like "`import torch`" and "`from torch`", which are designed to capture the usage of PyTorch in Python codebases. (2) We remove duplicates from our results to ensure that each repository is counted only once. (3) We adopt a similar approach to previousous studies [16, 20] select only projects with logging implementations using keywords like "`log`", "`logging`", and "`logger`". This methodology yields 392 unique repositories that use CUDA-based DL infrastructure and contain logging implementations in their codebase.

c) Projects sampling

From our initial collection of 392 repositories, we conduct a random sampling process to select a subset of 33 projects for in-depth analysis. Random sampling is employed to ensure that our selected projects represent the broader population of CUDA-related repositories without selection bias. The random sampling was implemented using Python's `random.sample` function, which provides an unbiased selection from the larger population. We opt for this sample size to balance comprehensive analysis with the practical constraints of manual inspection. A smaller, randomly selected subset allows us to conduct a more thorough qualitative analysis of each project's logging practices, code structure, and documentation, which would be infeasible across the entire corpus of 392 projects. The resulting dataset retains the diversity of the original collection in terms of project size, domain, and popularity metrics.

d) Extracting logging libraries

As this study focuses on Python projects, we require logging libraries that are specific to Python, with particular emphasis on those that are widely adopted and have a significant presence in the Python ecosystem. Following a similar approach to identifying CUDA-related libraries, we conduct a gray and white literature review to compile a list of logging libraries used in Python projects. As a result, our library list includes both general-purpose logging libraries and DL-specific logging

[2]https://pytorch.org/
[3]https://www.tensorflow.org/
[4]https://developer.nvidia.com/cuda-toolkit
[5]https://numba.pydata.org/
[6]https://pypi.org/project/pycuda/
[7]https://api.github.com/

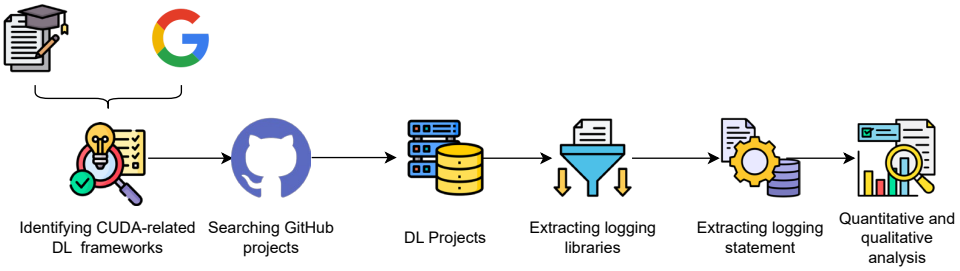


Figure 4: Overview of our research workflow.

libraries, which differ significantly in their functionality and purpose.

Understanding the distinction in logging libraries is crucial for our study. Treating them as equivalent would be inappropriate, as they serve fundamentally different purposes and use cases.

4. Case Study Result

In this section, we present the study results of the research questions.

4.1 *RQ1:* What are the characteristics of logging practice in deep learning systems?

4.1.1 Motivation

Logging practices have been extensively studied in traditional software systems [10, 11, 13], but their application in DL systems remains underexplored. Although there are studies [16] on logging practices in modern software systems, they focus on ML systems and lack exploration of DL systems. DL systems present distinct characteristics that differ significantly from traditional software systems, such as high computational complexity, non-deterministic behavior, and resource-intensive operations, making effective logging practices essential for debugging, monitoring, and performance optimization. By understanding the characteristics of logging practices in DL systems, we aim to uncover how developers address logging-related challenges and identify areas for improvement in logging practices. The results are expected to contribute to enhancing the quality, reliability, and maintainability of DL software projects.

4.1.2 Approach

Following a similar approach to the prior research [16], we study the characteristics of the logging practices in the DL systems from the following aspects:

- **Project-level characteristics:** According to the prior research approaches [16], we first identify the project-level characteristics of logging practices in DL systems using four metrics: (1) *number of stars*, (2) *number of commits*, (3) *number of contributors*, and (4) *source lines of code (SLOC)*.
- **Library-level characteristics:** We then study the library-level characteristics of logging practices in DL systems, with a focus on analyzing the use of DL-specific and general-purpose logging libraries. The process of extracting logging libraries from DL systems involves both automated and manual approaches. We began with an automated approach using Python's Abstract Syntax Tree (AST) to parse each code file and extract `import` statements. These import statements were then compared against our curated list of known logging libraries to identify logging framework usage across projects. For ambiguous cases or custom logging implementations, we conducted a manual analysis to ensure accurate identification.
- **Density of logging statements:** We measure the density of logging statements to reflect the popularity of logging in DL systems. Projects with higher density often exhibit a stronger emphasis on debugging and monitoring, which is critical for DL systems due to their complexity and non-deterministic behavior. The density is calculated as the ratio of the number of logging statements (N_{logs}) in the DL project to the total source lines of code (N_{SLOC}), expressed as $\frac{N_{\text{SLOC}}}{N_{\text{logs}}}$. This metric provides a quantitative measure of the emphasis on logging within a DL project. For each studied project, we use the *scc*[8] tool to calculate the total source lines of code in all Python files. We then count the number of logging statements by identifying calls to both general and DL-specific logging methods. Our logging statement extraction is based on the logging libraries identified in the previous aspect and is conducted using a Python AST parser as well.

4.1.3 Results

In this section, we present our study results based on the three aspects mentioned above.

Project-level characteristics. Figure 5 provides an overview of the project-level characteristics of logging practices, which shows the diversity of our project selection. In this figure, the distributions of stars, contributors, and commits show similar trends, with the majority of DL projects clustered at the lower end of each metric. This suggests that many DL systems are relatively new or less mature in terms of community engagement and development history. In contrast, SLOC differs evidently, with many projects having higher SLOC values, which indicates that DL projects can be complex in their implementation. Such complexity renders logging practices critical for understanding the behavior of DL systems.

[8]https://github.com/boyter/scc

TABLE I: Summary of logging libraries used in studied DL systems.

Type	Library	Stars	Contrib.	Releases	Main Purpose	Import keywords
DL-specific logging	MLflow	14,076	573	65	Experiment tracking and management	*mlflow*
	Wandb	5,800	127	113	Visualization and experiment tracking	*wandb*
	TensorBoard	6,195	224	49	Training visualization and metrics monitoring	*torch.utils.tensorboard*
	PyTorch Lightning	23,819	801	107	Training organization and logging	*lightning.pytorch*
	Comet_ml	2,114	17	42	Experiment and model management	*comet_ml*
	ClearML	4,388	112	54	ML workflow automation and tracking	*clearml*
	DLLogger	251	18	5	Performance tracking for deep learning	*dllogger*
	Accelerate	5,745	228	37	Training optimization and monitoring	*accelerate.logging*
	Neural_compressor	1,412	83	19	Model optimization and benchmarking	*neural_compressor.utils.logger*
	ColossalAI	34,583	309	29	Distributed training management	*colossalai.logging*
General logging	Logging	51,877	2,234	540	Standard Python logging facilities	*logging*
	Structlog	3,473	133	43	Structured logging for Python	*structlog*
	Loguru	16,113	78	30	Simplified logging with enhanced features	*loguru*
	Sentry_sdk	4,578	291	453	Error tracking and performance monitoring	*sentry_sdk*

Notes: Some libraries like PyTorch, TensorFlow, and Transformers contain logging functionality but are primarily DL frameworks rather than dedicated logging libraries. These libraries are not presented in this table to avoid ambiguity while logging practices using these libraries are included in this study.

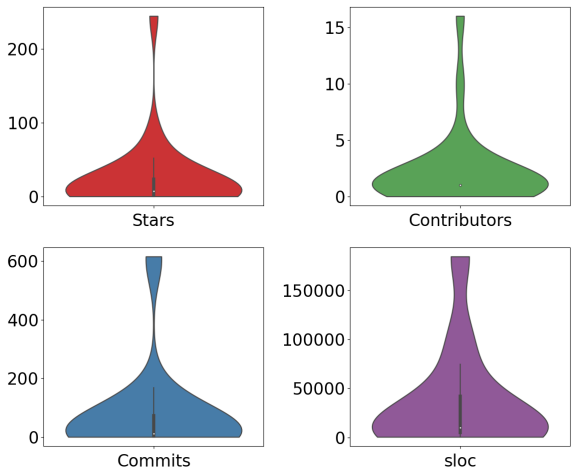
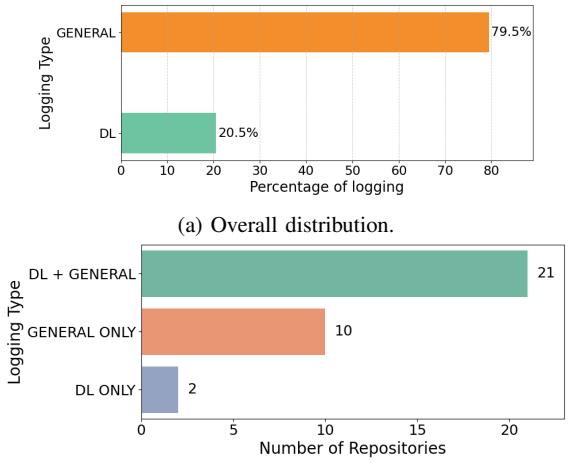


Figure 5: Distribution of selected deep learning projects across metrics



(a) Overall distribution.

(b) Distribution by repositories.

Figure 6: Distribution of logging library types.

Library-level characteristics. Table I presents an overview of the logging libraries involved in this study. It can be observed that DL systems include many DL-specific logging libraries that have been overlooked by prior logging studies, which highlights the necessity of this study.

We then analyzed the distribution of logging library types across the selected projects. Figure 6a illustrates the prevalence of different types of logging libraries. We can observe that general logging libraries are far more widely used as logging solutions in DL-based systems compared to DL-specific logging libraries. This result reflects that while DL-specific logging statements are valuable for visualizing training metrics and tracking experiments, general-purpose logging remains a fundamental aspect of DL development. The reliance on general logging indicates that developers still prioritize traditional software engineering practices, even in the context of complex DL systems.

As shown in Figure 6b, we can observe that most of the studied projects adopt both general-purpose and DL-specific logging libraries, with a few exceptions that utilize only one type of logging libraries. The prevalence of projects using both types of logging libraries (representing 64% of our sample) indicates a recognition of the unique monitoring needs in DL systems that cannot be fully addressed by general-purpose logging alone. The high adoption rate of both general and specialized logging libraries suggests that DL project developers recognize the need for different types of logging libraries to address various aspects of system observability. General logging handles traditional software concerns like error reporting and execution

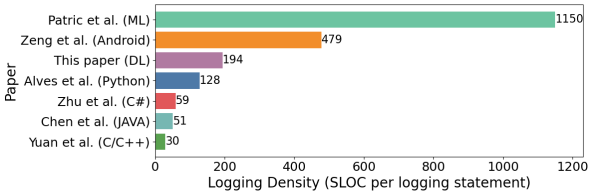


Figure 7: Distribution of logging density across different systems.

flow tracking, while DL-specific logging addresses unique DL requirements such as hyperparameter tracking, gradient monitoring, and visual representation of training dynamics.

Finding 1

DL projects primarily adopt a mix of general and DL-specific logging libraries, with general logging libraries being the dominant logging solution. This reflects the complexity of logging practice in DL projects.

Density of logging statement. For a more straightforward comparison with other system types, we incorporate our results into those reported in prior studies [10, 11, 21, 22, 13, 16]. In Figure 7, "The paper (DL)" is the result of this study, while the remaining items are from the existing paper. Among the 33 selected DL projects, we identified 7,561 logging statements and 1,466,769 source lines of code, resulting in a logging density of 194. The comparison reveals that modern software systems tend to have lower logging frequency compared to traditional software systems, with higher values in the figure indicating fewer logging statements per source line of code. DL systems rank among the top three with the sparsest logging, suggesting that current DL systems could be improved through denser and more deliberate logging practices.

Finding 2

DL systems exhibit sparse logging, significantly lower than that of traditional software systems, which highlights the need for denser logging practices in DL systems.

RQ1 Summary

DL projects are complex, often adopting a mix of general-purpose and DL-specific logging libraries, but they tend to have sparser logging compared to traditional software systems. This highlights the challenges of studying logging practices in DL systems, particularly in terms of system complexity, the diversity of logging libraries involved, and the need for denser logging practices.

4.2 *RQ2:* Which phases of the DL pipeline incur more logging?

4.2.1 Motivation

Understanding the phases of the DL model lifecycle where logging is most prevalent is crucial for improving development practices. While traditional software logging studies examine where [21, 12] and what to log [23, 24], the unique characteristics of DL systems—such as complex training pipelines, hyperparameter tuning, and model evaluation processes—require specialized logging considerations. Identifying which phases of the DL lifecycle (e.g., data preprocessing, model training, evaluation) are most frequently logged and understanding developers' intentions can provide valuable insights for practitioners. This knowledge can help establish effective logging patterns specific to DL systems, enabling better model monitoring, debugging, and reproducibility of experiments, which are essential challenges in DL system development.

4.2.2 Approach

To answer this RQ, we need to associate logging statements with the phases of the DL pipeline. This requires manual analysis to interpret the context of each logging statement and identify which phases of the DL pipeline it originates from. The manual analysis consists of two steps:

Logging statements sampling:
Due to the large number of logging statements across the selected projects, we apply random sampling, as adopted by a prior study [25], to narrow down the dataset for feasible manual analysis. The sampling is based on 95% confidence level with a 5% confidence interval. Specifically, we apply sampling to general and DL-specific logging separately, as these two types of logging serve different purposes and exhibit distinct characteristics. As a result, we obtained 265 and 203 logging statements from a total of 845 and 428 general and DL-specific logging statements, respectively.

Manual investigation: Our manual investigation is inspired by previous research [16, 17]. The first two authors of this paper first independently examined each of the sampled logging statements to identify the corresponding DL model phase. The DL pipeline phases involved in this study were not created from scratch. Instead, we carefully reviewed existing research to construct an initial version of the DL pipeline, which we then refined based on the characteristics of the specific DL systems analyzed in this study. For each logging statement, we analyzed its context by reviewing the surrounding code, method definitions, invocations in the call graph, comments, and documentation to determine which phase of the DL pipeline incurs the logging statement. To establish a reliable and consistent categorization scheme, we first conducted a preliminary labeling round on the randomly selected 100 logging statements. This initial round helped us obtain consensus on the categorization approach and develop a shared understanding of the model phases and the phases' identification rationale. After this preliminary analysis, we built a categorization scheme that received approval from two

raters, which could serve as a "codebook" for the full analysis. For the second round of labeling, we applied the established categorization scheme to analyze all remaining log samples. This round included both labeling the previously unexamined logging statements and revisiting the initial 100 statements to ensure consistency with our established categorization scheme. After independent reviewing, the two raters conducted reconciliation meetings to resolve any disagreements and achieve consensus on the final categorization. For model phase identification, we calculated *Cohen's kappa* coefficient [26], which measures agreement between two raters while accounting for chance agreement. This statistical metric helped validate the consistency and objectivity of our categorization approach.

4.2.3 Results

Based on our manual analysis of the sampled logging statements, we identified and categorized the DL pipeline phases incurring logging. The *Cohen's kappa* coefficients for DL pipeline phase categorization are 0.93 for DL-specific logging and 0.89 for general logging, indicating a high level of agreement between raters. In general, the most frequent phase in DL systems that incurs logging is Model Training, accounting for 37.2% of all the logged statements we analyzed. This is followed by Model Loading (14.8%) and Model Evaluation/Validation (14.3%). Model Configuration incurs 9.3% of logging statements, while Data Processing accounts for 7.5%. Other phases incurring a nontrivial amount of logging include Model Saving (6.4%), and Data Collection/Loading (2.9%). 4.6% of logging statements were left uncertain due to insufficient information in their log messages, surrounding context and documentation.

Finding 3

Logging in DL systems is predominantly concentrated in the model training phase (37.2%), followed by model loading (14.8%) and evaluation/validation (14.3%), reflecting the critical importance of monitoring these key processes in the DL lifecycle.

Table II presents the detailed distribution of logging statements across different phases of the DL pipeline. According to the table, general and DL-specific logging libraries are introduced during different phases of the DL development pipeline. DL-specific logging libraries are more likely to be introduced during model development phases, such as training and validation, while general logging libraries are more likely to be introduced during setup-related phases, including model configuration and data processing.

The concentration of logging in the training phase aligns with the computationally intensive and time-consuming nature of DL model training, where developers need visibility into progress and performance metrics. The substantial focus on model loading and evaluation phases highlights their critical importance in the DL workflow, particularly for ensuring model correctness and tracking performance. However, Model Monitoring accounts for only 0.7% of logging statements, suggesting a potential gap in current logging practices. As DL models increasingly move to production environments, more robust logging practices for ongoing monitoring may become necessary to ensure the DL pipeline's reliability.

RQ2 Summary

In general, the model training phase (37.2%) is the dominant phase of the DL pipeline for incurring logging, followed by model loading (14.8%) and evaluation/validation (14.3%). DL-specific logging libraries are more likely to be introduced during model development phases, while general logging libraries are more likely to be introduced during setup-related phases. These results indicate that general and DL-specific logging libraries serve different logging purposes in DL systems and complement each other, providing useful runtime information to developers, especially regarding the DL models.

4.3 *RQ3:* What rationales are behind DL system logging practice?

4.3.1 Motivation

In RQ2, we studied logging practice from the perspective of DL pipelines without considering the developers' intention. This RQ aims to answer what specific purposes these logging statements serve. By examining developers' motivations for logging, we can develop a comprehensive understanding of logging practices in DL systems that not only identifies the corresponding DL pipeline phases that incur logging but also why logging occurs at those specific points in DL system development.

4.3.2 Approach

To understand the rationales behind logging in DL systems, we conducted a qualitative analysis of the logging statements collected in RQ2. Our approach involved systematically examining each logging statement along with its contextual information to identify the underlying motivation for its inclusion in the source code. Similar to the approach used in RQ2, we manually analyzed each statement; however, unlike the DL pipeline phase identification, a single logging statement could be associated with multiple rationales. For example, the logging statement `logging.info(f'test time: test_elapseds | test loss test_loss')` records both as model monitoring metrics ($test\_loss$) and time information ($test\_elapsed$).

Since rationale classification allows for multiple labels per instance, we used *Krippendorff's alpha* [27]—a statistical measure of inter-rater reliability that accommodates multi-label data—to evaluate the level of agreement between raters.

4.3.3 Results

Based on our manual analysis, we categorized the rationales behind logging statements in DL systems. We achieve

TABLE II: Distribution of logging statements in the different phases of the DL pipeline.

DL pipeline	Description	Example	% overall	% general	% DL
Model Training	The chosen models are trained and tuned on the collected data and labels.	`logger.info(f'Continuing training from global step self.state.global_step')`	37.2%	25.4%	50.2%
Model Loading	Loading a pre-trained model.	`logger.info(f'Loading model_name checkpoint from: pretrained')`	14.8%	21.7%	7.0%
Model Evaluation/Validation	Engineers evaluate the output model on tested datasets using pre-defined metrics.	`logger.info(f'>> Validation: step loss: meter.loss_meter.avg:.4')`	14.3%	10.4%	18.6%
Model Configuration	Configuring or Initializing a model.	`logger.info(f'Initialize PyTorch weight key')`	9.3%	11.7%	6.55%
Data Processing	Preprocessing data for model training or evaluation.	`logging.info('Using Dataset Sharding')`	7.5%	10.8%	4.2%
Model Saving	Saving a trained model.	`self.logger.info('Saving checkpoint: ...' .format(filename))`	6.4%	5.0%	7.9%
Uncertain	The model phase cannot be determined.	`logging.info(f'TIMER name elapsed')`	4.6%	6.3%	2.8%
Data Collection/Loading	Data collection and processing.	`logger.info('loading archive file archive_file')`	2.9%	4.2%	1.4%
Model Deployment	The inference code of the model is deployed on the targeted device(s).	`logging.info(f'Running with single process. Device args.device.')`	1.5%	2.5%	0.5%
Model Monitoring	The deployed model is continuously monitored for errors during execution.	`logging.debug('Starting wandb.')`	0.7%	0.8%	0.5%
Model Fine-tuning	Fine-tuning a pre-trained model.	`logger.info('Next fine-tuning the entire model...')`	0.4%	0.8%	0.0%
Data Saving	Saving processed data.	`logger.info('Saved pseudo label data to pselab_path')`	0.4%	0.4%	0.5%

Krippendorff's alpha of 0.94 for DL-specific logging and 0.88 for general logging, indicating strong agreement between both raters. Table III illustrates the details and hierarchical distribution of these rationales.

Our analysis reveals that monitoring and model information are the primary rationales behind the studied logging practice that account for over 40% of all logging rationales in DL systems. This reflects the computationally intensive and iterative nature of DL system development, where monitoring long-running processes and tracking configuration parameters are crucial for effective development and reproducibility.

The significant presence of "validation" (14.8%) highlights developers' concerns about ensuring the correctness of models, datasets, and configurations throughout the development process. Meanwhile, tracing logs (12.8%) help developers follow execution flows in complex DL pipelines, and checkpoint-related logging (9.1%) supports the critical practice of saving and restoring model states during lengthy training processes.

This distribution of logging rationales demonstrates how DL systems require specialized logging approaches that address the unique challenges of model training, evaluation, and deployment. The focus on monitoring metrics and model information particularly distinguishes DL logging from traditional software logging practices, which typically emphasize error reporting and execution flow.

Finding 4

Monitoring (22.4%) and model information (18.1%) are the primary rationales for logging in DL systems, reflecting the need to track both training progress and model configuration.

4.3.4 Discussion

Figure 8 shows the distribution of general and DL-specific logging rationales. In general-purpose logging within DL systems, we observe a notably higher proportion of validation (23.7% vs 8.1%) and tracing (21.1% vs 12.3%) compared to DL-specific logging statements. This difference highlights that general logging functions still primarily serve traditional software engineering needs - validating configurations, dependencies, and data integrity, as well as tracing execution flow. These aspects are less directly tied to DL algorithms themselves and more focused on the surrounding program infrastructure.

Conversely, DL-specific logging shows significantly higher proportions in categories directly related to model operations, such as monitoring (30.5% vs 11.6%). This distribution suggests that DL-specific logging focuses more on the unique aspects of DL processes like model training dynamics and performance evaluation.

Based on the analysis, we can conclude that DL systems exhibit distinct logging rationales that reflect their unique

TABLE III: Categorization of logging rationales in DL systems.

General Rationale	Detailed Rationale	Rationale Explanation	%
Monitoring (22.4%)	Model Monitoring Metrics	Tracking training progress and model behavior (epoch number, learning rate, loss value)	13.0%
	Model Performance Metrics	Direct measures of model performance (accuracy, precision, recall, F1 score)	6.9%
	Resource Utilization	Hardware utilization tracking (CPU, GPU, memory consumption, VRAM usage)	1.5%
	Model Validation Metrics	Other metadata related to model training (throughput, latency)	1.11%
Model Information (18.1%)	Model Configuration	Non-trainable model parameters and configuration settings	9.3%
	Model Information	Basic model metadata (model name, selection)	4.8%
	Model Parameters	Trainable parameters that the model learns from training data	3.3%
	Model Architecture	Details of layers, activation functions, and connections	0.7%
Tracing (16.1%)	Logging Steps	Recording next action or step in execution flow	4.1%
	General Information	Descriptive natural language information about execution	3.7%
	Time Information	Elapsed time for model training (time taken per epoch, per batch, or total time).	3.3%
	Data Processing	Logging during data processing operations	3.0%
	General Configuration	Configuration details not directly related to model	2.0%
Validation (14.8%)	Configuration Validation	Validation of configuration settings and parameters	7.6%
	Data Validation	Validation of dataset properties and characteristics	3.0%
	Model Validation	Validation of model components (weights, encoders, etc.)	2.8%
	Dependency Validation	Validation of required libraries and dependencies	1.5%
Model & Data I/O (9.1%)	Model Loading	Loading model components (model, tokenizers, weights)	5.0%
	Model Saving	Saving model components (model, tokenizers, weights)	3.0%
	Dataset Saving	Saving raw or preprocessed datasets	0.7%
	Dataset Loading	Loading raw or preprocessed datasets	0.4%
Others (6.1%)	Cross-platform transition	Transition b/w Tensorflow and Pytorch, or other pairs of platforms	2.0%
	Debug	the log aims to assist in debugging a specific issue	2.0%
	Formatting	Special characters or (and) words to separate output content	1.3%
	Job submission	Submitting execution of computational tasks	0.7%
Visualization (5.7%)	Metadata logging	Log metadata for visualization, such as images, videos, etc.	5.7%

challenges and requirements. The emphasis on monitoring and model information in DL-specific logging underscores the need for specialized logging practices that cater to the complexities of deep learning development.

Finding 5

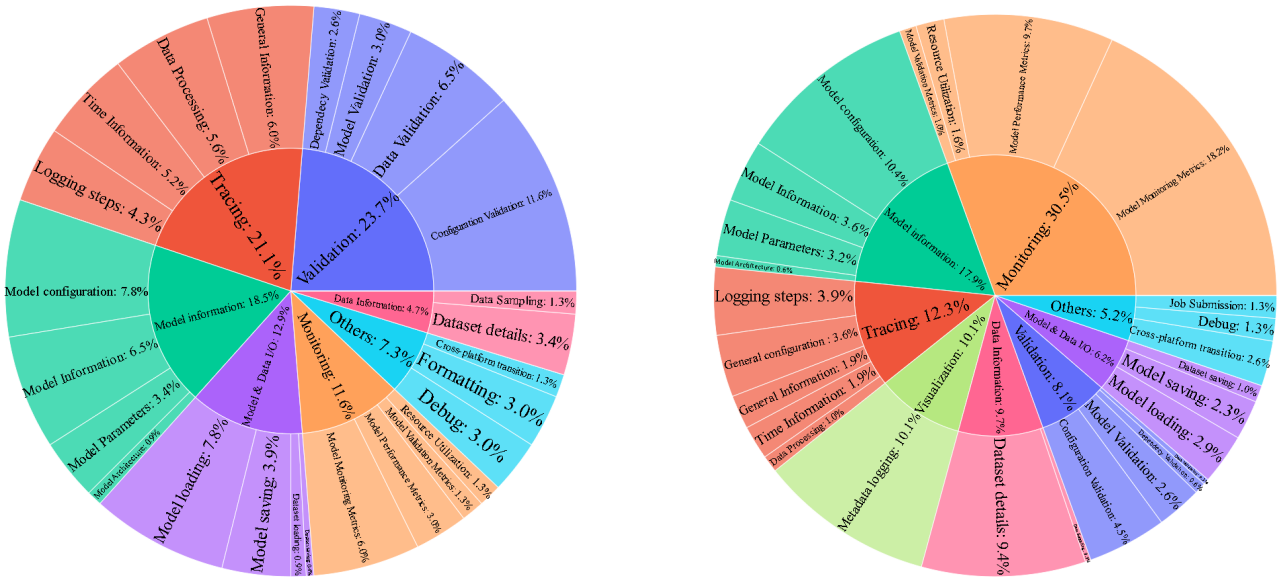
The rationales for logging in DL systems are more diverse than those in traditional software systems. General-purpose logging focuses on validation (23.7%) and tracing (21.1%), while DL-specific logging emphasizes monitoring (30.5%). This indicates that DL systems require specialized logging practices to address the unique challenges of model training and evaluation.

RQ3 Summary

The primary rationales for logging in DL systems are monitoring (22.4%) and model information (18.1%), with validation (14.8%), tracing (12.8%), and checkpoint management (9.1%) also playing significant roles. This distribution reflects the unique challenges of DL system development, where tracking training progress, ensuring reproducibility through configuration tracking, and validating complex models and datasets are essential for successful outcomes.
General-purpose logging within DL systems continues to serve traditional software engineering needs, with higher emphasis on validation (23.7%) and tracing (21.1%). DL-specific logging demonstrates a stronger focus on monitoring (30.5% vs 11.6% in general logging) and other aspects directly related to DL processes. These differences highlight the dual nature of logging in DL systems: maintaining traditional software engineering practices while incorporating specialized logging techniques to address the unique challenges of DL systems.

5. Threats to validity

We present the threats to the validity of our research.
Internal validity We conduct manual investigations of the model phases and rationales of logging practices. Despite our rigorous approach with two independent reviewers and reconciliation discussions, the categorization process may still be influenced by researchers' subjective judgments which may not fully align with developers' intentions. Additionally, although we carefully examined the contextual information of the logging statement, we may not have fully captured the



(a) Percentage of general logging rationales

(b) Percentage of DL-specific logging rationales

Figure 8: Distribution of general and DL-specific logging rationales

complete execution flow or domain-specific knowledge that motivated certain logging decisions based on a static code review.

External validity We conduct a qualitative study on a sampled subset of logging statements. While it is impractical to cover all logging usage and scenarios due to the sheer volume, our findings broadly capture logging practices and rationales in deep learning system development, offering insights that can help practitioners build more robust logging infrastructures.

Construct validity Our study focuses on logging practices in CUDA-based deep learning projects. Other DL frameworks, such as ROCm, may exhibit different logging behaviors that are outside the scope of this work. Additionally, our study of logging usage and deep learning systems focuses primarily on Python-based open-source projects, as Python is the most commonly used language for deep learning tasks. Different programming languages or proprietary deep learning systems may exhibit different logging practices, and the generalizability of our findings may be limited by both the language choice and the distribution of application domains in our dataset.

6. Related Work

In this section, we introduce the related work.

6.1 Empirical study on software logging

Many studies have been conducted to understand logging practices. For example, Yuan et al. [10] analyze the logging practice in some open-source C/C++ projects. Shang et al. [2] characterize the logging statements in two Java projects to uncover the relationship between the logging practices and the code quality. Fu et al. [12] uncover developers' logging practices in two large industrial systems. Chen and Jiang [11] perform a replication study in many Java applications and compare the logging practice in Java with that in C/C++ [10]. Zeng et al. [13] study the logging practice in mobile application and compare it with that in server and desktop applications. He et al. [28] study the topics in the natural language description in the logging statements. Kabinna et al. [29] study the stability of the logging statements and the log files. Li et al. [8] perform a qualitative study to understand developers' opinions about software logging. Tang et al. [30] study the logging practice related to the logging levels. More recently, Zhang et al. [31] study the logging practice in test code and Foalem et al. [16] study the logging practice in machine learning-based applications. Despite the high volume of studies performed regarding the logging practice, they are primarily concerned with the general logging practice without considering domain-specific characteristics, such as those specific to DL systems. Although Foalem et al. [16] provide an initial investigation into logging practices in DL applications, their analysis is limited to projects that utilize DL-specific logging libraries and does not consider DL projects that rely solely on general-purpose logging frameworks. To address this gap, our study investigates logging practices in DL systems by examining both general-purpose and DL-specific logging across projects that utilize CUDA-related libraries. Our findings reveal logging behaviors specific to DL systems and provide a foundation for designing more effective and targeted logging solutions.

6.2 Improving logging practice

The studies for improving the logging practice are primarily about where to log and what to log [31]. Research about where to log mainly focuses on where to place the logging statements. Zhu et al. [21] introduce a tool to help developers

make informed decisions about where to place the logging statements. Ding et al. [32] and Zhao et al. [33] propose a logging framework respectively that take the overhead and effectiveness of logging into consideration. Yao et al. [34, 35] present an automated logging tool that can suggest the logging locations for monitoring the software performance. Moreover, Li et al. [36] suggest where to log by using a DL model. Studies about what to log are mostly concerned with the static and dynamic information in the logging content and logging levels. Shang et al. [23] study what knowledge to incorporate into software logging. Liu et al. [24] propose a tool that helps developers to choose which variable to log. Ding et al. [37] introduce a tool that can automatically generate logging content based on the context information. Li et al. [38] present a tool to help developers decide which logging level to choose. Unlike prior studies that target general-purpose software systems, our research investigates logging practices and scenarios specific to DL systems.

7. Conclusion

In this paper, we provide an in-depth empirical analysis of logging practice in CUDA-based deep learning projects. By examining logging statements across 33 open-source projects, we find that the majority of logging activity occurs during the model training phase, with substantial logging also observed during the phases of model loading, model evaluation, and model validation. We also find that general logging and DL-specific logging often complement each other. These findings reveal the current logging practices in CUDA-based deep learning projects and can provide insights to developers to make informed decisions when logging with general-purpose logging libraries and DL-specific logging libraries. Future work can further investigate how to use general logging and DL-specific logging to assist software development and maintenance.

References

[1] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, 1999.

[2] Weiyi Shang, Meiyappan Nagappan, and Ahmed E. Hassan. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, 20(1):1–27, 2015.

[3] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 Ninth IEEE International Conference on Data Mining*, pages 149–158, 2009.

[4] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. Mining invariants from console logs for system problem detection. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 24, USA, 2010. USENIX Association.

[5] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 103–116, New York, NY, USA, 2009. Association for Computing Machinery.

[6] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 26, USA, 2012. USENIX Association.

[7] Qiang Fu, Jian-Guang Lou, Qingwei Lin, Rui Ding, Dongmei Zhang, and Tao Xie. Contextual analysis of program logs for understanding system behaviors. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, page 397–400. IEEE Press, 2013.

[8] 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*, pages 1–1, 2020.

[9] Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming (Jack) Jiang. An automated approach to estimating code coverage measures via execution logs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 305–316, New York, NY, USA, 2018. Association for Computing Machinery.

[10] 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, 2012.

[11] Boyuan Chen and Zhen Ming (Jack) Jiang. Characterizing logging practices in Java-based open source software projects – a replication study in Apache Software Foundation. *Empirical Software Engineering*, 22(1):330–374, 2017.

[12] 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. ICSE Companion 2014, page 24–33, 2014.

[13] Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun Peter Chen. Studying the characteristics of logging practices in mobile apps: a case study on F-Droid. *Empirical Software Engineering*, 24, 12 2019.

[14] Nvidia. What is cuda? URL https://blogs.nvidia.com/blog/what-is-cuda-2/.

[15] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. A survey of methods for explaining black box models. *ACM Comput. Surv.*, 51(5), August 2018. ISSN 0360-0300.

[16] Patrick Loic Foalem, Foutse Khomh, and Heng Li. Studying logging practice in machine learning-based applications. *Inf. Softw. Technol.*, 170:107450, 2024.

[17] 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, 2019.

[18] Python Software Foundation. logging — logging facility for python. URL https://docs.python.org/3/library/logging.html.

[19] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721, 2013.

[20] Boyuan Chen and Zhen Ming (Jack) Jiang. Studying the use of java logging utilities in the wild. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 397–408. ACM, 2020.

[21] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, page 415–425. IEEE Press, 2015.

[22] Marco Alves and Hugo Paula. Identifying logging practices in open source python containerized application projects. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, SBES '21, page 16–20, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Weiyi Shang, Meiyappan Nagappan, Ahmed E. Hassan, and Zhen Ming Jiang. Understanding log lines using development knowledge. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 21–30, 2014.

[24] Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. Which variables should I log? *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[25] Zhenhao Li, An Ran Chen, Xing Hu, Xin Xia, Tse-Hsun Chen, and Weiyi Shang. Are they all good? studying practitioners' expectations on the readability of log messages. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 129–140. IEEE, 2023.

[26] Mary L McHugh. Interrater reliability: the kappa statistic. *Biochemia medica*, 22(3):276–282, 2012.

[27] Klaus Krippendorff and Joseph L Fleiss. Reliability of binary attribute data, 1978.

[28] Pinjia He, Zhuangbin Chen, Shilin He, and Michael R. Lyu. Characterizing the natural language descriptions in software logging statements. ASE 2018, page 178–189, 2018.

[29] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, Mark D. Syer, and Ahmed E. Hassan. Examining the stability of logging statements. *Empirical Softw. Engg.*, 23(1):290–333, February 2018. ISSN 1382-3256.

[30] Yiming Tang, Allan Spektor, Raffi Khatchadourian, and Mehdi Bagherzadeh. Automated evolution of feature logging statement levels using git histories and degree of interest. *Science of Computer Programming*, 2022. ISSN 0167-6423.

[31] Haonan Zhang, Yiming Tang, Maxime Lamothe, Heng Li, and Weiyi Shang. Studying logging practice in test code. *Empir. Softw. Eng.*, 27(4):83, 2022.

[32] 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. USENIX ATC '15, page 139–150, USA, 2015. USENIX Association.

[33] 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. SOSP '17, page 565–581, New York, NY, USA, 2017. Association for Computing Machinery.

[34] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Steve Sporea, Andrei Toma, and Sarah Sajedi. Log4perf: Suggesting logging locations for web-based systems' performance monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 127–138, 03 2018.

[35] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. Log4perf: suggesting and updating logging locations for web-based systems' performance monitoring. *Empir. Softw. Eng.*, 25 (1):488–531, 2020.

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

[37] Zishuo Ding, Heng Li, and Weiyi Shang. LoGenText: Automatically generating logging texts using neural machine translation. In *SANER*. IEEE, 2022.

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