An Adaptive Logging System (ALS): Enhancing Software Logging with Reinforcement Learning Techniques

Amirmahdi Khosravi Tabrizi
Brock University
St. Catharines, ON, Canada
akhosravitabriz@brocku.ca

Naser Ezzati-Jivan
Brock University
St. Catharines, ON, Canada
nezzati@brocku.ca

Francois Tetreault
Ciena
Ottawa, ON, Canada
ftetreau@ciena.com

ABSTRACT

The efficient management of software logs is crucial in software performance evaluation, enabling detailed examination of runtime information for postmortem analysis. Recognizing the importance of logs and the challenges developers face in making informed log-placement decisions, there is a clear need for a robust log-placement framework that supports developers. Existing frameworks, however, are limited by their inability to adapt to customized logging objectives, a concern highlighted by our industrial partner, Ciena, who required a system for their specific logging goals in resource-limited environments like routers. Moreover, these frameworks often show poor cross-project consistency. This study introduces a novel performance logging objective designed to uncover potential performance-bugs, categorized into three classes—Loops, Synchronization, and API Misuses—and defines 12 source code features for their detection. We present an Adaptive Logging System (ALS), based on reinforcement learning, which adjusts to specified logging objectives, particularly for identifying performance-bugs. This framework, not restricted to specific projects, demonstrates stable cross-project performance. We trained and evaluated ALS on Python source code from 17 diverse open-source projects within the Apache and Django ecosystems. Our findings suggest that ALS has the potential to significantly enhance current logging practices by providing a more targeted, efficient, and context-aware logging approach, particularly beneficial for our industry partner who requires a flexible system that adapts to varied performance objectives and logging needs in their unique operational environments.

CCS CONCEPTS

• Software and its engineering → Software maintenance tools; **Software post-development issues**; **• General and reference → Performance**; **Empirical studies**; **• Computing methodologies → Reinforcement learning**; **Machine learning**.

ACM Reference Format:
Amirmahdi Khosravi Tabrizi, Naser Ezzati-Jivan, and Francois Tetreault. 2024. An Adaptive Logging System (ALS): Enhancing Software Logging with Reinforcement Learning Techniques . In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3629526.3645033

1 INTRODUCTION

Logging, as an ubiquitous programming technique, involves the insertion of code that records key runtime information. Careful consideration of log placement is imperative, as the data captured by logs constitute a crucial source of information for postmortem analysis. In the event of system failures, logs often remain the only available source of data. For successful log analysis, it is crucial to have a strong underlying logging, as it directly influences the quality of the collected logs.

Given the importance of logging, it is crucial to strike a balance [22]. Logging too little could result in missing essential runtime information needed for postmortem analysis, making it challenging to diagnose failures in the field. On the other hand, logging too much brings its own set of problems. This includes an increased code volume that requires time for writing and maintenance. Moreover, it consumes additional system resources, impacting overall system performance, especially when dealing with high log volumes. Importantly, excessive logging may generate numerous trivial and unnecessary logs, masking crucial information and complicating issue identification.

Despite its importance, not all developers possess the necessary expertise to make informed logging decisions [4]. Previous research has presented various frameworks to aid developers in making logging decisions. Some frameworks help developers determine which parts of the system to log [4, 22], while others help to select appropriate log-levels for log statements [10, 11], and to effectively structure log messages [5, 8]. Mastropoalo et al. [12] introduced a comprehensive framework that integrates these three logging aspects using transformer models. However, there are some limitations towards existing frameworks, 1) they are limited to logging objective of the project that trained on and are not capable of adapting themselves to a desired logging objective. 2) None of them have considered performance-bugs as a logging objective. 3) Poor cross-project performance.

In collaboration with Ciena, we recognized the need for a logging system that not only adapts to various performance objectives but also addresses the specific logging needs in resource-constrained environments. In response, we introduce the Adaptive Logging System (ALS) that leverages reinforcement learning, along with a new logging objective—Performance Bug Logging Objective—designed to capture and reveal performance-bugs through logs. This dual approach of ALS, combining adaptability with a targeted logging objective, makes it particularly suitable for varied operational needs, enabling it to efficiently handle different performance objectives and logging requirements. Such a system is invaluable in environments where resource constraints and the need for efficient logging and performance analysis are critical.

Our contributions in this paper are as following:

- Proposing an Adaptive Logging system being able to adapt itself to self-defined logging objectives using reinforcement learning.
- Introducing *performance-bugs* logging objective to capture and reveal performance-bugs through logs.
- Creating a dataset that includes static source code features related to performance-bugs at the function-level for 17 different Apache and Django projects.
- Illustrating the cross-project efficiency of the proposed adaptive logging system by evaluating it on unseen environments.

This paper is structured as follows: Section 2 provides a background on software logging and reinforcement learning. Section 2.3 explores the motivations for our study. Sections 3 and 4 detail our empirical study on performance-bugs and the ALS framework, respectively. Section 5 presents our evaluation methodology and results. Finally, Section 5.4 discusses the study's limitations and future directions, and Section 6 summarizes our findings.

2 BACKGROUND AND LITERATURE REVIEW

2.1 Software Logging

The placement of log statements is guided by one or multiple logging objectives chosen by developers. These objectives include *Performance* [21], which focuses on minimizing performance overhead of logs; *Unexpected Situations* [4], which aim to identify errors; and *Execution Points* [4], which track system runtime states and execution path for root-cause analysis.

Developers face three crucial decisions when implementing a logging strategy. Firstly, they must determine the most appropriate location in the source code for logging (*Where to log?*) [4, 19, 21, 22]. Secondly, they must select the information to be logged and the log statements to be used (*What to log?*) [8]. Lastly, they must choose the appropriate log-level from a range of options including trace, debug, warn, info, error, and fatal (*Which log-level to choose?*) [13].

Prior studies have introduced log-placement frameworks to aid developers. Yuan et al. introduced ErrLog [19], a static programming method that adds logging statements using generic error patterns. Zhao et al. proposed Log20 [21], a DP-based framework that recommends near optimal log placements with low overhead. J. Zhu et al. developed LogAdvisor [22], a machine learning framework that automatically learns common logging rules and provides guidance.

2.2 Reinforcement Learning

In this part, we present the fundamental ideas of Reinforcement Learning (RL), a subfield of machine learning (ML), which we utilize in our research.

Definition. *Reinforcement Learning* is a goal-directed learning method that allows agents to solve sequential decision problems through trial-and-error and interaction [6, 17]. RL aims to determine the optimal mapping of situations to actions by maximizing a reward signal that represents the problem's *goal*.

A RL model consists of two components: Agent and environment. The Environment provides information on the system state, and the Agent selects actions based on that information and interacts with the Environment. The Environment updates the state and returns a reward after each action, creating a cycle of (state → action → reward), depicted in Figure 1, until a predetermined terminal state or timestep is reached.

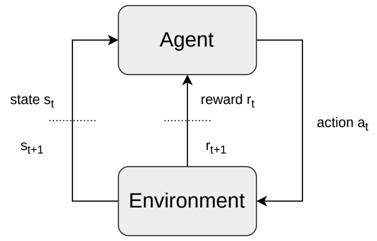

Figure 1: The reinforcement learning control loop

Difference from Other Methods. RL differs from other ML techniques such as supervised and unsupervised learning. Supervised learning requires external knowledge in the form of a training set, which specifies the correct behavior. However, in RL, this knowledge is not provided and must be acquired through the pursuit of objectives. In contrast, supervised learning only aims to learn the correct behavior provided, limiting it to that specific behavior. The objectives of RL and unsupervised learning are distinct from each other. Unsupervised learning aims to uncover hidden structures, while RL aims to maximize a reward signal to reach a specific goal.

The defining feature of RL models is their goal-seeking ability, which aligns them with the learning processes in humans and animals. This property provides RL with a high degree of adaptiveness. Its adaptive nature allows it to continually update and improve upon its knowledge, even when faced with unseen projects. This is a capability that is unique to RL, as other ML methods lack this adaptiveness.

2.3 Research Gaps

While the preceding sections have laid out the fundamental concepts in software logging and reinforcement learning, it is necessary to identify the existing limitations within these domains. This section briefly explains the motivation and research gaps in the current state of the art of log-placement frameworks, particularly highlighting the areas that remain unaddressed or inadequately tackled by existing solutions.

Logging Objectives. Traditional log-placement frameworks are typically limited in their scope, being designed to address specific logging objectives based on the datasets or environments they were originally trained in. This rigor presents significant challenges in dynamic and varied operational contexts, such as resource-limited environments, like network systems or routers. The logging objectives can vary greatly depending on the specific performance requirements and resource constraints of each system.

For instance, in a resource-constrained environment, the primary logging goal may be to minimize performance overhead while maximizing the utility of each log entry for effective bug detection and system monitoring. This differs from resource-abundant environments, where the emphasis may be on capturing comprehensive data for in-depth analysis. The ability to dynamically adjust log-placement strategies based on these different goals is crucial for maintaining system efficiency and reliability, yet it is currently lacking in existing frameworks [4, 22].

Moreover, the detection of performance-bugs, a critical aspect in ensuring the smooth operation of resource-limited systems, is often underrepresented in existing logging frameworks. While these frameworks primarily focus on capturing system errors and exceptions to aid in debugging and ensuring system reliability, they typically do not prioritize the identification and logging of performance anomalies. Effective logging of such anomalies is vital for preemptive maintenance and avoiding system downtimes, making the need for adaptable and performance-oriented logging frameworks even more essential.

Therefore, there is a strong need for log-placement frameworks that are not only adaptable to a broad spectrum of logging objectives but also sensitive to the unique demands of resource-limited settings. This adaptability is essential for tailoring logging strategies to effectively balance performance, resource utilization, and diagnostic needs, thereby enhancing the overall resilience and efficiency of the system.

Implementation Method. The complexity of log-placement, influenced by multiple factors such as system architecture, operational context, and specific performance requirements, poses significant challenges for a comprehensive and adaptable implementation. Traditional methods like *LogAdvisor* [22], while effective in certain settings, are limited by their lack of flexibility and cross-project accuracy. This constraint limits their effectiveness across diverse projects, particularly when transitioning from one domain or technology stack to another.

To address these limitations, we propose the Adaptive Logging System, which leverages Reinforcement Learning—a form of machine learning that excels in making decisions under uncertainty and adapting to new environments. By employing RL, ALS can dynamically learn from the specific characteristics and requirements of each project, continually refining its log-placement strategy to maximize efficiency and relevance. This capability enables ALS to provide tailored logging solutions that maintain high levels of accuracy and utility across various projects and environments, aligning with the diverse and evolving needs of modern software development.

3 LOGGING FOR PERFORMANCE-BUGS

To introduce performance-bugs as a logging objective, we must first gain a clear understanding of what they are. This understanding will enable us to subsequently establish appropriate metrics and features for their description. To achieve this, we conducted an empirical study of existing studies on performance issues. We have categorized performance issues into three distinct categories (Loops, Synchronization Issues and API Misuses) and devised 12 static source code features to characterize each of these categories (refer to Table 1). In the following parts of this section, we will provide detailed explanations for each category along with their corresponding features.

Table 1: performance-bugs and their defined features.

	Categories	Features
performance-bugs	Loops	- number-of-loops - nested-loop-level - loop-input-dependent-level
	Synchronization issues	- number-of-defined-threads - number-of-started-threads - number-of-join-threads - number-of-defined-locks - number-of-acquired-locks-threads - number-of-released-locks
	API misuses	- number-of-usage-of-extra - number-of-usage-of-order-by - number-of-usage-of-select-related

3.1 Loops

When addressing performance-bugs, loops create a challenging context for their occurrence. This is because loops have the potential to worsen the impact of performance-bugs, often accumulating issues across multiple iterations of the loop [3, 15, 16]. What's even more significant is that a large portion of performance-bugs occur within loops that depend on input data, accounting for nearly three-quarters of such cases [9]. This highlights the importance of paying close attention to loops in our efforts to mitigate performance-bugs.

We have established three distinctive features to encapsulate the vital aspects concerning loops, aimed at characterizing pertinent factors for integration into our RL model. These features are as follows:

(1) **Number of Loops** (number-of-loops): This feature quantifies the number of distinct loop constructs defined within a given function. This feature only counts the outer loops in case of having nested loops. For example, if a code snippet contains one nested loop (with a nesting level of 2) and one non-nested loop, the total number of loops in the snippet is 4, but the Number of Loops metric for the snippet is 2.
(2) **Nested Loop Level** (nested-loop-level): As a descriptor of loop complexity, this feature indicates the total number of loops within a function, including any nested loops. Using the previous example of a nested loop with a nesting level of 2 and a non-nested loop, the *Nested Loop Level* value here would be 4.
(3) **Loop Input Dependency Level** (loop-input-dependent-level): This feature identifies dependent variables, which are variables that rely on a function to determine their values, within the loop declaration. It provides information about the count of these dependent variables if any are found.

3.2 Synchronization Issues

Synchronization challenges can arise in concurrent programming, leading to performance issues [1, 7]. These problems often stem from the improper use of synchronization techniques, especially when selecting the wrong types of locks. Among the notable synchronization challenges, two stand out: deadlocks and race conditions.

Race conditions happen when two different threads try to change the same information at the same time without following a specific order. On the other hand, Deadlocks occur when several threads get stuck in their work because they are all waiting for something that another thread in the same group is using. We have identified six unique attributes related to synchronization challenges. These features are customized for Python's Thread library:

(1) **Number of Thread Objects** (number-of-defined-threads): This attribute indicates the count of threads that are defined within a given function.
(2) **Number of start() Function Calls** (number-of-started-threads): Following the creation of a Thread object, its activation is initiated by invoking the start() function associated with the created object. This feature quantifies the instances of start() function calls, representing thread activation within a specific function.
(3) **Number of join() Function Calls** (number-of-join-threads): The join() function, when invoked, causes the calling thread (typically the main thread) to wait until the thread on which join() is called terminates. This feature quantifies the instances of join() function calls.
(4) **Number of Lock Objects** (number-of-defined-locks): This attribute signifies the count of locks defined within a given function.
(5) **Number of acquire() Function Calls** (number-of-acquired-locks-threads): The acquire() method is used to acquire a lock. When a thread invokes acquire() for a lock, it gains ownership of the lock if it is available. If the lock is currently held by another thread, the calling thread will enter a blocking (waiting) state until the lock becomes available. Subsequently, when the thread successfully acquires the lock, it is able to execute the critical section of code that should be accessed by a single thread at a time. This attribute records the frequency of this method being called for a lock within a specific function.
(6) **Number of release() Function Calls** (number-of-released-locks): The release() method is employed to release a lock that the calling thread currently possesses. Upon the completion of the critical section of code protected by the lock, the thread is expected to invoke lock.release() to release the lock. This action enables other waiting threads to acquire the released lock. This attribute quantifies the instances of this method being called for a lock within a specific function.

3.3 API Misuses

API misuse refers to the incorrect use of an Application Programming Interface (API), which violates the implicit usage constraints set by the API. These constraints are in place to prevent errors and exceptions that can occur when the API is not used as intended. API misuse is a common cause of software bugs, crashes, and vulnerabilities [2, 9, 14, 20]. Guoliang Jin et al., [9] assert that more than one-quarter of software bugs are linked to API misuses. Based on our empirical study, we have categorized API misuses associated with performance-bugs into three distinct groups: Object-Relational Mapping (ORM) APIs, Deep Learning APIs, and Machine Learning Cloud API misuses. However, for the scope of this study, we focus exclusively on examining ORM API misuses, with a specific emphasis on Django ORM API misuses. We prioritize Django ORM because our projects primarily involve Python. The exploration of the other two categories is deferred to future research, as each requires a separate study to accurately identify the necessary features for precise characterization.

When it comes to Django ORM[1] API misuses there are some common misuses which can cause performance issues and need to be avoided [18]:

- Making complex queries: The more complex the query, the harder it is for the ORM to transform it into an actual database query. This can cause performance regressions, especially when dealing with large datasets. To avoid this, it is recommended to keep queries as simple as possible and avoid using sub-queries unless absolutely necessary.
- Retrieving too much data: When querying the database, it is vital to fetch only the required data. Retrieving excessive data can result in performance problems, particularly with large datasets. In Django ORM, there is a built-in method called select-related() that allows you to retrieve all relevant data in a single query instead of making multiple database queries. However, it is important to exercise caution and avoid overusing this method, as it can lead to fetching too much data and cause performance regression.
- Not using database-level constraints: Sorting records after fetching data from the database instead of sorting the database once can cause performance regressions. It is recommended to use database-level constraints instead of using order-by() to sort records at the query level.

Following three features represent the occurrence of three Django ORM functions which overusing them could potentially lead to one one of the performance issues mentioned above.

(1) **extra()** (number-of-usage-of-extra): Overusing extra() can make queries more complex by using sub-queries.
(2) **order_by()** (number-of-usage-of-order-by): Overusing order by() instead of using database-level constraints can lead to performance issues.
(3) **select_related()** (number-of-usage-of-select-related): Overusing select related() can also lead to retrieving too much data.

4 ADAPTIVE LOGGING SYSTEM (ALS)

To address the research limitation mentioned earlier, this study proposes an Adaptive Logging System, which utilizes Reinforcement Learning to provide a comprehensive log-placement framework. The adaptiveness of the system offers two key benefits: (1) it allows for the definition of any desired logging objective by modeling it into the RL's reward function, and (2) it ensures project independence of the framework by eliminating low cross-project accuracy.

ALS takes Python source code files as input and guides developers in identifying which functions in the provided source code should be logged to capture performance issues, along with the recommended log-level. In Figure 2, we can observe that ALS comprises three primary modules: web scraping, feature extraction, and an RL model. We will explore each of these modules in greater

[1]https://docs.djangoproject.com/en/3.2/topics/db/optimization/

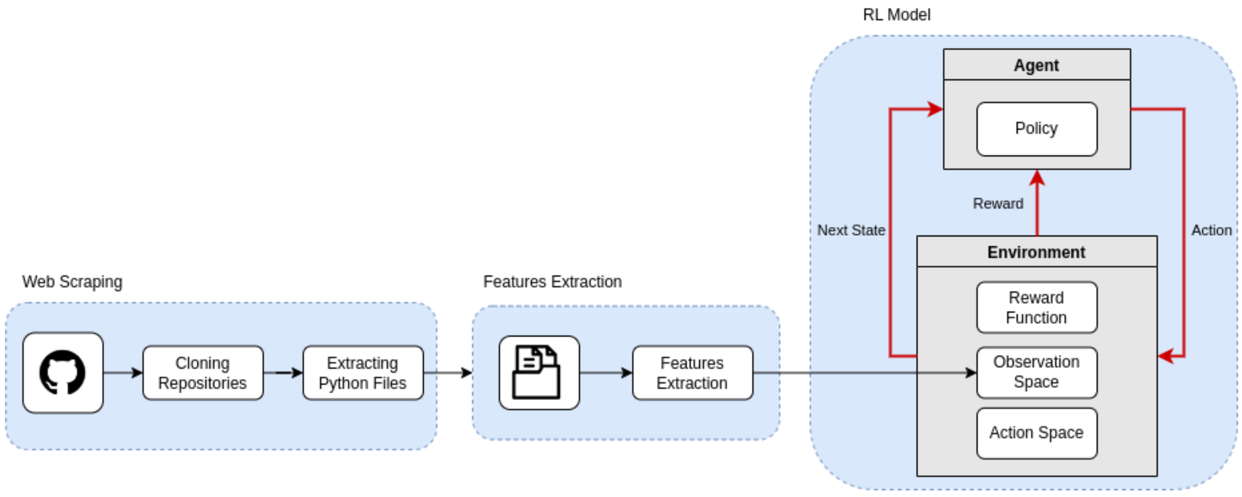

Figure 2: Adaptive logging system overview

detail in the subsequent parts of this section. The source code and training dataset for ALS are accessible via our git repository [2].

4.1 Web Scraping

Since ALS relies on an RL model at its core, the collection of a substantial amount of data is crucial for effective model training. Additionally, ALS is specifically designed for Python, requiring primary source code files to be in .py format. Manual data collection becomes impractical due to this requirement. The ALS web scraping module is designed to address these challenges.

This module's task is to download Python files from a given list of GitHub repositories using git command-line tool (PyGithub[3]). It accomplishes this in two main steps:

(1) clones each of the GitHub repositories to a specified local directory.
(2) Then, it scans through these cloned repositories to identify Python files and saves them in a separate directory.

4.2 Feature Extraction and Data Collection

This module has the important task of carefully extracting the features we defined in our previous discussion in part 3. However, successfully implementing it presents a significant challenge. This challenge revolves around obtaining the necessary access to comprehensively analyze the components of the source code.

To overcome this challenge, we have utilized Python's Abstract Syntax Tree (**AST**) library[4]. This advanced library provides us with the tools to explore the complex structures of the source code. This allows us to gain the required perspective to extract the specific static features accurately and with precision.

The process of building our dataset by collecting and extracting relevant features involves three distinct phases. In the following, we offer a comprehensive explanation of each of these crucial steps in our dataset preparation process.

Function Extraction (Phase 1). In the first step, we identify and isolate individual functions from the source code of the projects under investigation. For this, the feature extraction module takes the source code as input, generates its AST, and searches "FunctionDef" nodes in the generated tree to find the functions in the source code. The module do the same thing to every Python file obtained through web scraping, resulting in a comprehensive list of function nodes that serve as the foundation for our subsequent analysis and feature extraction.

Function Identification (Phase 2). Subsequently, in the second step, we assign meaningful and distinctive identifiers to each extracted function. This step is essential to facilitate seamless access to the origin source code which these functions belong to, laying the groundwork for subsequent in-depth investigations.

We follow a three-part approach: combining the function's name, the name of the file where the function is located, and its position (index) in the list of functions. Then, we compute the hash value of this combined string using the "hashlib" library[5]. This process generates the final, unique identifier (ID) for the function.

Feature Extraction (Phase 3). The third and final step involves the extraction of relevant features from the previously mentioned functions. These extracted features are then integrated into our dataset. Importantly, each row within the dataset is dedicated to a single function, along with its corresponding set of extracted features, ensuring a comprehensive and organized representation.

With the list of function nodes obtained from the preceding phase, we iterate through this list, conduct feature searches, extract the identified features, and seamlessly incorporate them into the

[2]https://github.com/amirmahdiKhosravi/Adaptive-Logging-System
[3]https://pygithub.readthedocs.io/en/stable/introduction.html
[4]https://docs.python.org/3/library/ast.html
[5]https://docs.python.org/3/library/hashlib.html

dataset. The methodology employed for this process remains consistent across all the features, involving a systematic traversal of the AST.

4.3 RL Model

The RL model is the core component of ALS. It enables ALS to adapt itself to different defined logging objectives through trial and error. It is divided into two main parts: the Agent and the Environment, which interact with each other to make decisions and learn from their interactions. This section focuses on detailing the various components of the RL model.

Environment. It is where the agent operates. It includes everything outside the agent and serves as the backdrop for the agent's actions and interactions.

To accurately describe the environment, it is crucial to identify its key components: the observation space, action space, and reward function. By precisely defining these three essential elements, we enable the RL model to make informed decisions and take suitable actions within this well-defined environment.

- **Observation Space** Observation space is the part that models the space in which the agent interacts. In our study, it is to demonstrate functions within the source code files. To represent functions as our observation space we take advantage of the collected dataset in 4.2, as each row of it represents a function by its related features. Equation 1 shows the observation space of the environment in the timestep t (S_t). It is a 1-dimensional Box[6] containing 12 features of our dataset. Also, our environment here is deterministic, meaning that the probability of the agent ending up in next state (S_{t+1}), while it is in S_t taking action A_t, is 1 (Equasion 2). The next state here is the next function of our dataset that the agent needs to take an action upon. Upon reaching the end of the dataset and the last function (terminal state), signifying the end of an episode, the agent seamlessly transitions back to the starting state of the environment—the first function of the dataset—to initiate a new episode of learning.

$$S_t = [feature_1, feature_2, feature_3, ..., feature_{12}] \tag{1}$$

$$P(S_{t+1}|S_t, A_t) = 1 \tag{2}$$

- **Action Space** It represents all the possible actions which the agent can take within the environment. In this study we defined a discrete action space which contains 5 different actions. These actions are defined in a way that makes it possible for the agent to make decision about either a function should be logged, and if so, what log-level should be chosen for it.
  Of the six existing log-levels (Trace, Debug, Info, Warn, Error, Fatal), we excluded Error and Fatal from the set of log-levels that our agent can select. This decision was based on the observation that Error and Fatal are primarily associated with runtime behavior, whereas our model relies solely on static source code features to make logging decisions.
  Equation 3 presents the action space of our environment. This set encompasses 5 distinct actions, each uniquely identified by a numerical assignment (Action ID). It is important to note that while there is an action labeled as "not-log," we deliberately omitted a separate "log" action. This decision was made because actions associated with specific log-levels inherently indicate the logging action. For example, selecting the "Trace" action implies that the corresponding function should be logged at the Trace level.

$$A = \{0 : not\_log,\ 1 : Trace,\ 2 : Debug,\ 3 : Info,\ 4 : Warn\} \tag{3}$$

- **Reward Function** The reward function guides the agent toward the goal it aims to achieve through its actions. To appropriately define the reward function, we need to incorporate our logging goal—deciding whether to log a function and the appropriate log-level—into it. This is accomplished through three main steps:
  - **Step 1:** We establish our fundamental rules for achieving our logging goal. These rules include: 1) logging functions that exhibit potential performance-bugs by examining performance-bug features in the observation space, and 2) assigning lower verbosity log-levels to performance-bugs with higher significance. This ensures that we capture all important information about performance-bugs, even when monitoring logs with the lowest verbosity. For this purpose, we assigned Trace and Debug log-levels to Synchronization issues, and Info and Warn log-levels to API misuses and Loops, respectively. If a function does not fall into one of these categories, it will not be logged.
  - **Step 2:** Based on the rules established in the previous step, we classify the actions performed by the agent into three distinct categories: "Good", "Intermediate" and "Bad". This categorization is determined through a comprehensive assessment of potential action outcomes: 1) Good: In scenarios where the agent's decision fully aligns with the criteria defined in the previous step, we provide the agent with a positive reward. For example, when a function exhibits potential synchronization issues based on its feature set in the observation space, the correct action is "Trace". The agent's action is deemed "Good" if it also selects "Trace" as the action. 2) Bad: Conversely, if the agent opts to log a function that should not be logged, or fails to log a function that should be logged based on the features extracted from the observation space, it incurs a negative reward as a punitive measure. 3) Intermediate: Any scenario falling outside the "Good" or "Bad" classifications is deemed "Intermediate." For example, if the agent correctly decides to log a function but selects an incorrect log-level, or if the action should have been categorized as "IDK" but the agent selects a different action, it does not receive a positive reward, as its behavior remains suboptimal. However, the negative reward incurred in such cases is comparatively smaller than the penalty associated with "Bad" behavior.
  - **Step 3:** The agent's reward depends on the category of its action and falls within a range of -4 to +2. We offer a

[6]https://stable-baselines.readthedocs.io/en/master/index.html

detailed breakdown of this reward scale for each possible category (good, bad, and intermediate) in the following section: **1) Good Reward (R_g)**: When the agent's action is categorized as "good," it receives an immediate reward of +1 (dense reward), supplemented by a sparse reward ranging from 0 to 1. The amount of the sparse reward depends on the function's vulnerability to performance-bugs, which is determined by the features within the function (S_t). To compute the sparse reward, we employ the Sigmoid function, with the coefficient providing adjustability for the slope of the Sigmoid curve (detailed elaboration is presented in chapters 4 and 5). Equation 5 shows the reward function for this specific category, denoted as R_g . ST is the sum of the 12 features in the vector S_t. **2) Bad Reward (R_b)**: The reward for the "bad" category is relatively straightforward, with the agent receiving the lowest reward within the range. In instances where the agent's action is categorized as "bad," it incurs a negative reward of -4 ($R_b = 4$). **3) Intermediate Reward (R_i)**: It is calculated as the negative absolute value of the difference between the action ID of the chosen action and the action that was intended to be selected. For instance, if the intended action is "Trace" (action ID = 1), but the agent selects "Warn" (action ID = 4), the reward will be -3. This relationship is depicted in Equation 6.

$$S_T = \sum_{f=0}^{11} S_t[f] \tag{4}$$

$$R_g(S_T) = 1 + \frac{1}{2}\left(\frac{1}{1 + e^{-\gamma S_T}}\right) \tag{5}$$

$$R_i = -|supposedActionID - takenActionID| \tag{6}$$

Agent. In this study, our primary focus is not on creating new RL algorithms. Instead, we choose to use well-established RL algorithms that already exist and apply them in our custom-designed environment. To make this possible, we rely on the Stable Baselines library, known for its high-quality implementations of Reinforcement Learning algorithms, all of which are based on OpenAI Baselines6 . Therefore, our approach involves making use of these pre-built and thoroughly developed algorithms from the Stable Baselines library. In part 5, dedicated to evaluation, we perform a comparative analysis to identify the most effective algorithm for our specific application.

5 EVALUATION

Our goal in this part is to thoroughly assess the practical usefulness and effectiveness of the Adaptive Logging System (ALS) in real-world scenarios by subjecting it to real-world software projects ranging from large-scale web applications to more specialized software, reflecting the diverse challenges faced by industry practitioners, such as those at Ciena. We will specifically explain how it performs in two distinct case studies, each offering its own set of challenges and opportunities.

In the upcoming sections, we detail our evaluation process. Section 5.1 explains our experimental setup, covering data collection, choice of RL algorithms, and libraries. Section 5.2 is the core, examining each case study, including ALS performance utilizing different RL methods and evaluations across projects. These sections showcase ALS performance in diverse scenarios.

Table 2: Dataset information

		Project Names	Number of Python files	Number of Functions
Training	Apache Projects	- Kibble - Libcloud - Allura - Spark	1421	56755
	Django Projects	- Connect - Chat-app - TrackTV		
Testing	Apache Projects	- Avro - Beam - Cloudstack - IoTDB - PLC4X - Thrift - Yetus	2814	36729
	Django Projects	- Django Website - Djangogirls Website - Django-jet		
Total			4235	93484

5.1 Experiment Setup

In this study, we provide a detailed account of our data collection process, shedding light on the datasets that serve as the foundation for our case studies. Then we shift our focus to configuring the RL model, using established RL algorithms from the Stable-Baselines library[7], which is implemented based on the OpenAI Baselines[8].

Data Collection. To prepare our dataset for training and testing the RL model of ALS, we have utilized the first two modules of the ALS framework. We conducted our experiments on a diverse set of 17 projects (Table 2), encompassing 11 Apache projects and 6 Django-based projects. The inclusion of Django projects was a deliberate choice, aimed at introducing diversity into our dataset, given that certain features, such as Django ORM API Misuses, tend to be less prevalent in Apache projects. These projects were selected based on their continued active status, ensuring their reliability as representative samples of ongoing software development projects.

As illustrated in Table 2, our dataset is partitioned into two distinct groups: training and testing. The training set is dedicated to training the RL model and comprises a larger volume of data, with 56,755 unique functions. The Testing dataset, explored in forthcoming sections, serves as the basis for our second case study and contains 36,729 functions. This approach not only ensures that our RL model is well-prepared with an extensive training dataset but also facilitates the assessment of its cross-project performance, as both datasets encompass a mix of Apache and Django projects. This diverse dataset allows us to evaluate the adaptability and effectiveness of our RL model on a wide array of software projects.

RL Model and Algorithms. With our experiment dataset in place, the next step involves configuring the RL model. For its environment we use our own environment which we defined in 4.3. On the other hand, for the agent, we use well-established RL algorithms available in the Stable-Baselines library.

[7]https://stable-baselines.readthedocs.io/en/master/index.html
[8]https://github.com/openai/baselines

Table 3: Stable-Baselines algorithms and features

	A2C	ACER	ACKTR	DDPG	DQN	GAIL	PPO	SAC	TD3	TRPO
Discrete Action Support	Yes	Yes	Yes	No	Yes	Yes	Yes	No	No	Yes
Box Observation Support	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
On/Off Policy	On	On	On	Off	Off	Off	On	Off	Off	On

For our RL model, we selected Deep Q-Network (DQN), Advantage Actor-Critic (A2C), and Proximal Policy Optimization (PPO) based on their proven efficacy in complex decision-making tasks. DQN's stability in discrete action spaces, A2C's balance between policy and value-based methods, and PPO's robustness in varying environments make them ideal for evaluating ALS's performance in log-placement.

Table 3 presents an overview of the built-in model-free RL algorithms integrated into the stable baselines framework. Within this set of algorithms, there are a total of 11 options at our disposal. However, due to the discrete nature of our action space, three of these algorithms, namely DDPG, SAC, and TDT, are not compatible.

It is noteworthy that among the remaining nine algorithms, only DQN and GAIL operate as off-policy algorithms, while the remainder are on-policy methods. This distinction is pivotal as it affects the manner in which these algorithms update their policies based on historical data.

This experimental setup, encompassing a diverse dataset and a range of RL algorithms, is specifically designed to test ALS's core objectives. The varied dataset ensures ALS's adaptability to different software architectures, while the selection of RL algorithms allows us to assess the system's effectiveness in making accurate log-placement decisions under varying conditions.

5.2 Case Studies

To evaluate feasibility of ALS and to see whether it fulfills our research goals we defined two different case studies which we introduce and explain in more details in the following. We used the reward received by the model as the evaluation metric, which we illustrate by demonstrating the learning curve of the RL model.

ALS and Different RL Methods. This case study is to assess how well ALS performs with various combinations of RL algorithms. For this we test the RL model by setting three different RL methods (DQN, A2C and PPO) to its agent and demonstrate the learning curve of each of them while they are applied on the training environment (dataset). This case study is to evaluate the following aspects of the ALS:

- The effectiveness and functionality of the ALS in adapting itself to the defined logging objective.
- A comprehensive assessment of the performance of various RL algorithms, shedding light on how effective suitable each of them are for our environment.

Cross-project Evaluation. In this case study we apply the three agents that have been previously trained on the training environment on the testing environment which includes functions from projects that the agents have not been introduced to. This case study is to evaluate the cross-project performance of the ALS.

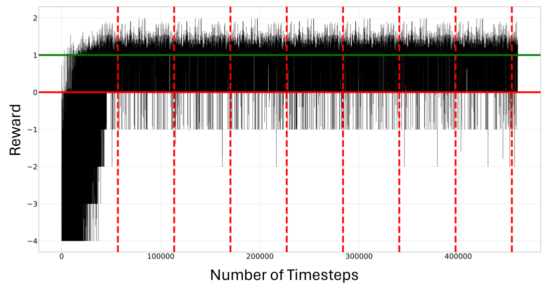

Figure 3: RL model's learning curve for DQN

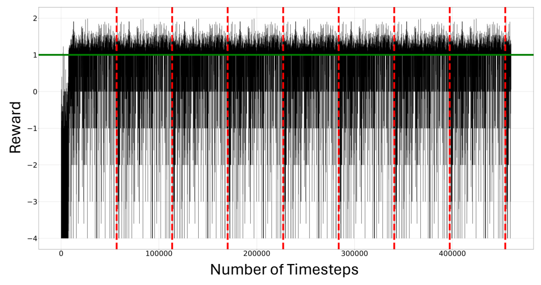

Figure 4: RL model's learning curve for A2C

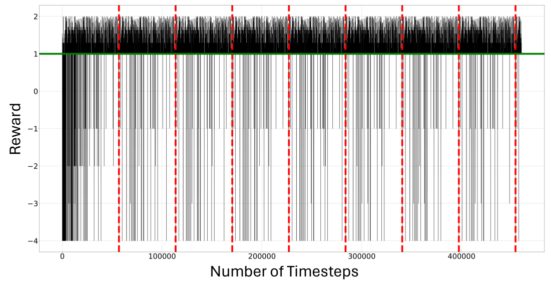

Figure 5: RL model's learning curve for PPO

5.3 Results and Discussion

Figures 3, 4 and 5 illustrate the outcomes of our first case study for DQN, A2C, and PPO, respectively. The broken vertical lines mark the end of each episode. Each model interacted with the environment across eight episodes. Initially, in the early timesteps, all three RL methods incurred negative rewards, signifying their initial struggle to make appropriate logging decisions, resulting in negative rewards. However, by the end of the first episode, all three models display a positive trend in reward values, showing

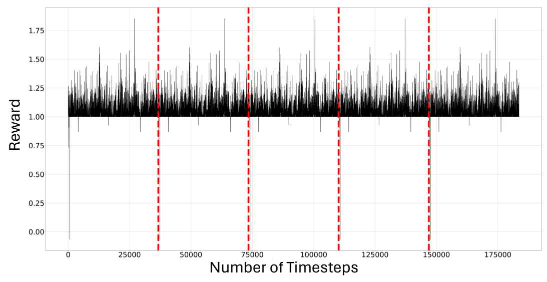

Figure 6: Evaluating DQN on testing environment

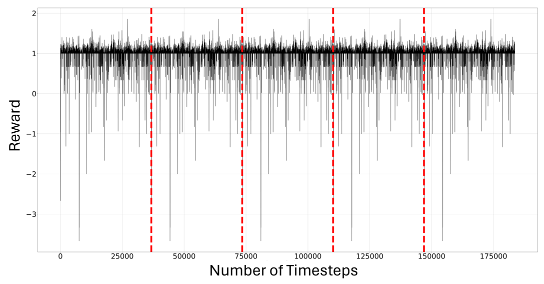

Figure 7: Evaluating A2C on testing environment

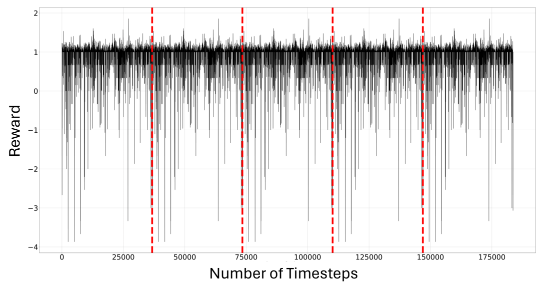

Figure 8: Evaluating PPO on testing environment

their capability to learn effective logging decisions and receive positive rewards. Notably, among the examined RL methods, DQN demonstrates the most stable results.

The results of our second case study are presented in Figures 6, 7 and 8. Observing the results, all three models exhibit a positive reward trend from the initial episode, even though they are being evaluated in a testing environment containing functions not encountered during their training. This underscores the reliable cross-project performance of ALS. The rationale behind this lies in the nature of RL methods, distinct from supervised learning; RL methods do not attempt to imitate the behavior of the training data or projects they encounter. Instead, they leverage this data to acquire more generalized knowledge applicable across diverse environments. In this context, DQN demonstrates the most stable results in the second case study.

Further analysis of these results reveals insights into the adaptability and effectiveness of different RL methods within ALS. The superior stability of DQN, for instance, suggests that its approach to learning and decision-making is particularly well-suited for the complexities of log-placement in varied software projects. This stability is critical when deploying ALS in real-world environments, where consistent performance is key to maintaining system reliability and efficiency. Additionally, the positive reward trends across all models highlight ALS's overall robustness and potential as a scalable solution for diverse logging needs. These findings are significant for industry applications, where adaptable and reliable logging strategies are essential for optimizing system performance and minimizing downtime, particularly in resource-constrained settings like those encountered by companies such as Ciena.

In summary, our evaluation results demonstrate: 1) ALS's functionality and its ability to adapt to a predefined logging objective, 2) its reliable cross-project performance, and 3) the compatibility of DQN as the most stable RL method for our environment, outperforming A2C and PPO in terms of stability.

The adaptability and effectiveness of ALS in diverse settings, as demonstrated by our evaluations, are particularly pertinent for our industrial partner, Ciena. In their resource-limited environments, the ability of ALS to dynamically adjust log-placement strategies is critical for maintaining system efficiency and reliability. These attributes of ALS not only meet the specific requirements of Ciena but also exemplify the system's potential for broader application in similar industrial contexts, where flexible and efficient logging solutions are critical.

5.4 Limitations

While our proposed Adaptive Logging System framework, utilizing Reinforcement Learning, effectively achieved its predefined logging objective of identifying performance-bugs, we identified certain limitations during the evaluation. These limitations open avenues for further enhancements:

(1) **Covered Programming Languages**: At present, ALS is limited to use in Python projects. This limitation arises from the tool[9] we use to search the Abstract Syntax Tree of the source code, which is designed specifically for Python. To enhance the feature extraction module of ALS, future studies could explore either using more comprehensive tools that support other programming languages or leveraging Large Language Models (LLMs) such as GPT-4[10], along with feature extraction models like Jina[11].
(2) **Dynamic Performance Features**: The current iteration of ALS focuses on specific performance-related logging objectives. Future versions could benefit from incorporating dynamic performance metrics, such as CPU and memory

[9]https://docs.python.org/3/library/ast.html
[10]https://openai.com/gpt-4
[11]https://huggingface.co/jinaai/jina-embeddings-v2-base-code

overhead, which were not explored in this study. Integrating these metrics into the RL model's reward function could enable ALS to tackle more complex performance logging objectives, enhancing its applicability in various software environments.

(3) **RL Model Architecture**: In our study, ALS employs a single RL agent for both logging decisions and log-level determination. Future research might explore a dual-agent approach, with one agent dedicated to logging decisions and another to log-level determination. This two-tiered approach could provide more precise control and potentially improve overall effectiveness.

These limitations highlight areas for potential improvement and demonstrate the evolving nature of adaptive logging systems in the field of software performance engineering.

6 CONCLUSION, AND FUTURE DIRECTIONS

In this research, we developed the Adaptive Logging System, a novel logging framework that utilizes Reinforcement Learning to adapt dynamically to varying logging objectives, with a focus on identifying and mitigating performance bugs. Our approach began with an empirical study categorizing performance bugs into three main types: Synchronization issues, Loops, and API misuses. This classification guided the definition of 12 distinct static source code features, which formed the basis of our dataset. Training ALS's RL model on this dataset, we evaluated its performance across 17 Django and Apache projects. The results were promising, demonstrating ALS's effectiveness in adapting to different logging objectives. Notably, the Deep Q-Network (DQN) model showed the most stable results in terms of learning curve, performing well in both training and cross-project evaluation scenarios.

The adaptability and robustness of ALS, while particularly beneficial for Ciena in their resource-limited settings, extend its significance to a wider range of industrial applications. The system's ability to tailor its logging strategies is not only essential for Ciena's operational efficiency and reliability but also indicative of its potential impact in broader software logging and performance evaluation contexts.

However, this research study has certain limitations. The selection of RL algorithms and the scope of our dataset, while extensive, may not fully capture the array of scenarios in different software environments. Moreover, our focus was primarily on function-level decisions, without delving into more detailed log-level determinations.

Looking forward, there are several opportunities to expand this research. One avenue of interest is the potential integration of ALS with LLMs, where ALS could serve as the RLHF (Reinforcement Learning from Human Feedback) reward model to optimize LLMs for logging solutions. Another possibility is integrating ALS with existing logging frameworks, which could lead to a more comprehensive logging solution. By combining ALS's dynamic adaptability with the proven capabilities of language models and traditional systems, this approach could greatly enhance the overall effectiveness of software logging practices.

We further collaborated with them to develop an adaptive logging system for their resource constrained environment. This paper discusses our first steps in that direction, both defining a logging system that solves their challenges and evaluating that system on open source systems. This work is defined as a first gate towards testing a system in production. In such a project, ALS's effectiveness in operational environments would be validated more accurately, and essential feedback from users and developers would be gathered. This feedback is vital for fine-tuning ALS, ensuring it meets the complex demands of real-world applications. Future plans in this regard include extensive deployment and evaluation within Ciena's operational context, aiming to demonstrate the system's practical utility and inform further development.

Further development could also involve incorporating dynamic performance metrics such as CPU and memory usage into the RL model, offering a more detailed understanding of system demands. Moreover, exploring a dual-agent architecture in the RL model—dividing responsibilities between logging decisions and log-level determination—could potentially refine the system's precision and efficiency.

REFERENCES

[1] Alam, M. M. U., Liu, T., Zeng, G., and Muzahid, A. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), pp. 298–313.

[2] Amann, S., Nguyen, H. A., Nadi, S., Nguyen, T. N., and Mezini, M. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering 45*, 12 (2018), 1170–1188.

[3] Cao, J., Chen, B., Sun, C., Hu, L., Wu, S., and Peng, X. Understanding performance problems in deep learning systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2022), pp. 357–369.

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

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

[6] Graesser, L., and Keng, W. L. *Foundations of deep reinforcement learning.* Addison-Wesley Professional, 2019.

[7] Gu, R., Jin, G., Song, L., Zhu, L., and Lu, S. What change history tells us about thread synchronization. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015), pp. 426–438.

[8] He, P., Chen, Z., He, S., and Lyu, M. R. Characterizing the natural language descriptions in software logging statements. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (2018), pp. 178–189.

[9] Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu, S. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices 47*, 6 (2012), 77–88.

[10] Kim, T., Kim, S., Park, S., and Park, Y. Automatic recommendation to appropriate log levels. *Software: Practice and Experience 50*, 3 (2020), 189–209.

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

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

[13] Mizouchi, T., Shimari, K., Ishio, T., and Inoue, K. Padla: a dynamic log level adapter using online phase detection. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)* (2019), IEEE, pp. 135–138.

[14] Ren, X., Ye, X., Xing, Z., Xia, X., Xu, X., Zhu, L., and Sun, J. Api-misuse detection driven by fine-grained api-constraint knowledge graph. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (2020), pp. 461–472.

[15] Sandoval Alcocer, J. P., Bergel, A., and Valente, M. T. Learning from source code history to identify performance failures. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering* (2016), pp. 37–48.

[16] Song, L., and Lu, S. Performance diagnosis for inefficient loops. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (2017), IEEE, pp. 370–380.

[17] Sutton, R. S., and Barto, A. G. *Reinforcement learning: An introduction.* MIT press, 2018.

[18] Yang, J., Subramaniam, P., Lu, S., Yan, C., and Cheung, A. How not to structure

your database-backed web applications: a study of performance bugs in the wild. In *Proceedings of the 40th International Conference on Software Engineering* (2018), pp. 800–810.

[19] Yuan, D., Park, S., Huang, P., Liu, Y., Lee, M. M., Tang, X., Zhou, Y., and Savage, S. Be conservative: Enhancing failure diagnosis with proactive logging. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (2012), pp. 293–306.

[20] Zhang, Y., Kabir, M. M. A., Xiao, Y., Yao, D., and Meng, N. Automatic detection of java cryptographic api misuses: Are we there yet? *IEEE Transactions on Software Engineering 49*, 1 (2022), 288–303.

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

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