Where Shall We Log? Studying and Suggesting Logging Locations in Code Blocks

Zhenhao Li
l_zhenha@encs.concordia.ca
Concordia University
Montreal, Quebec, Canada

Tse-Hsun (Peter) Chen
peterc@encs.concordia.ca
Concordia University
Montreal, Quebec, Canada

Weiyi Shang
shang@encs.concordia.ca
Concordia University
Montreal, Quebec, Canada

ABSTRACT

Developers write logging statements to generate logs and record system execution behaviors to assist in debugging and software maintenance. However, deciding where to insert logging statements is a crucial yet challenging task. On one hand, logging too little may increase the maintenance difficulty due to missing important system execution information. On the other hand, logging too much may introduce excessive logs that mask the real problems and cause significant performance overhead. Prior studies provide recommendations on logging locations, but such recommendations are only for limited situations (e.g., exception logging) or at a coarse-grained level (e.g., method level). Thus, properly helping developers decide finer-grained logging locations for different situations remains an unsolved challenge. In this paper, we tackle the challenge by first conducting a comprehensive manual study on the characteristics of logging locations in seven open-source systems. We uncover six categories of logging locations and find that developers usually insert logging statements to record execution information in various types of code blocks. Based on the observed patterns, we then propose a deep learning framework to automatically suggest logging locations at the block level. We model the source code at the code block level using the syntactic and semantic information. We find that: 1) our models achieve an average of 80.1% balanced accuracy when suggesting logging locations in blocks; 2) our cross-system logging suggestion results reveal that there might be an implicit logging guideline across systems. Our results show that we may accurately provide finer-grained suggestions on logging locations, and such suggestions may be shared across systems.

ACM Reference Format:
Zhenhao Li, Tse-Hsun (Peter) Chen, and Weiyi Shang. 2020. Where Shall We Log? Studying and Suggesting Logging Locations in Code Blocks. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416636

1 INTRODUCTION

Logs play an important role in maintaining software systems and diagnosing issues that happen during runtime. Developers rely on logs for various maintenance activities, such as debugging [25, 69, 71], testing [13–15, 47], and system comprehension [50, 51]. Developers insert logging statements in the source code with different verbosity levels (e.g., *trace*, *debug*, *info*, *warn*, *error*, and *fatal*) to record system execution information and values of dynamic variables. For example, in the logging statement: *log.warn("Invalid groupingKey:", + key)*, the static text message is *"Invalid groupingKey:"*, and the dynamic message is the value of the variable *key*. The logging statement is at the *warn* level, which is the level for recording information that may potentially cause system oddities [3].

The great value of logs results from proper logging decisions that are made by practitioners during software development [39]. The logging decisions are often made in order to balance the benefit and cost from logs [39]. On one hand, inserting too few logging statements may increase the maintenance difficulty due to missing important system execution information for debugging and analysis [78]. On the other hand, inserting too many logging statements may increase system performance overhead and produce excessive trivial logs which increase the difficulty of log analysis [39, 66, 78]. However, such a crucial task of making logging decisions remains challenging due to the lack of concrete logging specification and guidelines [25]. As a result, developers have to rely on their intuitions and experiences to compose, review, update and even fix logging statements in an ad-hoc manner [11, 29, 41, 44, 45, 70].

To address the challenge of making logging decisions, prior studies [20, 25, 38, 67, 78] provide automated recommendations on logging locations. However, there exist two main limitations in prior research: 1) Such recommendations are often only for a very limited number of situations. Approaches from prior research may only provide suggestions for exception handling blocks and method return values [25, 78]. 2) The logging locations are recommended at a coarse-grained level, e.g., method level [38]; while practitioners still need to decide the specific location to place a logging statement inside a method. As a result, in many cases, practitioners often still face challenges when making decisions on logging locations, despite the advance from recent research outcomes.

In this paper, we conduct a study to uncover guidelines and provide suggestions on logging locations (i.e., where do developers log) at a finer-grained level (i.e., block level) by analyzing logging statements and their surrounding code. Through a manual study on the logging statements from seven open source systems, we find that the decisions of logging location are often influenced by both the syntactic and semantic information in the source code. Moreover, the logging statements often record execution information related to the block in which they reside. Driven by our manual study results, we extract syntactic (e.g., nodes in abstract syntax trees) and semantic (e.g., variable names) information from the source

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416636

code and propose an automated deep learning based approach to suggest logging locations at the block level. We find that our deep learning models outperform the baseline, and the syntactic block feature achieves the best results (an average balanced accuracy of 80.1%) compared to semantic and fused (a fusion of syntactic and semantic) features. Moreover, syntactic information of blocks may be leveraged to provide general logging guidelines across different software systems. In summary, this paper makes the following contributions:

- We uncover six categories of logging locations, which are exception logging in *catch* blocks, branch logging in blocks associated with decision-making statements, program iteration logging, logging the start or the end of a method, and function logging in domain-specific methods. We also discuss the common types of information that is recorded in each category.
- We propose a deep learning based approach to suggesting logging locations at the block level by leveraging syntactic, semantic and fused block features extracted from the source code[1]. We find that models trained using the syntactic features have the highest balanced accuracy (80.1%) among the three types of features. Although there are some differences in the suggestion results among the three features, syntactic features can capture around 80% of all the suggested true positives. Our finding shows that most logging decisions may be related to the syntactic structure of the code.
- The cross-system suggestion results achieve an average balanced accuracy of 67.3%. We also find that there is a moderate to substantial agreement among the cross-system models trained using the syntactic features, which shows that developers of different systems may follow certain implicit guidelines on deciding logging locations.

Paper organization Section 2 discusses the background and related work of our study. Section 3 describes the setup of our manual study and the categories of logging locations we find. Section 4 discusses how do we extract block-level features and describes our deep-learning based approach. Section 5 presents the evaluation metrics and the results by answering two research questions. Section 6 discusses the False Positives and False Negatives in our suggestion results. Section 7 discusses the threats to validity of our study. Section 8 concludes the paper.

2 BACKGROUND AND RELATED WORK

Developers insert logging statements into the source code to record system runtime information and use the generated logs to assist in software debugging and maintenance. For example, as shown in the simplified code snippet from Zookeeper below, the logging statement is at the *error* level, contains the static message *"Missing count node for stat"*, and records the dynamic value of the variable *statNode*.

```
DataNode node = nodes.get(statNode);
if (node == null) {
    // should not happen
    LOG.error("Missing count node for stat {}", statNode);
    return;
}
```

[1]We share the replication package of this paper at: https://github.com/SPEAR-SE/ASE2020_Logging_Location_Data.

The logging statement is closely related to the specific value of the DataNode object and records an unexpected execution behavior in an *if* block when the value of the node is *null*. Helping developers decide where to log is an on-going research problem. Fu et al. [25] studied where do Microsoft developers add logging statements in their projects written in C# and focused on studying the characeristics of logging in some specific code snippets (i.e., catch blocks and return value checks). They found that developers often add logging statements to check the returned value of a method and record exceptions. Zhu et al. [78] further extended the work by providing a tool for suggesting log placement in the two above-mentioned cases. Li et al. [38, 41] provide suggestions on whether a method or commit requires a logging statement. In short, prior studies either only target a limited number of logging locations or provide a coarse-grained suggestion. Therefore, in this paper, we explore the potential of providing a finer-grained support on deciding general logging locations through a manual study (Section 3) and propose an automated deep-learning based approach to suggest logging locations at the code block level (Section 4).

Below, we further discuss the related works of this paper.

Studies on Logging Practices. There are several studies on characterizing the logging practices in software systems. Yuan et al. [70], Chen et al. [9], and Zeng et al. [72] conducted quantitative characteristics studies on log messages in large-scale open source C/C++, Java systems, and mobile applications. Chen et al. [12] studied the logging utilities, and Zhi et al. [77] studied the logging configurations in Java. They found that logs are essential for debugging and maintenance.

Given the importance of logs, other studies try to help developers improve logging practices. Chen et al. [10] found that developers commonly make some mistakes when writing logging statements (e.g., logging objects whose values may be null) and concluded five categories of logging anti-patterns from code changes. Hassani et al. [29] identified seven root-causes of the log-related issues from log-related bug reports and found that inappropriate log messages and missing log statements are the most common issues. Li et al. [43, 46] uncovered potential problems with logging statements that have the same text message and developed an automated tool to detect the problems. Yuan et al. [71] proposed an approach that can automatically insert additional variables into logging statements to enhance the error diagnostic information. Li et al. [40] propose the use of prediction models to suggest the log level of a newly added logging statement. Liu et al. [48] proposed a deep learning framework to suggest the variables that should be recorded in logging statements.

Different from prior studies, this paper focuses on studying logging locations in the purpose of providing suggestions and guidelines on the decisions of logging locations. The findings and approaches in this paper can complement prior studies in providing more comprehensive logging supports to developers.

Applying Deep Learning in Software Engineering Tasks. Due to the advances in deep learning, recent research starts to investigate source code representation and apply deep learning models in software engineering tasks. Zhang et al. [74] proposed an AST-based neural network for source code representation. They evaluated their approach on several software engineering tasks,

Table 1: An overview of the studied systems.

System	Version	LOC	NOL	#LB	#NLB	%LB
Cassandra	3.11.4	432K	1.3K	1.0K	25.0K	3.8%
ElasticSearch	7.4.0	1.50M	2.5K	1.9K	54.0K	3.4%
Flink	1.8.2	177K	2.5K	2.4K	27.5K	8.0%
HBase	2.2.1	1.26M	5.5K	4.1K	81.1K	4.8%
Kafka	2.3.0	267K	1.5K	1.0K	9.0K	10.0%
Wicket	8.6.1	216K	0.4K	0.3K	9.1K	3.2%
Zookeeper	3.5.6	97K	1.2K	0.9K	5.2K	14.8%

Note: LOC refers to the lines of code, NOL refers to the number of logging statements, #LB and #NLB refers to the number of *logged* and *non-logged* blocks respectively, %LB refers to the percentage of *logged* blocks over all the blocks.

such as source code classification and code clone detection, and the results outperformed existing approaches. Tufano et al. [58] evaluated different representation of source code (e.g., abstract syntax tree and control flow graph) and their effect on applying deep learning models in SE tasks. Hu et al. [32] proposed a deep learning based approach to automatically generate comments for Java methods. Nghi et al. [52] applied deep learning models to identify the programming language used in an algorithm. Different from prior studies, we focus on extracting source code features to suggest which blocks need to be logged. We conduct a comprehensive manual study on the characteristics of logging locations and propose a deep learning based approach to provide automated suggestions.

3 STUDYING THE CHARACTERISTICS OF LOGGING LOCATION IN CODE BLOCKS

To better understand developers' logging decisions and provide more concrete logging suggestions, in this section, we manually inspect the logging statements and their surrounding code. We examine if there exist finer-grained (e.g., at code block levels) implicit or explicit common characteristics of the locations where developers insert logging statements.

Studied Systems. We conduct a manual study on seven large-scale open source Java systems: Cassandra, Elasticsearch, Flink, HBase, Kafka, Wicket, and ZooKeeper. Table 1 shows an overview of the systems. The studied systems cover different domains (e.g., message broker, search engine, and database), have high quality logging code, and are commonly used in prior log-related studies [10, 11, 38, 46]. The size of the studied systems ranges from 97K to 1.5M LOC, and they contain from 0.4K to 5.5K logging statements.

Manual Study Setup. Our goal is to manually inspect the logging statements and their surrounding code to study the characteristics of logging locations. To prepare the data for our manual study, we extract the logging statements from the source code by implementing a static code parser. Our parser identifies every logging statement that invokes common logging libraries (e.g., Log4j [3] and SLF4J [5]) in the code. Then, for each logging statement, we extract its static message and dynamic variables, its verbosity level, its location (i.e., the file and method that contains the logging statement), and its surrounding code (i.e., the method that contains the logging statement). After getting all the logging statements and the extracted information, we randomly sample 375 out of 14.9K logging statements based on a 95% confidence level and 5% confidence interval [8]. For each sampled logging statement, we study its structural information and data flow of the surrounding code, in order to see the potential factors taking part in the decision of inserting the logging statements in a block. Specifically, the first two authors of this paper (i.e., A1 and A2) follow an open coding-like process similar to prior studies [21, 36, 46, 73], and involve in the following three phases to conduct the manual study:

Phase I: A1 studies 100 randomly sampled logging statements and their extracted information, and record the characteristics of their data flow, structural, and semantic information (e.g., the dependency of variables, control flow, and the business logic of the code). A1 further derives a draft list of categories of logging locations based on the information recorded. Then A1 and A2 follow the draft list to label the 100 samples collaboratively. During this phase, the categories are revised and refined.

Phase II: A1 and A2 independently assign the categories derived in Phase I to the rest of the 375 sampled logging statements. There is no new category derived in this phase.

Phase III: A1 and A2 compare the assigned categories in Phase II. Any disagreement of the categorization is discussed until reaching a consensus. No new categories are introduced during the discussion. The results in this phase have a Cohen's Kappa of 0.86, which is a substantial-level of agreement [56].

Categories of Logging Locations. In our manual study, we uncover six categories of logging locations that are associated with four different types of blocks (i.e., try-catch, branching, looping, and method declaration). In particular, we find that three categories are associated with try-catch, branching, and looping blocks; and three categories are associated with method declaration blocks that record method execution information. Below, we discuss each category in detail with an example.

Category 1 (Try-Catch Block): Exception information logging in catch blocks (122/375, 32.5%). Exceptions are widely used to capture errors. Developers rely on logs for debugging and error diagnostics when exceptions occur [25, 68]. The code snippet below shows an example of logging statements in this category. Similar to a prior study [25], we find that a large number of sampled logging statements reside in catch blocks. Most of them are at *error* (52/122, 42.6%) or *warn (46/122, 37.7%)* level. The logging statements often record messages or execution information related to the prior try block.

```
try {
    listener.onCache(shardId, fieldName, fieldData);
} catch (Exception e) {
    logger.error("Failed to call listener on atomic field data loading", e);
}
```

Category 2 (Branching Block): Branch logging in blocks associated with decision-making statements (139/375, 37.1%). We find that many sampled logging statements reside in blocks associated with decision-making statements [4] (e.g., if-else and switch) to record the execution information in different branches. The variable or the invoked method in the condition of the decision-making statement (e.g., the arguments in the if statement) are processed or defined in the prior code. Among the logging statements in this category, around half of them (68/139, 48.9%) record the occurrence of an unexpected execution behavior (e.g., an error or a failure) with a *warn* level or above (e.g. *error*), as shown in the code snippet below. The remaining cases record the occurrence of a normal execution behavior with an info, debug, or trace level for system comprehension or debugging purposes.

```
final TaskId id = partitionsToTaskId.get(tp);
...
```

```
if (id != null) {
    taskIds.add(id);
} else {
    log.error("Failed to lookup taskId for partition {}", tp);
}
```

Category 3 (Looping Block): Program iteration logging (25/375, 6.7%). We find that some sampled logging statements reside in blocks that are associated with looping statements [4, 65] (e.g., code blocks that are associated with for, while, and do-while statements). These logging statements often record the execution state during iterating (e.g., recording the i_{th} execution inside a for block) or variables that are processed or defined in prior blocks. We also find that no logging statements under this category are at *error* or *fatal* level. All logging statements are at the level of *info* (13/25, 52.0%), *debug* (6/25, 24.0%), or *trace* (6/25, 24.0%). In short, developers are more likely to add logging statements in such blocks for debugging and recording program execution.

```
IndexStatistics[] stats = getIndexStatistics(total);
...
for (IndexStatistics s : stats) {
    LOG.info(" Object size " + s.itemSize() + " used=" + s.usedCount());
}
```

Category 4 (Method Declaration Block): Logging the start of a method (33/375, 8.8%). We find that some sampled logging statements reside at the beginning of a method, mostly for recording the program execution state or debugging purposes. These logging statements record the start of the method execution (e.g., *"Start to build the program from JAR file."*) at the *info* (21/33, 63.6%), *debug* (8/33, 24.3%), or *trace* (4/33, 12.1%) level. Different from other categories, we do not find the location of logging statements in this category depend on prior code in the method. However, we find that these logging statements record the execution of some methods of which the process is important to know and with some specific semantics in the code (e.g., recovery(), perform(), and queue()), as shown in the code snippet below.

```
public void perform() throws Exception
{
    LOG.info(String.format("Performing action: Rolling batch restarting {} of
          region servers", (int)(ratio * 100)));
    List<ServerName> selectedServers = selectServers();
    Queue<ServerName> serversToBeKilled = new LinkedList<>(selectedServers);
    ...
    //code for performing server-killing related tasks
    ...
}
```

Category 5 (Method Declaration Block): Logging the end of a method (27/375, 7.2%). In this category, the logging statements reside at the end of a method, recording the successful method execution (e.g., *"Removed job graph from ZooKeeper"*, as shown in the code snippet below). We find that most of them (22/27, 81.5%) are at the *info* level, and the rest are in *debug* (3/27, 11.1%), *trace* (1/27, 3.7%) and *warn* (1/27, 3.7%) level, which may show that such logs are mostly for debugging and recording program execution. The logging statement may record variable values that are declared or modified in prior blocks when the method execution finishes. Similar to *Category 5*, we find that the logging statements in this category might reside in semantically similar methods of which the execution is important to be recorded (e.g., shutdown(), delete(), and remove()).

```
public void removeJobGraph(JobID jobId) throws Exception
{
    checkNotNull(jobId, "Job ID");
    String path = getPathForJob(jobId);
    ...
    addedJobGraphs.remove(jobId);
    //code for removing ZooKeeper job graph
    ...
    LOG.info("Removed job graph {} from ZooKeeper.", jobId);
}
```

Category 6 (Method Declaration Block): Function logging in domain-specific methods (29/375, 7.7%). We find that developers sometimes insert logging statements in some domain-specific methods (e.g., handling a specific request) to record the execution of this method. We also find that these methods are usually very short (i.e., within 10 lines of code). As shown in the example below, in the method handleResponse() in Elasticsearch's *JoinHelper.java*, there are only a few lines of functional code statements but has a logging statement recording the execution behavior of this method. Among the logging statements in this category, 21/29 (72.4%) are at the *info* level or below (i.e., *debug* or *trace* level) to record the methods handling normal requests, and the rest 8/29 (27.6%) logging statements are at the *warn* or *error* level to record the methods handling abnormal situations (e.g., onFailure()). We also find that these short methods might be semantically similar based on our manual observation (e.g., share many common words such as *handle* and *execute*).

```
public void handleResponse(Empty response) {
    pendingOutgoingJoins.remove(dedupKey);
    logger.debug("successfully joined {} with {}", destination, joinRequest);
    lastFailedJoinAttempt.set(null);
}
```

In summary, our findings show that there may be an implicit logging guideline that developers follow in the studied systems. Both **syntactic** and **semantic** information are important considerations in such logging guidelines. In particular, we find that 76.3% (286/375, combining *Category 1*, *Category 2* and *Category 3*) of the sampled logging statements are related to recording information in blocks associated with **syntactic** information of the source code (e.g., try-catch, branching, or looping blocks). These logging statements also often record information (e.g., variable values or execution states) that is related to prior blocks. We find that 23.7% (89/375, combining *Category 4*, *Category 5* and *Category 6*) of the sampled logging statements may be inserted based on the **semantic** information (i.e., business logic) of the method inside the method declaration block. These logging statements often record the start and end of method execution, or record the execution of some domain-specific methods (e.g., request handling or task execution).

> By uncovering six categories of logging locations, we find that both **syntactic** and **semantic** information are important considerations in such logging guidelines. 76% of the sampled logging statements are related to recording exception, branching, and program iteration; while 24% are related to recording the start, end, or execution of certain methods.

4 AUTOMATICALLY SUGGESTING LOGGING LOCATIONS AT THE CODE BLOCK LEVEL

As we find in Section 3, developers usually insert logging statements to record the behavior or state of the program in blocks (e.g., exception handling in catch blocks or branch logging in if/else blocks). We also find that some logging locations may be related to the semantics of a method (e.g., recording the start of a certain method execution). Hence, such syntactic and semantic information may

compose implicit logging guidelines that developers follow when deciding on logging locations. In this section, we seek to explore the potential of automatically suggesting logging locations at the block level. Such an automated approach might further assist developers in making logging decisions and improving logging practice. Below, we describe our approaches that extract block features and build a deep learning model to suggesting logging locations.

4.1 Extracting Block Features

Identifying Logged Blocks. Our goal is to provide suggestions on deciding blocks that require logging statements. We choose to provide a suggestion at the block level because as we find in Section 3 that many logging statements are recording the behaviour or state of the program in blocks. In addition, blocks provide a finer-grand suggestion which may be more actionable compared to coarse-grand suggestions (e.g., method or file level) [61, 62]. We analyze the source code by parsing the abstract syntax tree (AST) of every method in the studied systems. Then, we identify the AST nodes in a method that represent blocks, such as the block nodes that are associated with if, for, and catch. Hence, each method may contain multiple blocks. For the block nodes that we identified, we then label them as either *logged* block or *non-logged* block by analyzing if the block contains at least one logging statement. Specifically, only the block that directly contains a logging statement is labelled as a *logged* block. For example, as shown in Figure 1, block B0 (line 3 - 6) is labelled as a *logged* block because there is a logging statement in line 4. Block B1 is labelled as a *non-logged* block, because the logging statement in line 4 is not directly contained by block B1. Table 1 shows the statistics of blocks in the studied systems. #LB refers to the number of *logged* blocks, #NLB refers to the number of *non-logged* blocks, and %LB is the percentage of *logged* blocks over all the blocks. Note that there might be multiple logging statements in a block, so the number of *logged* block is smaller than the number of logging statements in each system. In general, we find that only a small portion, i.e., 3.2% to 14.8% of the blocks contain logging statements. Hence, **accurately suggesting logging locations at the block level is a challenging task.**

As we find in Section 3, the locations of logging statements may be influenced by either the syntactic, semantic, or both types of information in source code. In order to obtain the features for training deep learning models and to further study the effectiveness of these features in suggesting logging locations, we then extract the syntactic, semantic, and fused block-level features when we are analyzing the source code of each block.

Extracting Block Features. In our manual study, we find that logging statements often have dependencies with the preceding code in the same method. For example, the arguments in the if statement are processed or defined in the prior code (as shown in Section 3). As also found in prior studies [25, 54], developers may insert logging statements based on the execution flow prior to the logging point.

Therefore, for each identified block, we analyze the source code from the start of the method, in which the code block is located, to the end of the block. This could also reflect developers' sequential workflow by suggesting whether or not a block needs a logging statement when developers finish implementing the block [23]. For

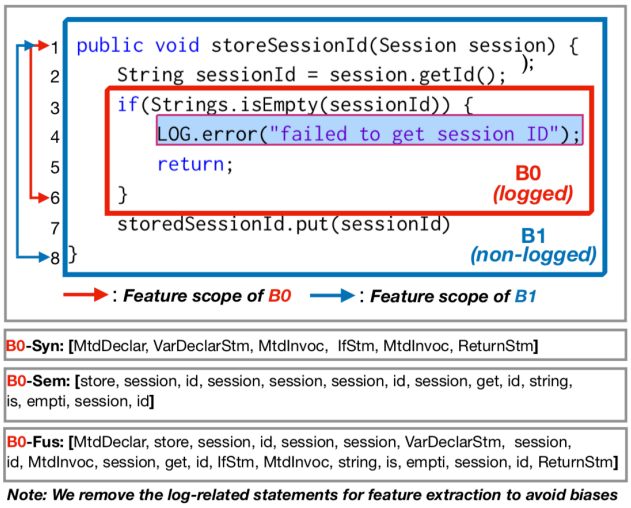


Figure 1: An example of how we label code blocks and extract the tokens for generating the features. We illustrate the tokens extracted from Code Block B0.

example, in Figure 1, for block B0, we consider the code statements from line 1 to line 6. Similarly, for block B1, we consider the code statements from line 1 to line 8. Specifically, for each code block in a studied system, we find all the AST nodes from the start of the method to the end of this block. Then for each AST node, we record its type (e.g., MethodInvocation, VariableDeclaration, or CatchClause), the associated semantic information (e.g., the name of the variable declared in the VariableDeclaration node) as well as the location (i.e., class and method, and the start line and end line). We then extract three types of code block features using the above-mentioned information from these AST nodes.

Below, we discuss the approaches that we use to extract syntactic, semantic, and fused block features, respectively.

Syntactic Block Features: We extract the syntactic features that represent the structural information from the AST nodes in code blocks. We capture the syntactic information by extracting the AST nodes that are related to the control flow of the code. We exclude AST nodes, such as SimpleName (i.e., identifier name) and SimpleType (i.e., identifier type), which do not contain structural information of the code. For each block, we count the occurrence of each AST node in the block and all preceding code in the same method. At the end of the syntactic feature extraction, for each block, we obtain a vector that represents the occurrence of each structural AST node in the block and its preceding code. We call each element in the vector as a token. Figure 1 shows an example of the syntactic features for the B0 block, where we extract the AST nodes from the feature scope.

Semantic Block Features: We extract the semantic features from the textual information inside the code blocks. Prior studies found that information such as variable names may capture the semantic information of the code [16, 17, 33, 78]. Therefore, we process variable names and invoked methods in the block as plain text. For each block, we consider all the semantic information in the block and in the preceding code in the same method. Note that we exclude all reserved keywords in the programming languages, such

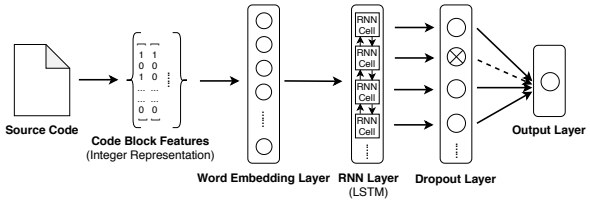


Figure 2: The overall architecture of our approach.

as if, else, and for, to avoid capturing structural information. We follow common source code preprocessing techniques: splitting the words using camel case, converting all words to lower case, and applying stemming [16, 48]. Figure 1 shows an example of the semantic block features of the B0 block.

Fused Block Features: Developers may add logging statements based on both the syntactic and semantic of the code. Therefore, in addition to building separate models using the above-mentioned syntactic and semantic features, we also combine both types of information together (i.e., fused features). To obtain fused features of code blocks, we build an unified corpus containing both syntactic and semantic information of the source code by following a prior study [42]. Specifically, we merge the syntactic and semantic features in a block together, while keeping the original orders of those AST nodes in the source code. Then, for each fused code block, we obtain the vector representation similar to the process discussed in other types of features. Figure 1 shows an example of the fused code block features of the B0 block.

4.2 Deep Learning Framework and Implementation

We formulate the process of suggesting logging locations as a binary classification problem. Given a block, we apply deep learning models to suggest whether or not the block should contain a logging statement. In this subsection, we discuss the overall architecture and implementation of our deep learning model.

Overall Architecture. Figure 2 shows the overall architecture of our approach. We first map our input vectors (i.e., syntactic, semantic, and fused features) through an embedded layer. The embedded layer learns the relationship and similarity among the vectors in each block feature and processes each vector based on integer encoding to probabilistically distributed representations. We then employ a recurrent neural network (RNN) layer to model the relationship between the logging decision of a block and the vectors returned from the embedding layer. Finally, the output layer of our deep learning model is a one-dimension dense layer with the sigmoid activation function to suggest whether a block should be logged or not. Below, we discuss the details of each layer.

Embedding Layer. After extracting the syntactic, semantic, and fused features (i.e., in the forms of vectors, as illustrated in Figure 1), we feed them to the embedding layer. The embedding layer captures the linear relationships among tokens in the input vector, and outputs a set of new vectors, called word embeddings [49, 59]. Compared to simple integer encoding or one-hot encoding which does not consider the relationship among the tokens, word embeddings can learn the similarities among tokens and return probabilistically distributed representations of the words (e.g., run and execute might be similar in vector space).

RNN Layer. Since source code provides instruction on system execution, there are dependencies between consecutive lines of source code. For example, as we discussed in our manual study, the condition variable in *IfStatement* may have dependency on prior source code, because the variable is defined or processed priorly. Hence, we follow prior studies [19, 27, 52, 75] and model source code as sequential data (i.e., we consider the order of the source code tokens in the data). We include a layer of Long Short Term Memory (LSTM) in the deep learning model, which is a variant of RNN that includes a memory cell and gate mechanisms in the recurrent unit to preserve long term dependencies of the code [22, 28, 31, 35].

Output Layer. After the previous layers, the block features are still high-dimensional vectors. In order to make a binary suggestion of whether a block is *logged* or *non-logged*, we use a one-dimensional dense layer with sigmoid activation function as the output layer of our approach. This layer takes all outputs from the previous layer to its unique neuron, then the neuron provides the final suggestion (i.e., *logged* or *non-logged*) of this block.

Implementation and Training We use Keras [2] to implement our deep learning model. For the embedding layer, we adopt Skip-gram from Word2vec [1] and set the dimension to 100 [48] to obtain the word embeddings of each type of the three features separately. For the RNN layer, we set the dimension of hidden states as 128 and attach a dropout layer with a 0.2 dropout rate in order to reduce the potential impact of overfitting and immoderate reliance on the trained system [30, 57, 76]. We train our model for 100 epochs on each studied system and set the batch size to 24. Because there is a noticeable imbalance between the number of *logged* blocks and *non-logged* blocks (overall only 3.2% to 14.8% blocks are *logged* blocks, as shown in Section 4), for each studied system and each type of code block features, we apply stratified random sampling [55] (i.e., ensure the random sample has the same distribution of classes as the original data) to split the block features into training set (60%), validation set (20%) and testing set (20%) [48, 75]. **Note that we remove the log-related statements when we are generating the features to avoid biases in the suggestion results.** Finally, we upsample the *logged* block features in the training set after the splitting process to mitigate the impact of data imbalance [7, 53].

5 EVALUATION

In this section, we evaluate our approach by introducing the evaluation metrics and answering two research questions.

5.1 Evaluation Metrics

Given the features of a code block as inputs, our deep learning model suggests if this block is *logged* or *non-logged*. To evaluate the performance of our model, we use Balanced Accuracy, Precision, Recall, and F-measure as our evaluation metrics.

Balanced Accuracy. Balanced accuracy is widely used by prior studies to evaluate model performance on imbalanced data [41, 78]. It calculates the average of True Positive Rate (i.e., how many suggested *logged* blocks are correct) and True Negative Rate (i.e., how many suggested *non-logged* blocks are correct). Balanced accuracy is computed as:

$$BalancedAccuracy = (\frac{TP}{TP+FN} + \frac{TN}{TN+FP})/2$$

Table 2: The results of suggesting logging locations using syntactic (Syn.), semantic (Sem.), and fused (Fus.) block features.

	Balanced Accuracy				Precision				Recall				F_1			
Systems	Syn.	Sem.	Fus.	RG.	Syn.	Sem.	Fus.	RG.	Syn.	Sem.	Fus.	RG.	Syn.	Sem.	Fus.	RG.
Cassandra	**83.0**	65.8	65.2	49.8	**51.7**	37.1	31.8	3.0	**56.6**	33.7	33.2	3.5	**54.0**	35.3	32.5	3.2
Elasticsearch	**81.9**	67.7	69.9	50.1	**52.0**	29.7	24.3	3.6	**55.6**	38.6	44.7	3.7	**52.9**	33.6	31.5	3.6
Flink	**83.0**	74.2	75.0	50.0	**58.9**	36.0	37.6	5.6	**70.9**	54.4	55.6	8.7	**64.3**	43.3	44.9	6.8
HBase	**80.5**	69.7	72.9	49.9	**56.1**	45.2	43.2	4.8	**63.4**	41.9	49.1	5.0	**59.5**	43.5	45.9	4.9
Kafka	**74.4**	68.3	67.5	50.1	**41.5**	30.8	37.4	9.5	**58.2**	48.5	49.0	11.0	**47.3**	37.7	42.5	10.2
Wicket	**84.7**	76.6	72.2	50.0	**45.7**	28.1	26.9	3.7	**72.3**	58.5	49.2	3.2	**56.0**	37.9	34.8	3.4
Zookeeper	**72.9**	64.6	70.5	49.8	**48.3**	39.6	47.5	12.8	**55.6**	39.2	50.3	16.8	**51.7**	39.4	48.9	14.5
Average	**80.1**	69.6	70.5	50.0	**50.6**	35.2	35.6	6.1	**61.8**	45.0	47.3	7.4	**55.1**	38.7	40.2	6.7

Note: RG. represents the result of the baseline. For each system and for each evaluation metric, the block feature that yields the best performance is marked in bold. All the numbers represent percentage.

where TP, TN, FP and FN refer to True Positive, True Negative, False Positive (i.e., suggested as a *logged* block but is actually a *non-logged* block) and False Negative (i.e., suggested as a *non-logged* block but is actually a *logged* block), respectively. A high balanced accuracy means both the majority class (i.e., *non-logged* block) and minority class (i.e., *logged* block) are accurately suggested.

Precision. In our study, precision represents the ability of our approach to correctly suggest *logged* blocks (i.e., how many *logged* blocks suggested by our model are correct). Specifically, precision is defined as:

$$Precision = \frac{TP}{TP + FP}$$

Note that only positive labels (i.e., *logged* block) are considered for this metric (i.e., the performance on *non-logged* data does not affect our calculation of precision). Hence, a high precision means that most of the suggested *logged* blocks are indeed logged.

Recall. Recall represents the ability of finding *logged* blocks from the data set (i.e., how many *logged* blocks can be suggested by our approach). It is computed as:

$$Recall = \frac{TP}{TP + FN}$$

Same as precision, only positive labels are considered for this metric. A higher recall means that we can identify more code blocks that need to be logged.

F1 Score. F1 score is a metric that considers both precision and recall. It is computed as:

$$F_1 = 2 * (\frac{Precision * Recall}{Precision + Recall})$$

F1 score balances the use of precision and recall and provides a more realistic measure of the performance by using both of them. A high F1 score means that we can both accurately and sufficiently suggest *logged* blocks.

5.2 Case Study Results

In this subsection, we present the results for our research questions (RQs). For each RQ, we describe the motivation, approach, and results and discussions.

RQ1: How effective are different block features when suggesting logging locations?

Motivation. Deciding where to log is a challenging practice [25, 41, 78]. As we find in our manual study, there exist some common characteristics of where developers insert logging statements. Logging location might be related to either the syntactic information, semantic information of the code, or both. In this RQ, we investigate the performance of our deep learning models and how each block feature performs in suggesting logging location. Our finding may help validate our manual study results that there may be an implicit logging guideline that developers follow, and identify the important features in suggesting logging location. Specifically, we split this RQ into three sub-RQs:

RQ1.1: What is the performance of the three block features when suggesting logged blocks?

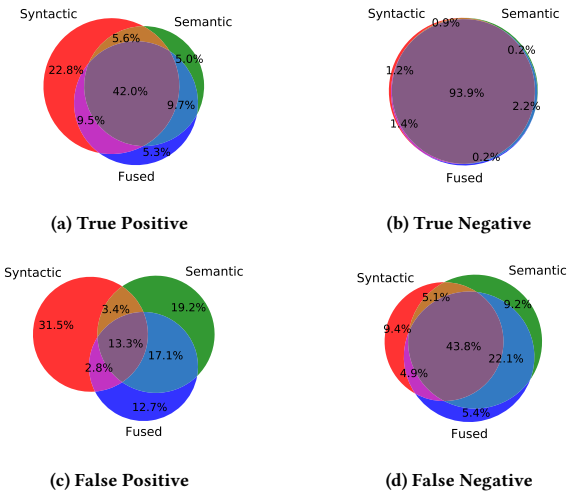
RQ1.2: Do different block features capture different information?

RQ1.3: What are the suggestion accuracies for different categories of *logged* blocks?

Approach. We train our deep learning framework on the training data by following the process discussed in Section 4. We conduct our experiments on the same systems that we used in our manual analysis. For each studied system, we train three models using different types of block features (i.e., syntactic, semantic, and fused). Finally, we evaluate the model performance on the testing set using the above-mentioned evaluation metrics. Note that we pre-determined the training (60%), validation (20%), and testing (20%) data set before extracting the block features. Hence, we use the same set of code blocks for each system when evaluating the syntactic, semantic and fused blocks features.

RQ 1.1: What is the performance of the three block features when suggesting logged blocks? To evaluate the effectiveness of our models, we compare the results of the models trained using three block features with a baseline. Since there is no prior study that suggests logging locations at the block level, we use Random Guess (RG) as our baseline, which is commonly used by prior studies [18, 26, 46, 48, 60, 63, 64]. Given a block in a studied system, Random Guess suggests whether this block should be a *logged* block or *non-logged* block based on the proportion of *logged* block in this system. For example, 10% of the code blocks in Kafka are *logged* block as shown in Table 1. Then, for each code block being tested, there is a 10% chance for Random Guess to suggest it as a *logged* block and a 90% chance to suggest it as an *non-logged* block. We repeat the Random Guess 30 times (as suggested by previous studies [18, 26]) for each system to reduce the biases. We report the average values of the four evaluation metrics computed based on the 30 times of iterations as the result of Random Guess.

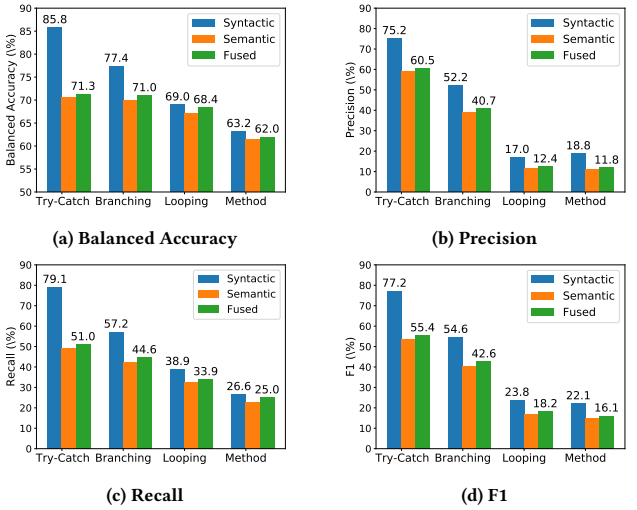
RQ 1.2: Do different code block features capture different information? To further investigate if different block features capture different information in the source code, we examine the overlap and differences of the results generated from the models trained by using three block features. For each type of block feature, we collect the prediction results on the testing data of seven studied systems, analyze the True Positives, True Negatives, False Positives, and False Negatives, and compute the percentage of overlap among the syntactic, semantic and fused block features.



(a) True Positive (b) True Negative

(c) False Positive (d) False Negative

Figure 3: Venn diagrams of TP, TN, FP and FN of the three block features. Each number represents the percentage of the corresponding intersecting set out of the union set.



(a) Balanced Accuracy (b) Precision

(c) Recall (d) F1

Figure 4: The (a) Balanced Accuracy, (b) Precision, (c) Recall, and (d) F1 of the models trained from three block features when applied on different types of blocks.

RQ 1.3: What are the suggestion accuracies for different categories of logged blocks? Since three code block features may capture different information in the source code, they might have varied performance when predicting different categories of *logged* blocks. Hence, we further evaluate the performance of the three block features on the different categories of logging statements (Section 3). We report the suggestion results based on the type of blocks that the logging statement is associated with (i.e., try-catch block, branching block, looping block, and method declaration block).

Results and Discussions.

RQ 1.1. Table 2 presents the results of the models built using the Syntactic (Syn.), Semantic (Sem.) and Fused (Fus.) code block features, and the baseline Random Guess (RG.). Overall, for all the evaluation metrics, models trained by using the block features outperform the baseline. The precision of RG ranges from 3.0% to 12.8%, recall ranges from 3.2% to 16.8%, and the balanced accuracy ranges from 49.8% to 50.1%. Note that RG makes suggestion based on the distribution of training data, the distribution of *logged* and *non-logged* blocks in the testing data is the same as the original data (as shown in Table 1). Therefore, given sufficient trails, the balanced accuracy of RG will be close to 50%. We find that models trained using the syntactic block features have the best performance compared to other block features across all studied systems. In particular, the balanced accuracy of semantic and fused features ranges from 64.6% to 76.6%, while for syntactic feature it is over 72.9% on all the studied systems (with an average of 80.1%). The average precision ranges from 24.3% to 47.5% when using semantic and fused block features, and the average recall ranges from 33.2% to 58.5%. In comparison, the average precision and recall on syntactic feature are 50.6% and 61.8%, respectively. The results show that syntactic information might play an important role in logging decisions and may be leveraged to suggest logging locations.

RQ 1.2. Figure 3 shows the percentage overlap on (a) True Positive, (b) True Negative, (c) False Positive, and (d) False Negative among the models trained using syntactic, semantic, and fused block features. Note that Red, Green, and Blue circle represents the suggestion results of syntactic, semantic, and fused block features, respectively. Each number represents the percentage of the corresponding intersecting data set (e.g., for TP, 42.0% represents the common set of True Positive among Syn., Sem. and Fus.) out of the entire set (e.g., for TP, the entire data set is the set that combines the TP from syntactic, semantic and fused altogether across all studied systems). There is a 42.0% overlap in TP among the three features, while syntactic covers most of the TPs (79.9% out of all the TPs) compared to semantic (62.3%) and fused (66.5%) block features. Only 20.1% of the TPs are missed by syntactic but captured by two other block features. For TNs (i.e., correctly suggest as *non-logged* block), almost all (93.9%) are overlapping among the three block features. The results show that there is a high level of agreement among the models when suggesting the *non-logged* blocks. For FPs, there is no considerable overlap among the three features (13.3%). For FNs, syntactic has the lowest number of FNs (63.2% of the FNs are covered by syntactic, compared to 80.2% covered by semantic and 76.2% covered by combined feature). The results show that different block features might capture different information from source code. As semantic and fused block features still capture TPs that are missed by syntactic block feature (20.1%), future work could further investigate how to better combine the two sources of information to provide a sufficient and accurate suggestion. Moreover, we manually investigate a sample of FPs and FNs. We identify their characteristics and find that many of them are not indeed FPs and FNs (details in Section 6).

RQ 1.3. Figure 4 shows the (a) Balanced Accuracy, (b) Precision, (c) Recall, and (d) F1 of the models when suggesting on different types of blocks associated with the categories in Section 3. Overall, the three block features have a similar trend for the results on different types of blocks. Syntactic features have the best results for all types of blocks on all the evaluation metrics. Among the four types of

Table 3: The results of cross-system logging locations suggestion using syntactic block features.

	Balanced Accuracy			Precision			Recall			F_1			Fleiss' Kappa	
Systems	Within	Cross	Ratio	Within	Cross	Ratio	Within	Cross	Ratio	Within	Cross	Ratio	logged	non-logged
Cassandra	83.0	67.8 (σ3.0)	81.7	51.7	37.5 (σ9.1)	72.6	56.6	41.9 (σ7.8)	74.1	54.0	39.1 (σ7.3)	72.5	0.47 (*Mod.*)	0.90 (*Sub.*)
Elasticsearch	81.9	65.5 (σ4.5)	80.0	52.0	36.2 (σ11.5)	69.7	55.6	42.0 (σ5.2)	75.6	52.9	37.8 (σ7.0)	71.5	0.45 (*Mod.*)	0.90 (*Sub.*)
Flink	83.0	70.2 (σ3.0)	84.6	58.9	30.5 (σ8.8)	51.8	70.9	49.2 (σ9.1)	69.4	64.3	36.7 (σ7.6)	57.1	0.46 (*Mod.*)	0.91 (*Sub.*)
HBase	80.5	67.5 (σ2.3)	83.9	56.1	37.5 (σ4.8)	66.9	63.4	41.8 (σ6.7)	66.0	59.5	40.5 (σ4.6)	68.1	0.49 (*Mod.*)	0.92 (*Sub.*)
Kafka	74.4	65.7 (σ4.1)	88.4	41.5	32.0 (σ4.2)	77.2	58.2	42.5 (σ6.8)	73.1	47.3	36.2 (σ3.9)	76.6	0.42 (*Mod.*)	0.88 (*Sub.*)
Wicket	84.7	67.8 (σ3.3)	80.1	45.7	40.3 (σ5.2)	88.2	72.3	42.1 (σ8.0)	58.3	56.0	40.8 (σ4.7)	72.9	0.43 (*Mod.*)	0.85 (*Sub.*)
Zookeeper	72.9	66.8 (σ2.7)	91.7	48.3	33.6 (σ6.2)	69.6	55.6	44.8 (σ5.6)	80.6	51.7	38.3 (σ5.8)	74.1	0.37 (*Fair*)	0.81 (*Sub.*)
Average	80.1	67.3	84.0	50.6	35.4	70.0	61.8	43.5	70.4	55.1	38.6	70.0	0.44 (*Mod.*)	0.88 (*Sub.*)

Note: Within shows the results of within-system suggestion. Cross shows the average results and the standard deviation (σ) when applying the models trained using other systems. Ratio shows the percentage of Cross over Within. Fleiss' Kappa shows the degree of agreement on the suggestion result of the cross-system models on *logged* and *non-logged* blocks. *Mod.* and *Sub.* represent moderate and substantial agreement [37], respectively.

blocks, logging statements associated with try-catch blocks have the best results on all the evaluation metrics (85.8% balanced accuracy, 75.2% precision, 79.1% recall and 77.2% F1 for syntactic). As also found in prior studies [25, 78], logging statements in such blocks may be better defined. We also find that logging statements associated with branching blocks have a good overall suggestion result. In contrast, the results of suggesting logging statements associated with looping and method declaration blocks are relatively lower (balanced accuracy ranges from 63.2% to 69.0%, and F1 ranges from 22.1% to 23.8%). Although the three block features have a similar trend of results on different types of blocks, syntactic features are better than the other two for suggesting logging locations on all the studied types of blocks. Moreover, our study shows that there is a clearer pattern of inserting logging statements in try-catch and branching blocks (i.e., higher precision and recall). Practitioners may prioritize reviewing and deciding the given logging suggestions in such blocks. In addition, future research may investigate other sources of information in order to better assist in making logging decisions for looping and method declaration blocks.

> All the trained models noticeably outperform the baseline. Among the three types of block features, models trained using syntactic block features achieve the best results on all the evaluation metrics. The results show that syntactic information might be leveraged to suggest logging locations.

RQ2: Are the trained models transferable to other systems?

Motivation. When working on a new system, developers may encounter difficulties when deciding logging locations. Different from matured systems with a long period of development and maintenance history, developers working on new systems may not have sufficient knowledge on deciding where to log. Therefore, in this RQ, we investigate whether different systems share similar implicit guidelines of logging locations. Our findings may provide evidence on the existence of common logging characteristics across systems and help future research derive a universal logging guideline. In particular, we study two sub-RQs:

RQ2.1: What is the effectiveness of cross-system logging suggestion?

RQ2.2: What is the level of suggestion agreement on cross-system models?

Approach. In this RQ, we study if *logged* blocks share similar syntactic block features by doing a cross-system transferable learning. Namely, we study if a model that is trained using the syntactic features from one system can be used to suggest logging location in another system. We choose to study syntactic block features because they are extracted from the AST nodes in the source code, which are common across all Java systems, and they have the best performance compared to the other two block features as shown in RQ1. Moreover, since the syntactic block features capture the underlying code structure [33], a high cross-system suggestion accuracy may show the potential of deriving a logging guideline based on code structure in future studies.

RQ 2.1: What is the effectiveness of cross-system logging suggestion? For each studied system, we build a model using the syntactic block features and apply the model on each of the other systems. For example, we train a model using the syntactic block features in Cassandra, and apply the model on six other studied systems. Finally, we compute and report the average balanced accuracy, precision, recall, and F-measure of the cross-system logging suggestion.

RQ 2.2: What is the level of suggestion agreement on cross-system models? To study whether the models trained using different systems capture similar information (i.e., the relationship between the syntactic features and logging location), we analyze the agreement level of cross-system suggestion results. We separately examine the suggestion agreement of the cross-system models on *logged* blocks and *non-logged* blocks. Namely, for each studied system, we apply the models trained using other systems, and study the suggestion results of the cross-system models on the *true* logged blocks and the *true* non-logged blocks, respectively. In particular, we compute Fleiss's Kappa to study the agreement among the suggestion results from cross-system models [24]. Fleiss's Kappa computes the inter-rater agreement among a fixed set of raters (i.e., suggestion results from different cross-system models). A higher level of agreement may show that the syntactic block features have very similar relationships with logged or *non-logged* blocks across all studied systems.

Results and Discussions.

RQ 2.1. Table 3 shows the results of our cross-system suggestions using syntactic block features. In general, we find that the results of cross-system suggestions are lower than within-system suggestions using syntactic block features. However, the results are still comparable to within-system suggestions using semantics and fused block features. For balanced accuracy, the cross-system suggestions achieve over 80% (i.e., Ratio column in Table 3) of the corresponding within-system suggestion using syntactic block features. On average, the balanced accuracy ranges from 65.5% to 70.2%, with standard deviations range from 2.3 to 4.5. For precision, recall, and F_1, the cross-system suggestions achieve 51.8% to 88.2% ratio of the within-system suggestion results. In short, even though we find that the results of cross-system suggestion are slightly lower than those of within-system, we may still achieve a reasonable performance.

Similar to RQ1, we also manually study a sample of FPs and FNs from the results of RQ2 (details in Section 6).

RQ 2.2. Table 3 also shows the Fleiss's Kappa [24] for each studied system. For *logged* blocks, the agreements are moderate in six studied systems. The agreement level is fair in Zookeeper, but the value is also close to the threshold of a moderate agreement (i.e., 0.41 [24]). Our results show that the models trained using the syntactic block features may share certain underlying properties. Namely, there are some commonalities in the code structure on how developers decide logging locations across the studied systems. For *non-logged* blocks, the agreements are substantial across all studied systems. In short, our findings show that developers are rather consistent on deciding which blocks *do not* need logging statements. Although there are some inconsistencies across the studied systems, we may still apply cross-system models to help suggest logging locations in other systems.

> We find that cross-system logging location suggestion achieves a reasonable performance compared to within-system suggestion (i.e., 84% of the within-system balanced accuracy). We also find that the cross-system models have moderate agreements on *logged* blocks and substantial agreements on *non-logged* blocks. Our results show that developers in different systems may follow certain implicit guidelines on deciding logging locations.

6 DISCUSSION

As shown in the RQs, our models can provide promising results of suggesting logging locations. To further inspire future studies and better assist practitioners, we conduct a manual study to understand the FPs and FNs in the suggestion results. For each studied system in RQ1 and for each of the three models (i.e., Syntactic, Semantic, and Fused, simplified as Syn., Sem. and Fus.), we select the top-five FPs and FNs for our manual study, ranked by their suggested probabilities of being *logged* and *non-logged*, respectively (a total of 105 FPs and 105 FNs). For the cross-system models in RQ2, we also select the top-five FPs and FNs from each system (a total of 35 FPs and 35 FNs).

False Positives. For Syn. in RQ1, we find that 25/35 of the studied FPs are actually TP. The code block either contains some other types of print statements to record the execution information (e.g., System.out.print()), or contains only one child block and has no other code statements, and the child block contains a logging statement. For Sem., Fus., and cross-system models, we also find 15/35, 16/35, and 18/35 cases that belong to this category, respectively. For the remaining studied FPs, the suggestions are made when the code block is at the beginning of a method (9 cases for Syn., 12 for Sem., 10 for Fus., 16 for cross-system models), or in complex code with multiple nested blocks (1 case for Syn., 8 for Sem., 9 for Fus., 1 for cross-system models).

False Negatives. We find that 15/35 of the studied Syn. FNs may not truly be FN. Similar to the situation in FP that a code block only contains a *logged* child block and has no other code statements, the child block is suggested as a *non-logged* block and thus becomes an FN. For Sem., Fused. and cross-system models, we find 7/35, 6/35, and 15/35 cases that belong to this category. We also find that for 4/35, 3/35, 3/35 and 5/35 of the studied FNs from Syn., Sem., Fus. and cross-system models, they are blocks that have many very similar sibling blocks nearby (e.g., many similar if blocks having similar structures), while only the FN cases here contain logging statements. For the remaining studied FNs, similar to what we find in FPs, they locate at the beginning of a method (15 cases for Syn, 18 for Sem., 14 for Fus., 14 for cross-system models), or in complicated code structure with multiple nested code blocks (1 case for Syn., 7 for Sem., 12 for Fus., 1 for cross-system models).

Our findings show that the actual performance of our model may be even better due to the diverse nature of how developers write logging code. We also find that it may be more difficult to suggest a logging statement at the beginning of a method due to the lack of prior information in the code block.

7 THREATS TO VALIDITY

Construct Validity. Our approach presumes that the training data has high-quality source code and follow good logging practice. However, there exist no industrial standards guiding developers to write logging statements. In this paper, we choose seven large-scale, well-maintained systems with different sizes, across various domains to conduct the study. They are commonly used in prior log-related studies and are considered as following good logging practice [10, 11, 38, 46]. We evaluate our models on the test data set of each studied system. Different test data set might lead to very different results. To mitigate the fluctuation caused by different test data set, we apply stratified random sampling by following prior studies [48, 55, 75] to split the data set and ensure each randomly sampled data set has the same distribution of labels as the original data.

Internal Validity. We conduct manual studies to investigate the characteristics and uncover the categories of logging locations. To avoid biases, the authors examine the data independently. For most of the cases, the authors reach an agreement. Any disagreement is discussed until a consensus is reached with a substantial-level agreement (Cohen's Kappa 0.86) [56]. Involving third-party logging experts to verify our results might further reduce this threat. Different parameters used in the neural networks might affect the effectiveness of the trained models. We follow prior studies [48, 75] to set the parameters for our deep learning models. The models trained using our approach might not be optimal on some of the evaluation metrics (e.g., an average F1 score of 66.7 on syntactic code block features). Future study may further improve the performance of our approach and provide a more comprehensive perspective of the suggestion results by surveying software engineering practitioners. We use word embeddings [49, 59], which is widely used by prior studies [48, 75] as the distributed representations of source code. Future study may consider other code representation approaches (e.g., code2vec [6, 34]) to examine the performance on suggesting logging locations.

External Validity. We conducted our study only on seven large-scale open source systems. However, we selected the studied systems in various domains and sizes (from 97K to 1.5M LOC as shown in Table 1) in order to improve the representativeness of our studied systems. Our studied systems are all implemented in Java. The results and models may not be transferable to systems in other

programming languages. Future studies should validate the generalizability of our findings and the transferability of our models in systems that are implemented written in other programming languages.

8 CONCLUSION

In this paper, we aim to tackle the challenges that developers might encounter when deciding logging locations by first conducting a comprehensive manual study. We uncover six categories of logging locations and find that developers usually insert logging statements to record execution information that happens in various types of code blocks. We propose a deep learning based approach to provide finer-grained (i.e., at the code block level) suggestions on logging locations. Our approach achieves promising results on suggesting logging locations in both within-project and cross-project predictions. Our results highlight the potential of providing finer-grained suggestions on logging locations by leveraging syntactic information in the source code, and such suggestions may be shared across systems. Future studies could explore a more advanced way of combining syntactic and semantic information in the source code, in order to provide better suggestions on logging locations.

REFERENCES

[1] [n.d.]. gensim Word2vec embeddings. https://radimrehurek.com/gensim/models/word2vec.html. Last checked Feb. 2020.

[2] [n.d.]. Keras: The Python Deep Learning library. https://keras.io/. Last checked Feb. 2020.

[3] [n.d.]. Log4j. http://logging.apache.org/log4j/2.x/.

[4] [n.d.]. Oracle Java Documentation. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html. Last checked Mar. 2020.

[5] [n.d.]. Simple Logging Facade for Java (SLF4J). http://www.slf4j.org. Last checked Feb. 2018.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29.

[7] Harald Altinger, Steffen Herbold, Friederike Schneemann, Jens Grabowski, and Franz Wotawa. 2017. Performance tuning for automotive Software Fault Prediction. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*. 526–530.

[8] S. Boslaugh and P.A. Watters. 2008. *Statistics in a Nutshell: A Desktop Quick Reference*. O'Reilly Media.

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

[10] Boyuan Chen and Zhen Ming (Jack) Jiang. 2017. Characterizing and Detecting Anti-patterns in the Logging Code. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. 71–81.

[11] Boyuan Chen and Zhen Ming (Jack) Jiang. 2019. Extracting and studying the Logging-Code-Issue- Introducing changes in Java-based large-scale open source software systems. *Empirical Software Engineering* 24, 4 (01 Aug 2019), 2285–2322.

[12] Boyuan Chen and Zhen Ming (Jack) Jiang. 2020. Studying the Use of Java Logging Utilities in the Wild. In *Proceedings of the 42nd International Conference on Software Engineering, (ICSE 2020)*.

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

[14] Tse-Hsun Chen, Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2016. CacheOptimizer: Helping Developers Configure Caching Frameworks for Hibernate-based Database-centric Web Applications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. 666–677.

[15] Tse-Hsun Chen, Mark D. Syer, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2017. Analytics-driven Load Testing: An Industrial Experience Report on Load Testing of Large-scale Systems. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '17)*. 243–252.

[16] Tse-Hsun Chen, Stephen W. Thomas, and Ahmed E. Hassan. 2016. A Survey on the Use of Topic Models when Mining Software Repositories. *Empirical Software Engineering* 21, 5 (2016), 1843–1919.

[17] Tse-Hsun Chen, S. W. Thomas, Meiyappan Nagappan, and A.E. Hassan. 2012. Explaining Software Defects Using Topic Models. In *Proceedings of the 9th Working Conference on Mining Software Repositories (MSR '12)*.

[18] Tse-Hsun Chen, Shang Weiyi, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting Performance Anti-patterns for Applications Developed using Object-Relational Mapping. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. 1001–1012.

[19] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *CoRR* abs/1901.01808 (2019).

[20] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. 2015. Log2: A Cost-Aware Logging Mechanism for Performance Diagnosis. In *2015 USENIX Annual Technical Conference, USENIX ATC '15,*. 139–150.

[21] Zishuo Ding, Jinfu Chen, and Weiyi Shang. 2020. Towards the Use of the Readily Available Tests from the Release Pipeline as Performance Tests. In *Proceedings of the 42nd International Conference on Software Engineering, (ICSE 2020)*.

[22] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. 2019. A Quantitative Analysis Framework for Recurrent Neural Network. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*. 1062–1065.

[23] Ekwa Duala-Ekoko and Martin P. Robillard. 2007. Tracking Code Clones in Evolving Software. In *29th International Conference on Software Engineering (ICSE 2007)*. 158–167.

[24] Joseph L. Fleiss. 1971. Measuring Nominal Scale Agreement among Many Raters. *Psychological Bulletin* 76, 5 (1971), 378–382.

[25] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where Do Developers Log? An Empirical Study on Logging Practices in Industry. In *Proceedings of the 36th International Conference on Software Engineering (ICSE-SEIP '14)*. 24–33.

[26] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*. 57–76.

[27] X. Gu, H. Zhang, and S. Kim. 2018. Deep Code Search. In *2018 IEEE/ACM 40th International Conference on Software Engineering, ICSE 2018*. 933–944.

[28] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. 2019. An Empirical Study Towards Characterizing Deep Learning Development and Deployment Across Different Frameworks and Platforms. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*. 810–822.

[29] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. 2018. Studying and Detecting Log-Related Issues. *Empirical Software Engineering* (2018).

[30] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019*. 34–45.

[31] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[32] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018*. 200–210.

[33] Yuan Huang, Xinyu Hu, Nan Jia, Xiangping Chen, Yingfei Xiong, and Zibin Zheng. 2020. Learning Code Context Information to Predict Comment Locations. *IEEE Trans. Reliability* 69, 1 (2020), 88–105.

[34] Hong Jin Kang, Tegawendé F. Bissyandé, and David Lo. 2019. Assessing the Generalizability of Code2vec Token Embeddings. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*. 1–12.

[35] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A Neural Approach to Decompiled Identifier Naming. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*. 628–639.

[36] Maxime Lamothe and Weiyi Shang. 2020. When APIs are Intentionally Bypassed: An Exploratory Study of API Workarounds. In *Proceedings of the 42nd International Conference on Software Engineering, (ICSE 2020)*.

[37] J. R. Landis and G. G. Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* 33 (1977), 159–174.

[38] Heng Li, Tse-Hsun (Peter) Chen, Weiyi Shang, and Ahmed E. Hassan. 2018. Studying software logging using topic models. *Empirical Software Engineering* (Jan 2018).

[39] Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E. Hassan. 2020. A Qualitative Study of the Benefits and Costs of Logging from Developers' Perspectives. *IEEE Transactions on Software Engineering* (2020), 1–17.

[40] Heng Li, Weiyi Shang, and Ahmed E. Hassan. 2017. Which log level should developers choose for a new logging statement? *Empirical Software Engineering* 22, 4 (Aug 2017), 1684–1716.

[41] Heng Li, Weiyi Shang, Ying Zou, and Ahmed E. Hassan. 2017. Towards just-in-time suggestions for log changes. *Empirical Software Engineering* 22, 4 (2017),

1831–1865.

[42] Xiaochen Li, He Jiang, Yasutaka Kamei, and Xin Chen. 2020. Bridging Semantic Gaps between Natural Languages and APIs with Word Embedding. *IEEE Transactions on Software Engineering* (2020).

[43] Zhenhao Li. 2019. Characterizing and detecting duplicate logging code smells. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019*. 147–149.

[44] Zhenhao Li. 2020. Studying and Suggesting Logging Locations in Code Blocks. In *Proceedings of the 42nd International Conference on Software Engineering: Companion Proceedings, ICSE 2020*.

[45] Zhenhao Li. 2020. Towards Providing Automated Supports to Developers on Writing Logging Statements. In *Proceedings of the 42nd International Conference on Software Engineering: Companion Proceedings, ICSE 2020*.

[46] Zhenhao Li, Tse-Hsun (Peter) Chen, Jinqiu Yang, and Weiyi Shang. 2019. DLFinder: characterizing and detecting duplicate logging code smells. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*. 152–163.

[47] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log Clustering Based Problem Identification for Online Service Systems. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. 102–111.

[48] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li. 2019. Which Variables Should I Log? *IEEE Transactions on Software Engineering* (2019). Early Access.

[49] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013*.

[50] Meiyappan Nagappan, Kesheng Wu, and Mladen A. Vouk. 2009. Efficiently extracting operational profiles from execution logs using suffix arrays. In *ISSRE'09: Proceedings of the 20th IEEE International Conference on Software Reliability Engineering*. IEEE Press, 41–50.

[51] Karthik Nagaraj, Charles Edwin Killian, and Jennifer Neville. 2012. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*. 353–366.

[52] Bui D. Q. Nghi, Yijun Yu, and Lingxiao Jiang. 2019. Bilateral Dependency Neural Networks for Cross-Language Algorithm Classification. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019*. 422–433.

[53] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. 2009. Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009*. 87–102.

[54] He Pinjia, Zhuangbin Chen, Shilin He, and Michael R. Lyu. 2018. Characterizing the Natural Language Descriptions in Software Logging Statements. In *Proceedings of the 33rd IEEE international conference on Automated software engineering*. 1–11.

[55] Heidar Pirzadeh, Sara Shanian, Abdelwahab Hamou-Lhadj, and Ali Mehrabian. 2011. The Concept of Stratified Sampling of Execution Traces. In *The 19th IEEE International Conference on Program Comprehension, ICPC 2011*. 225–226.

[56] Julilus Sim and Chris C. Wright. 2005. The Kappa Statistic in Reliability Studies: Use, Interpretation, and Sample Size Requirements. *Physical Therapy* 85, 3 (March 2005), 257–268.

[57] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* 15, 1 (2014), 1929–1958.

[58] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep Learning Similarities from Different Representations of Source Code. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR 2018)*. 542–553.

[59] Peter D. Turney and Patrick Pantel. 2010. From Frequency to Meaning: Vector Space Models of Semantics. *J. Artif. Intell. Res.* 37 (2010), 141–188.

[60] Harold Valdivia Garcia and Emad Shihab. 2014. Characterizing and Predicting Blocking Bugs in Open Source Projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. 72–81.

[61] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*. 262–273.

[62] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019*. 326–337.

[63] Xin Xia, David Lo, Emad Shihab, Xinyu Wang, and Xiaohu Yang. 2015. ELBlocker: Predicting blocking bugs with ensemble imbalance learning. *Information & Software Technology* 61 (2015), 93–106.

[64] Xin Xia, Emad Shihab, Yasutaka Kamei, David Lo, and Xinyu Wang. 2016. Predicting Crashing Releases of Mobile Applications. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2016*. 29:1–29:10.

[65] Xiaofei Xie, Bihuan Chen, Liang Zou, Yang Liu, Wei Le, and Xiaohong Li. 2019. Automatic Loop Summarization via Path Dependency Analysis. *IEEE Trans. Software Eng.* 45, 6 (2019), 537–557.

[66] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. 2020. Log4Perf: suggesting and updating logging locations for web-based systems performance monitoring. *Empirical Software Engineering* 25, 1 (2020).

[67] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Steve Sporea, Andrei Toma, and Sarah Sajedi. 2018. Log4Perf: Suggesting Logging Locations for Web-based Systems' Performance Monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18)*. 21–30.

[68] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. 249–265.

[69] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 143–154.

[70] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *ICSE 2012: Proceedings of the 2012 International Conference on Software Engineering* (Zurich, Switzerland). IEEE Press, Piscataway, NJ, USA, 102–112.

[71] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2011. Improving software diagnosability via log enhancement. In *ASPLOS '11: Proceedings of the 16th international conference on Architectural support for programming languages and operating systems*. ACM, 3–14.

[72] Yi Zeng, Jinfu Chen, Weiyi iShang, and Tse-Hsun (Peter) Chen. 2019. Studying the characteristics of logging practices in mobile apps: a case study on F-Droid. *Empirical Software Engineering* (2019), 1–41.

[73] H. Zhang, S. Wang, T. Chen, and A. E. Hassan. 2019. Reading Answers on Stack Overflow: Not Enough! *IEEE Transactions on Software Engineering* (2019).

[74] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*. 783–794.

[75] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*. 783–794.

[76] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael R. Lyu, and Miryung Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019*. 104–115.

[77] Chen Zhi, Jianwei Yin, Shuiguang Deng, Maoxin Ye, Min Fu, and Tao Xie. 2019. An Exploratory Study of Logging Configuration Practice in Java. In *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*. 459–469.

[78] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2015. Learning to Log: Helping Developers Make Informed Logging Decisions. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*. 415–425.