TELL: Log Level Suggestions via Modeling Multi-level Code Block Information

Jiahao Liu
National University of Singapore
Singapore
jiahao99@comp.nus.edu.sg

Jun Zeng
National University of Singapore
Singapore
junzeng@comp.nus.edu.sg

Xiang Wang*
University of Science and Technology of China, China
xiangwang@ustc.edu.cn

Kaihang Ji
National University of Singapore
Singapore
kaihang@comp.nus.edu.sg

Zhenkai Liang*
National University of Singapore
Singapore
liangzk@comp.nus.edu.sg

ABSTRACT

Developers insert logging statements into source code to monitor system execution, which forms the basis for software debugging and maintenance. For distinguishing diverse runtime information, each software log is assigned with a separate verbosity level (e.g., *trace* and *error*). However, choosing an appropriate verbosity level is a challenging and error-prone task due to the lack of specifications for log level usages. Prior solutions aim to suggest log levels based on the code block in which a logging statement resides (i.e., intra-block features). Such suggestions, however, do not consider information from surrounding blocks (i.e., inter-block features), which also plays an important role in revealing logging characteristics.

To address this issue, we combine multiple levels of code block information (i.e., intra-block and inter-block features) into a joint graph structure called *Flow of Abstract Syntax Tree* (FAST). To explicitly exploit multi-level block features, we design a new neural architecture, *Hierarchical Block Graph Network* (HBGN), on the FAST. In particular, it leverages graph neural networks to encode both the intra-block and inter-block features into code block representations and guide log level suggestions. We implement a prototype system, TELL, and evaluate its effectiveness on nine large-scale software systems. Experimental results showcase TELL's advantage in predicting log levels over the state-of-the-art approaches.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Maintaining software**.

KEYWORDS

Log Level Suggestion, Multi-level Code Block Information, Graph Neural Network

*Corresponding authors.

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07.
https://doi.org/10.1145/3533767.3534379

ACM Reference Format:
Jiahao Liu, Jun Zeng, Xiang Wang, Kaihang Ji, and Zhenkai Liang. 2022. TELL: Log Level Suggestions via Modeling Multi-level Code Block Information. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534379

1 INTRODUCTION

As modern software systems become increasingly complex, explaining the nature of runtime bugs or even simple misconfigurations is prohibitively difficult. A common practice is to perform software execution logging for various maintenance tasks, such as testing [8, 9, 25], debugging [11, 49, 55], and program comprehension [34, 35]. Towards this end, developers insert logging statements into programs to monitor system execution. However, as the volume of software logs is always overwhelming, log-based program analysis typically requires tedious manual labor, which undermines its applications in practice.

To address this problem, developers manually assign verbosity levels (i.e., *trace*, *debug*, *info*, *warn*, *error*, and *fatal*) to individual logging statements for ranking their importance and severity in investigation. For example, the logging statement *LOG.warn("Couldn't find the leader with id = " + id)* is set at the level of *warn*, indicating that this execution status is unexpected but not causing critical issues, e.g., software crashes. Moreover, log levels provide the capability to filter information noises in software logs and allow developers to identify system failures through simple keyword searches, e.g., *error* and *fatal*. Therefore, choosing appropriate log levels is of crucial importance because they greatly influence the quality of underlying logging statements. In the above example, a higher log level (e.g., *fatal*) would misguide analysts to a false root cause in postmortem program analysis. In comparison, a lower log level (e.g., *trace*) would crowd out the runtime execution of interest in the noise of extensive trivial logs.

Unfortunately, manually deciding log levels is a challenging and error-prone task [22]. Developers often choose suboptimal levels for logging statements in the first place due to factors such as the developer's subjective interpretation of logging principles or just plain human mistakes. As such, developers typically maintain logging statements in a trials-and-errors manner [21]. According to a recent survey, a large portion (specifically, 42.5%) of log-related issue reports are associated with log levels [27].

```
1  protected QuorumServer findLeader() {                    Block: findLeader Start
2      QuorumServer leaderServer = null;
3      Vote current = self.getCurrentVote();
4      for(QuorumServer s : self.getView().values()) {
5          if(s.id == current.getId()) {
6              s.recreateSocketAddresses();
7              leaderServer = s;
8              break;
9          }
10     }
11     if(leaderServer == null) {
12         LOG.warn("Couldn't find the leader with id = "          Logged Block: warn
                                    + current.getId());
13     }
14     return leaderServer;
15 }
```

(a) Code fragment with warn logging statement.

```
1  public String getPeersMatching(ServerState state) {
2      StringBuilder hosts = new StringBuilder();
3      for (QuorumPeer p : getPeerList()) {
4          if (p.getPeerState() == state) {
5              hosts.append(String.format("%s:%d,", LOCALADDR,
               p.getClientAddress().getPort()));
6          }
7      }
8      LOG.info("getPeersMatching ports are {}", hosts);     Logged Block: info
9      return hosts.toString();
10 }
```

(b) Code fragment with info logging statement.

Figure 1: Two code fragments with logging statements at different verbosity levels in Zookeeper.

Recently, researchers have started to tackle the problem of automatic log level suggestions [3, 22, 27]. These efforts propose to predict levels for new logging statements based on logging principles learned from existing software logs. In particular, prior approaches focus on logging characteristics from the blocks in which logging statements reside (i.e., *intra-block information*). However, log levels also depend on program behaviors in surrounding blocks [10] (i.e., *inter-block information*). For example, even though two logging statements have identical intra-block features, they are likely to be assigned with different levels if they *flow* into distinct contexts. Figure 1 shows two logging statements from the same open-source software system, Zookeeper. Despite sharing similar intra-block information, they are given different log levels (*warn* and *info*) due to different neighboring blocks. As such, we wish to incorporate *inter-block information* as auxiliary supervision to help interpret the rationale behind logging statements. Unfortunately, this has been largely overlooked by existing solutions.

In this paper, we aim to leverage multi-level block information (i.e., intra-block and inter-block features) in a cooperative fashion to predict log levels. Abstract syntax tree (AST) and inter-procedure control flow graph (ICFG) are fundamental code abstractions in program analysis. To be specific, AST is a specialized tree that preserves syntactic structures of code blocks [4], while ICFG is a graph notation that extracts control flows among code blocks. Therefore, we use the AST and ICFG as the basis of intra-block and inter-block features. To further exploit multi-level block information cooperatively, we combine the AST and ICFG into a joint graph structure named *Flow of AST (FAST)*, where nodes in the graph denote code blocks attributed with their ASTs, and edges represent control flow transfers. Thereafter, we develop a new neural architecture named *Hierarchical Block Graph Network (HBGN)* that hierarchically distills useful signals in the FAST to predict log levels. More specifically, HBGN leverages the information propagation and aggregation mechanisms in graph neural networks [19] to explicitly model multiple levels of code block information towards more effective log level suggestions.

We implement a prototype system, TELL (**Te**ll **L**og **L**evels), which exploits multi-level block information and performs post-facto log level suggestions for code blocks with logging statements in an end-to-end manner. We evaluate TELL on nine large-scale software systems and compare it with the state-of-the-art approaches, e.g., DeepLV [27]. Experimental results show that: 1) TELL significantly improves the accuracy in log level suggestions from 61.1% to 71.0%; 2) intra-block and inter-block information cooperatively contribute to characterizing log levels; 3) different software systems share implicit guidelines for log level usages.

In summary, we make the following contributions:

- We are the first to incorporate multi-level block information into log level characterization. We propose a joint code graph, Flow of AST, to encode intra-block and inter-block features.
- We design a novel neural architecture, Hierarchical Block Graph Network, upon the FAST, which explicitly exploits multiple levels of block information to make suggestions on log levels.
- We implement TELL and evaluate it on nine real-world systems. Results show that TELL predicts log levels with high accuracy, significantly outperforming state-of-the-art approaches.

The rest of this paper is organized as follows. Section 2 introduces the background and motivation of our study. Section 3 describes our approach with full details. Section 4 provides the evaluation metrics and results. Section 5 discusses the threats to validity. Section 6 summarizes the related work. Finally, Section 7 concludes the paper.

2 BACKGROUND & MOTIVATION

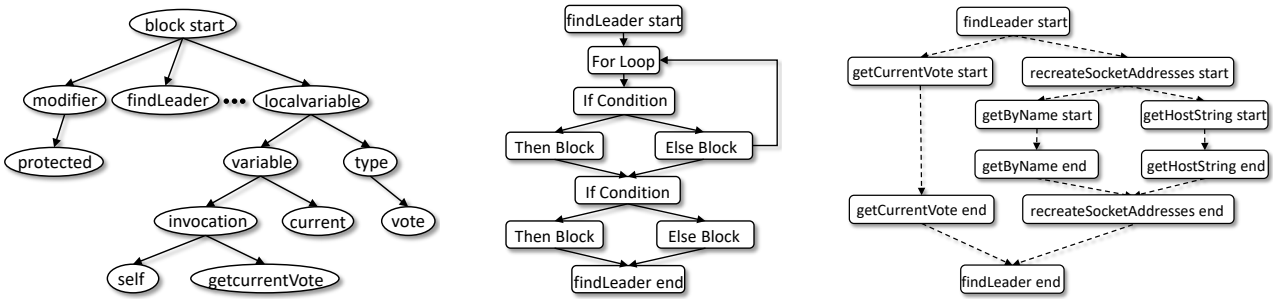
In this section, we use a running example to introduce the problem of log level suggestions. Then, we analyze the limitations in prior work to motivate our insight.

2.1 Running Example

Figure 1a presents a code fragment in Zookeeper [1] that contains a *warn* logging statement. The member function `findLeader` attempts to locate the leader node in a quorum by comparing node IDs, where a log at the level of *warn* will be created if the leader cannot be found. This example shows a typical case where developers apply verbosity levels to distinguish system running information of different importance. In particular, the *warn* logging statement implies that "an unexpected incident occurs, but the application carries on execution". If this statement is mistakenly assigned a lower level (e.g., *info*), this incident would be buried in the noise of trivial logs. On the contrary, a severer level (e.g., *fatal*) would guide developers to a false system failure, despite the fact that this incident has been appropriately handled by the remaining code. Therefore, suggesting suitable levels for logging statements is essential during software development and maintenance.

Log Level Suggestions. We formulate the problem of suggesting log levels as classification — *given a logging statement in source code, we aim to automatically classify it into a verbosity level.* For example, suppose developers have not decided on a level for the logging statement on line 12 in Figure 1a. Then, we suggest a suitable level (e.g., *warn*) based on the source code around the statement.

[1] https://github.com/apache/zookeeper/releases/tag/release-3.5.6



(a) Abstract Syntax Tree (`findLeader start`) (b) Control Flow Graph (c) Call Graph

Figure 2: Different representations of source code for the running example.

2.2 Representations of Code

Multiple representations of code have been developed to reason about different aspects of programs, and each representation typically focuses on a unique aspect [48]. Here, based on our running example, we introduce three classic code representations, i.e., *abstract syntax tree*, *control flow graph* and *call graph*.

Abstract Syntax Tree (AST). AST provides a tree representation to encode both syntactic and lexical knowledge of source code [53]. For example, Figure 2a shows an AST of the code fragment wrapped by the red box in Figure 1a, which includes a variety of syntactic and lexical symbols, e.g., `modifier` and `findLeader`. As an AST captures both syntactic and semantic code features, it has been widely used in program comprehension tasks, e.g., code clone detection [17], program repair [45], and log placement [25].

Control Flow Graph (CFG). CFG emulates all possible paths that may be traversed during program execution at the granularity of basic code blocks. More specifically, it describes the sequence in which blocks are reached under specific execution conditions. As illustrated in Figure 2b, nodes of a CFG correspond to basic blocks, and edges denote their control flows. Generally speaking, CFG encapsulates the dependencies among code blocks, which reveals the contextual information (i.e., neighborhood information of code blocks) in source code[10].

Call Graph (CG). CG depicts the invocation relationships among functions. Figure 2c depicts a CG built upon our running example, where nodes are functions (e.g., `findLeader` and `getCurrentVote`) and edges indicate their calling relationships (e.g., `findLeader` calls `getCurrentVote`). Especially, the CFG and CG represent complementary (intra- and inter-procedure) control flows, which can be combined as an **Inter-procedural Control Flow Graph (ICFG)** to provide a complete control flow view on source code.

2.3 Our Insight

Recent studies primarily characterize logging practices at the level of *code blocks* [5, 26, 27, 58]. That is, they partition source code into blocks and then model the features of *logged blocks* — code blocks with logging statements — to predict log levels. The intuition is that the rationale behind logging statements can be revealed by understanding behaviors or states of programs in logged blocks [25]. For example, as logging statements in try-catch blocks are commonly developed to monitor program exceptions, most of them are assigned with high verbosity levels, e.g., *warn* and *error*.

More specifically, existing solutions [21, 22, 27] largely make decisions of log levels based on the code blocks in which logging statements reside (i.e., *intra-block information*). Towards this end, they have investigated open-source software systems and suggested that syntactical structures from ASTs and log messages (i.e., static texts in logs) are beneficial for log level suggestions. For example, DeepLV [27] extracts syntactical symbols from ASTs and then combines them with log messages to predict log levels.

However, we discover that intra-block information does not capture sufficient code features for deciding log levels. Take two logged blocks from Zookeeper in Figure 1 as examples. Both logged blocks only include logging statements without additional information of program behaviors and thus share similar intra-block information — *{Expression Statement, Method Invocation, Parameter List, Parameter, Variable}*. Consequently, it is challenging to distinguish their logging characteristics based solely on intra-block information. In the analysis of *inter-block information* from Figure 2c, we observe that most logging statements in the callee functions of `findLeader` in Figure 1a (e.g., `recreateSocketAddress`) are at *warn* level, which indicates the high severity of `findLeader`. On the contrary, when analyzing neighboring code blocks of `getPeersMatching` in Figure 1b, we find no exception processing or failure handling logic. Therefore, it is straightforward to conclude that these two logged blocks should be assigned with different log levels — the former logged block should be given a higher log level than the latter due to its higher severity.

Unfortunately, prior approaches spare limited attention on *inter-block* features, making it difficult to suggest log levels when it requires knowledge from neighboring code blocks. While DeepLV attempts to enlarge the size of code blocks (i.e., a code block starts from the start of a function and ends at the target logging statement) to include additional block contexts, we argue that it still suffers from the lack of inter-block information. This is because DeepLV only considers AST structures that do not reflect the dependencies among code blocks. In particular, under the view of DeepLV, two logged blocks in Figure 1 should be given the same log level as they share similar structures: the logging statements located after a *for-each* loop with a nested *if* statement. However, this contradicts the fact that they are actually at different verbosity levels.

In this work, we propose leveraging intra-block and inter-block information cooperatively to suggest log levels. Following recent literature [27, 53], we present the *intra-block information* as syntactical/lexical symbols and structures in an AST. In terms of the

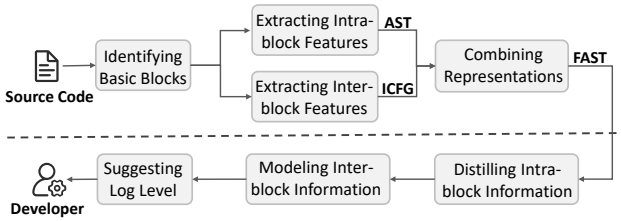


Figure 3: Overview of TELL

inter-block information, we extract contexts of code blocks by identifying their neighbors in an ICFG.

To cooperatively model multiple levels of code block features (i.e., intra-block and inter-block information), we first encode them into a joint data structure. To do so, we combine the AST and ICFG into a new code graph called *Flow of Abstract Syntax Tree* (FAST), where nodes in the graph are code blocks attributed with their ASTs, and edges represent control flows.

Afterward, an automated approach is required to gather information from the FAST to suggest log levels. Towards this end, we take inspiration from the recent developments of graph neural networks (GNNs) in many problem domains [12, 23, 43], e.g., social networks and recommendations. Since GNNs excel at incorporating graph structures into the node representations by recursively propagating information over a graph, we believe GNN should also exhibit well in extracting multi-level block features given the graph nature of our FAST. Inspired by this, our objective is to design a specialized GNN upon the FAST to facilitate log level suggestions. We reach this by developing a hierarchical neural architecture, *Hierarchical Block Graph Network* (HBGN), to model information from ASTs to ICFGs. Full details will be presented in Section 3.3.

3 APPROACH

3.1 Overview

Figure 3 illustrates the high-level view of TELL, which receives the source code of programs and suggests verbosity levels for logging statements. TELL provides an end-to-end solution with two phases: building the Flow of Abstract Syntax Tree (FAST) from source code and predicting log levels upon the FAST.

Given a program, we first identify code blocks and extract their abstract syntax trees (ASTs) as the intra-block information. Then, we build an inter-procedure control flow graph (ICFG) upon code blocks to capture the inter-block information. Afterward, the AST and ICFG are merged into a joint graph structure called FAST.

We further adopt a neural architecture called *Hierarchical Block Graph Network* (HBGN) to exploit multiple levels of code block information cooperatively for log level suggestions. The key idea behind our HBGN is to *leverage graph neural networks to propagate and aggregate information over the FAST to learn multi-level block features*. More specifically, HBGN first parameterizes each code block as a vector representation (i.e., embedding) by distilling intra-block information from the block's AST. Then, HBGN updates the block embeddings by modeling inter-block information from neighboring blocks. As recent studies discover that logging decisions are related to not only code structures but log messages [25, 27], we treat static messages of logging statements as auxiliary information to further refine the block embeddings.

After obtaining latent representations for code blocks, HBGN predicts suitable log levels for logged blocks to assist developers in improving logging practices.

3.2 Building FAST Representation

In this section, we present how to parse the source code of programs into the FAST representation, which encodes both intra-block and inter-block information.

Our objective is to suggest log levels at the granularity of code blocks by following prior studies [21, 22, 27]. To do so, the first design choice is deciding the granularity of code blocks for extracting their ASTs and ICFG. Intuitively, the granularity can be at the fine-grained basic block level or coarse-grained function level. Our analysis of nine large-scale software systems in Table 1 suggests that most coarse-grained blocks, if not all, consist of complicated program behaviors, resulting in various logging statements at different levels. In contrast, the logic of a basic block is simple, and thus we hardly observe such a block containing multiple logging statements or verbosity levels. In particular, our manual analysis on systems in Table 1 shows that only 0.8% of logged basic blocks include more than one log level. Therefore, we choose to analyze the fine-grained code blocks, namely basic blocks, in this work.

3.2.1 Identifying Basic Blocks. In order to obtain code basic blocks, we first extract ASTs from source code on the basis of functions. Then, we traverse each AST in pre-order and identify symbols related to control flow transfers (e.g., `ForStatement`) to separate functions into basic blocks. Take the code fragment in Figure 1a as an example. By cutting the code at the `ForStatement` on line 4, we identify the `findLeader start` block wrapped with the red box.

3.2.2 Extracting Intra-block and Inter-block Information. As both intra-block and inter-block information play essential roles in making decisions of logging statements, we extract and combine them into a joint representation for follow-up log level analysis. In particular, we use the AST built upon each identified block as its intra-block representation. For example, the two gray boxes in Figure 4 shows the intra-block information of `findLeader start` and `If Condition` basic blocks. In recent years, AST has been widely adopted for modeling code blocks because nodes in ASTs can preserve both semantic and syntactic knowledge of source code [53]. Specifically, the leaf nodes of ASTs (e.g., `leaderServer`) reflect the lexical features that represent latent semantics, while the non-leaf nodes of ASTs (e.g., `ForStatement`) capture syntactic features that represent syntactic structures. It is worth noticing that we further exclude all AST nodes related to logging guards (e.g., `if(isTraceEnabled)`) to avoid biases in log level suggestions like previous work [27].

After extracting ASTs to preserve intra-block information, we construct an ICFG to capture control flow transfers (e.g., branches and calling relationships) that represent inter-block information. In general, an ICFG provides a comprehensive view on the context of code blocks. Following the definition in Section 2.2, we construct an ICFG by unifying a control flow graph (CFG) and a call graph (CG). For code blocks identified within a function, we first correlate

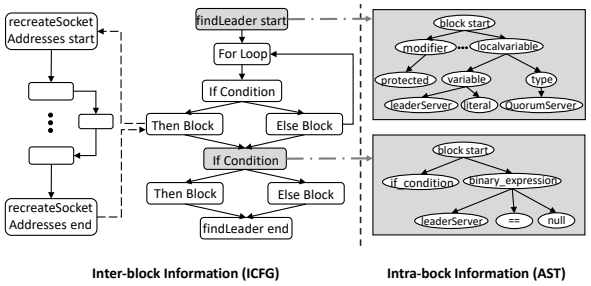


Figure 4: The Flow of AST built upon the running example. ICFG reflects inter-block information and AST encodes intra-block information.

them based on control flows to constitute a CFG. More specifically, we connect two blocks if they are correlated by control statements, e.g., `ForStatement`. For instance, the `BreakClause` statement from line 8 in Figure 1a connects the `Then Block` and the second `If Condition` blocks as shown in Figure 2b. In addition to a CFG representing control flows within functions, we further construct a CG to analyze calling relationships among functions. For this purpose, we first identify caller functions by finding the AST nodes that initiate function invocations, e.g., `Invocation`. Then, we extract callee functions by searching for functions with names and argument types matching those of caller functions. Afterward, we link caller and callee functions by creating invocation and return edges, as illustrated in Figure 2c.

3.2.3 Combining Representations. To cooperatively characterize log levels, we integrate multiple levels of block information (i.e., AST and ICFG) into the FAST. Figure 4 provides an example of a simplified FAST constructed from the code fragment in Figure 1a. We differentiate intra-procedural (CFGs) and inter-procedural (CGs) control flows with directed solid and dashed edges. As can be seen, FAST hierarchically integrates syntactical and semantic features from symbols in the AST and contextual features from topology structures in the ICFG, which provides an informative and compact representation for the log level suggesting task.

To gain further insights, we use two logged blocks in Figure 1 to demonstrate how different properties in a FAST can facilitate log level prediction. At first glance, since both logged blocks contain only logging statements, it is challenging to determine suitable log levels based solely on intra-block information. However, by integrating multi-level block information, the log levels for these two logged blocks can be clearly distinguished. In specific, we can easily identify the severity of the logged block in Figure 1a by incorporating one of its callee functions, `recreateSocketAddresses`, in the FAST. Since most of logging statements in `recreateSocketAddresses` are at the *warn* level, the caller function `findLeader` should also be of importance and its failure signifies potential runtime issues. On the contrary, as for the other logged block in Figure 1b, we cannot find logging statements with high verbosity levels or any failure/exception handling semantics throughout the FAST. This discovery indicates that the second logged block is only informative and can be ignored on a regular basis without missing critical runtime information. As a result, it is not difficult to determine that its log level should be lower than that of the first logged block.

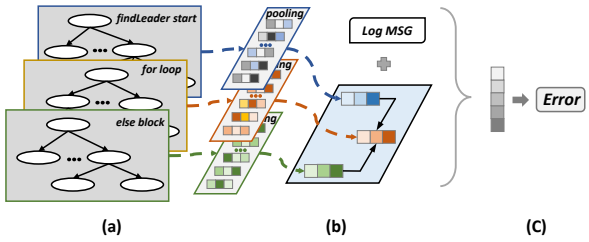


Figure 5: Illustration of our proposed Hierarchical Block Graph Network. (a), (b) and (c) present intra-block information distillation, inter-block information modeling, and log level suggestions, respectively.

3.3 Learning Multi-level Information

We now present our proposed neural architecture, *Hierarchical Block Graph Network* (HBGN), that exploits multiple levels of code block information in the FAST for log level predictions. Figure 5 illustrates the HBGN's workflow, which consists of three major components: 1) distilling the intra-block information, which parameterizes each code block as a vector (i.e., embedding) by preserving the structure of its AST; 2) modeling the inter-block information, which updates the block embedding by propagating embeddings from neighboring blocks; 3) suggesting log levels, which yields suitable levels for individual logged blocks based on their embeddings. We will elaborate on these components in the following sections.

3.3.1 Distilling Intra-Block Information. AST is a kind of tree that captures both syntactic and semantic information within code blocks [53]. Nonetheless, various structures and symbols (e.g., `type` and `leaderServer`) of ASTs make it challenging to distill the intra-block information. To address this issue, we propose a graph embedding approach, which takes as input an AST instance in an arbitrary size and outputs an embedding (i.e., vectorized representation) to encode both its syntactic and semantic features.

Formally, given an AST instance t with its topology structure, we first initialize its symbol embeddings $\mathbf{W}_w \in \mathbb{R}^{S\times d}$ via word2vec [33], where S is the number of unique symbols in the AST and d is the size of a symbol embedding. For a symbol $s \in t$, we can retrieve its initial embedding $\mathbf{e}_s$ by:

$$\mathbf{e}_s = \mathbf{W}_w^T x, \tag{1}$$

where x is the one-hot encoding of the symbol s. As such, $\mathbf{e}_s \in \mathbb{R}^d$ encodes s' own characteristics.

Beyond the own characteristics, information of neighboring symbols in ASTs also plays a crucial role in representing the ego symbol. Taking the symbol `vote` in Figure 2a as an example, `localvariable` → `type` → `vote` describes that there exists a local variable whose type is `vote`. Without such a multi-hop relation path, it will be hard to infer the semantics of the symbol `vote`. To model such relation information, TELL formulates an AST as a graph and applies a graph neural network [12] (GNN) to recursively propagate symbol embeddings on the AST. Formally, given a symbol s, the l-th embedding propagation layer updates its embedding by aggregating the information propagated from neighboring symbols as follows:

$$\mathbf{e}_s^{(l)} = \sigma((\mathbf{e}_s^{(l-1)} || \mathbf{e}_{\mathcal{N}_s}^{(l-1)})\mathbf{W}_\alpha^{(l)}), \tag{2}$$

where $\mathbf{e}_s^{(l)} \in \mathbb{R}^{d^{(l)}}$ is the embedding of s after stacking l layers, $d^{(l)}$ presents the embedding size at l-th layer, and $\mathbf{e}_s^{(0)}$ denotes the word2vec-initialized embedding $\mathbf{e}_s$; $\mathbf{W}_\alpha^{(l)} \in \mathbb{R}^{2d^{(l-1)} \times d^{(l)}}$ is a trainable weight matrix to distill useful information from neighboring symbols; $||$ is the concatenation operation, and σ is an activation function set as LeakyReLu [41]. $\mathbf{e}_{\mathcal{N}_s}^{(l-1)}$ stands for the neural messages aggregated from s's neighbors $\mathcal{N}_s$, which is formulated as:

$$\mathbf{e}_{\mathcal{N}_s}^{(l-1)} = \sum_{s' \in \mathcal{N}_s} \frac{\mathbf{e}_{s'}^{(l-1)}}{\sqrt{|\mathcal{N}_{s'}||\mathcal{N}_s|}}, \tag{3}$$

where $1/\sqrt{|\mathcal{N}_{s'}||\mathcal{N}_s|}$ is the discount factor between two symbols s and s'. After applying L embedding propagation layers, we are able to obtain a series of representations for the symbol s, $\{\mathbf{e}_s^{(0)}, \cdots, \mathbf{e}_s^{(L)}\}$, which encode different-hops of relation information in the AST. We hence adopt the layer-aggregation mechanism [47] to concatenate the representations of s into a single vector, as follows:

$$\mathbf{e}_s^* = \mathbf{e}_s^{(0)} || \cdots || \mathbf{e}_s^{(L)}, \tag{4}$$

Having established the GNN-based representations for all symbols in an AST t, we would like to have a holistic view of this AST and encapsulate all syntactic and semantic information into an AST representation $\mathbf{h}_t$. Here we employ a pooling function over these symbol representations:

$$\mathbf{h}_t = \rho([\mathbf{e}_{i_1}^*, \mathbf{e}_{i_2}^*, \cdots, \mathbf{e}_{i_S}^*]), \tag{5}$$

where ρ is set as the average pooling function to delineate important features from all symbol representations; $\mathbf{e}_{i_1}^*, \mathbf{e}_{i_2}^*, \cdots, \mathbf{e}_{i_S}^*$ are the representations of symbols $i_1, i_2, \cdots, i_S$, respectively.

In summary, TELL compresses the intra-block information from an AST instance, obtaining both syntactic and semantic features.

3.3.2 Modeling Inter-Block Information. Inter-block information complements AST's intra-block information and provides additional clues to log level predictions. Therefore, we design the second GNN, which updates block embeddings obtained from the previous step by propagating inter-block information in a FAST.

Specifically, TELL treats each block's AST in the FAST as an individual node and then hires another GNN to gather neighboring blocks' information to update node embeddings. Formally, given the AST representation $\mathbf{h}_t$ of an AST t, TELL updates its representation by combining neural messages from its neighboring blocks:

$$\mathbf{h}_t^{(k)} = \sigma((\mathbf{h}_t^{(k-1)} || \sum_{t' \in \mathcal{N}_t} \frac{\mathbf{h}_{t'}^{(k-1)}}{\sqrt{|\mathcal{N}_{t'}||\mathcal{N}_t|}}) \mathbf{W}_\beta^{(k)}), \tag{6}$$

where $\mathbf{h}_t^{(k)} \in \mathbb{R}^{d^{(k)}}$ is the updated representation of t at the k-th inter-block propagation layer, and $\mathbf{h}_t^{(0)}$ is initialized as $\mathbf{h}_t$ in Equation (5); $\mathcal{N}_t$ is the set of blocks directly connecting to t in a FAST, and $\mathbf{W}_\beta^{(k)} \in \mathbb{R}^{2d^{(k-1)} \times d^{(k)}}$ is a trainable matrix to refine the neural message passing from the neighboring block t'.

After K inter-block propagation layers, we obtain multiple updated representations of the AST $t - \{\mathbf{h}_t^{(0)}, \cdots, \mathbf{h}_t^{(K)}\}$. As different layers emphasize varying order connections to the AST t, these representations could capture different levels of block information in a FAST. Hence, we concatenate them to constitute the final representation for t:

$$\mathbf{h}_t^* = \mathbf{h}_t^{(0)} || \cdots || \mathbf{h}_t^{(K)}. \tag{7}$$

Note that $\mathbf{h}_t^*$ represents the embedding of not only an AST t but a code block. As a result, our hierarchical GNNs (HBGN) allow TELL to encode multi-level information (i.e., intra-block and inter-block information) into the representations of code blocks, thus facilitating the downstream log prediction task.

3.3.3 Suggesting Log Level. In addition to intra-block and inter-block features, TELL also incorporates *log messages* as auxiliary information to refine code block representations. More formally, TELL models the representation q_m of log message m as:

$$\mathbf{q}_m = \rho([\mathbf{z}_{i_1}, \mathbf{z}_{i_2}, \cdots, \mathbf{z}_{i_N}]), \tag{8}$$

where $\mathbf{z}_{i_1}, \mathbf{z}_{i_2}, \cdots, \mathbf{z}_{i_N}$ are the word2vec-initialized embeddings of tokens $i_1, i_2, \cdots, i_N$ in log message m, and N is the number of unique tokens; ρ is the average pooling function that combines token embeddings into the representation of log message.

Having modeled multi-level block information and log messages of code blocks, TELL's subsequent task is to suggest suitable log levels (i.e., *trace, info, warn, debug, and error*) for logged blocks. That is, we need to answer the following question: "If node t is one logged block of interest, which level should we label it?". Here, we build a classifier upon embeddings of multi-level information h_t^* and log messages q_m. Technically, the classifier is a fully connected layer that converts multi-level information and log messages to logits over five log levels:

$$\hat{\mathbf{x}}_b = (\mathbf{h}_t^* \mathbf{W}_t || \mathbf{q}_m \mathbf{W}_p) \mathbf{W}_f + \mathbf{b}_f, \tag{9}$$

where $\mathbf{W}_t \in \mathbb{R}^{\sum_{i=0}^{K} d^{(i)} \times d}$, $\mathbf{W}_p \in \mathbb{R}^{d \times d}$ and $\mathbf{W}_f \in \mathbb{R}^{2d \times 5}$ are the transformation matrices, and $\mathbf{b}_f$ is the bias term.

To learn model parameters in HBGN, we cast the task of log level suggestions as a classification problem. Before designing the final objective function for classification, we first need to choose an encoding schema of log levels. The two primary choices are one-hot encoding and ordinal encoding. Specifically, one-hot encoding treats log levels as unrelated classes. For example, *info* and *warn* are encoded as $[0, 0, 1, 0, 0]$ and $[0, 0, 0, 1, 0]$, respectively. Correspondingly, log level suggestion is multi-class classification, and hence we adopt the softmax-normalized categorical cross-entropy as HBGN's objective function.

However, a recent study suggests that log levels have an ordinal nature [27] — the levels preserve an order among them. For example, if a software system is configured to generate *info* logs, the system will also enable logging statements at higher levels, e.g., *warn*. Therefore, it would be more reasonable to encode log levels while preserving their ordinal logging nature, e.g., encoding *info* and *warn* as $[1, 1, 1, 0, 0]$ and $[1, 1, 1, 1, 0]$. As a result, we convert log level suggestions from multi-class into binary classification and then adopt the standard sigmoid-normalized binary cross-entropy as HBGN's objective function.

In summary, TELL adopts HBGN to represent logged blocks with multi-level information in a FAST and log messages. It then utilizes the high-quality representations to suggest suitable log levels.

4 EVALUATION

In this section, we focus on evaluating the effectiveness of TELL in suggesting log levels. In particular, we investigate the following research questions (RQs):

- **RQ1:** How does TELL perform on log level suggestions compared with the state-of-the-art?
- **RQ2:** To what extent do diverse design decisions affect the performance of TELL on log level suggestions?
- **RQ3:** How well does TELL characterize log levels across systems?

4.1 Implementation

We use the *tree-sitter* toolset[2] to extract ASTs for Java programs. The rest of the FAST building (e.g., ICFG construction) is developed in 3,437 lines of Python code. We implement our HBGN model using TensorFlow [1]. The model is optimized by Adam optimizer [18], where the batch size and epochs are fixed at 16 and 100. To address the over-fitting problem, we attach a dropout layer with a dropping ratio of 0.2. All model parameters are initialized with Kaiming [13]. We adopt word2vec with the skim-gram algorithm [33] to obtain the embeddings for symbols in AST and tokens in log messages.

To select hyper-parameters in HBGN, we apply a grid search: the learning rate is tuned amongst {0.001, 0.01, 0.1}; the symbol and token embedding size is searched in {16, 32, 64}; the number of GNN layers is tuned amongst {1, 2, 3}. In light of the best performance, we report experimental results in a setting with the learning rate as 0.1, the embedding size of AST symbols and message tokens as 64, a 2-layer GNN with hidden dimensions of 64 and 32 for intra-block information distilling, and a 3-layer GNN with hidden dimensions as 64, 32, and 16 to model inter-block information. The threshold for ordinal encoding-based classification is set to 0.5 by following the way in [27]. Note that the one-hot encoding-based classification does not require a threshold as it uses softmax to predict the log level of the greatest probability.

All experiments are performed on a server with Intel Xeon Gold 6248 CPU @ 2.50GHz, 188GB physical memory, and an NVIDIA Tesla V100 GPU. The OS is Ubuntu 20.04.2 LTS.

4.2 Experiment Setup

Datasets. We collect nine widely used Java software systems from various domains (e.g., databases and search engines) to evaluate TELL's effectiveness in suggesting log levels. Since these systems are well-maintained and follow good logging practices, they have been commonly adopted in recent log-related studies [5, 25–27]. Table 1 summarizes the log level distribution of these systems. Note that our statistics are not necessarily the same as existing studies due to the different granularity of code blocks — we analyze the most fine-grained basic blocks. As the *fatal* level only occupies 0.04% of all logged blocks, we focus our evaluation on the other five log levels, i.e., *trace, debug, info, warn, error*. For each experimental system, we select 60%, 20%, and 20% of its logged blocks to constitute the disjoint training, validation, and testing sets. The stratified random sampling [39] is applied to split logged blocks to ensure that sampled sets have the same log level distribution.

[2] https://tree-sitter.github.io/tree-sitter/

Table 1: Log level distribution of nine software systems.

Systems	Version	LB	TB	DB	IB	WB	EB	FB	MIXB
Cassandra	3.11.4	1,317	325	202	280	220	280	0	10(0.8%)
Elasticsearch	7.4.0	5,363	902	1,195	2,048	671	473	3	71(1.3%)
Flink	1.8.2	2,475	35	793	815	455	352	6	19(0.8%)
Hbase	2.2.1	5,146	461	1,153	1,454	1,286	765	0	27(0.5%)
Jmeter	5.3.0	1,762	1	654	295	406	401	0	5(0.3%)
Kafka	2.3.0	1,426	199	414	335	171	293	0	14(1.0%)
Karaf	4.2.9	698	15	152	192	151	185	0	3(0.4%)
Wicket	8.6.1	408	11	167	45	88	96	0	1(0.2%)
Zookeeper	3.5.6	1,496	33	225	599	372	262	0	5(0.3%)
Average	—	2,232	220	551	674	425	345	1	17(0.8%)

Note: LB is the number of logged blocks. TB, DB, IB, WB, FB and MIXB refer to the number of Trace, Debug, Info, Warn, Fatal and Mixed level blocks.

Ground Truth. Each logged block is labeled by the level of its constituent logging statement(s). For example, as the logging statement in Figure 1a is at the *warn* level, we label the logged block in the green box as *warn*. Notice that given a single block with multiple verbosity levels, we only consider the highest level by following the ordinal nature of log levels [27].

Metrics. We adopt *Accuracy*, *Area Under the Curve* (AUC) and *Average Ordinal Distance Score* (AOD) as metrics for evaluating and comparing the performance of TELL. In specific, the Accuracy measures the correctly suggested logged blocks against all logged blocks. Multi-class AUC, ranging between $(0, 1)$, evaluates the capability of a model in distinguishing different classes. In this study, the higher the AUC value is, the better TELL is at log level classification. AOD designed by [27] calculates the average distance between the suggested log level and actual log level in logged blocks, which is formulated as: $AOD = \frac{\sum_{i=1}^{N}(1-Dis(a_i,s_i)/MaxDis(a_i))}{N}$, where N is the number of logged blocks predicted by TELL, $Dis(a_i, s_i)$ measures the distance between the actual log level a_i and the suggested log level s_i (e.g., the distance between *warn* and *error* is 1), and $MaxDis(a_i)$ stands for the maximum possible distance of the actual log level a_i (e.g., the maximum distance of *error* is 4 from *trace*). Intuitively, a higher AOD value indicates a suggested log level closer to the actual level.

4.3 Improvement over the State-of-the-art

Our first research question is how TELL performs against the state-of-the-art work, DeepLV [27], that suggests log levels based on syntax symbols from ASTs and log messages. To answer this question, we use both TELL and DeepLV to predict log levels on our studied systems in Table 1. For short, we call DeepLV using syntax features as DL(Syn), DeepLV using both syntax features and log messages as DL(Comb), TELL using multi-level block information as TL(Mul), and TELL using both multi-level information and log messages as TL(Comb) henceforth. Note that because the source code of DeepLV is not available to us, we quote the results reported in the original paper to show its effectiveness.

4.3.1 Overall suggestion results. Table 2 summarizes the overall performances of TELL and DeepLV on nine studied systems. We observe that TELL consistently outperforms DeepLV on all systems regarding Accuracy, AUC, and AOD. More specifically, based on DeepLV, TELL achieves an average relative improvement of 9.9%, 7.1%, 3.5% for the Accuracy, AUC and AOD, respectively. The results demonstrate that TELL is able to suggest more suitable levels for

Table 2: Comparison between TELL and DeepLV in terms of Accuracy, AUC and AOD.

Systems	Accuracy				AUC				AOD			
	DL(Syn)	DL(Comb)	TL(Mul)	TL(Comb)	DL(Syn)	DL(Comb)	TL(Mul)	TL(Comb)	DL(Syn)	DL(Comb)	TL(Mul)	TL(Comb)
Cassandra	0.537	0.606(+6.9%)	0.625(+8.8%)	**0.635(+9.8%)**	0.788	0.842(+5.4%)	0.873(+8.5%)	**0.884(+9.6%)**	0.789	0.805(+1.6%)	0.809(+2.0%)	**0.812(+2.3%)**
ElasticSearch	0.519	0.577(+5.8%)	0.674(+15.5%)	**0.703(+18.4%)**	0.779	0.813(+3.4%)	0.891(+11.2%)	**0.905(+12.6%)**	0.776	0.802(+2.6%)	0.823(+4.7%)	**0.841(+6.5%)**
Flink	0.525	0.652(+12.7%)	0.687(+16.2%)	**0.729(+20.4%)**	0.782	0.851(+6.9%)	0.910(+12.8%)	**0.925(+14.3%)**	0.787	0.838(+5.1%)	0.855(+6.8%)	**0.863(+7.6%)**
Hbase	0.559	0.603(+4.4%)	0.702(+14.3%)	**0.707(+14.8%)**	0.831	0.842(+1.1%)	0.905(+7.4%)	**0.921(+9.0%)**	0.814	0.817(+0.3%)	0.857(+4.3%)	**0.873(+5.9%)**
Jmeter	0.551	0.623(+7.2%)	0.700(+14.9%)	**0.737(+18.6%)**	0.835	0.839(+0.4%)	0.899(+6.4%)	**0.921(+8.6%)**	0.823	0.809(-1.4%)	0.839(+1.6%)	**0.872(+4.9%)**
Kafka	0.507	0.518(+1.1%)	0.621(+11.4%)	**0.642(+13.5%)**	0.787	0.795(+0.8%)	0.864(+7.7%)	**0.888(+10.1%)**	0.769	0.775(+0.6%)	0.778(+0.9%)	**0.812(+4.3%)**
Karaf	0.565	0.672(+10.7%)	0.700(+13.5%)	**0.750(+18.5%)**	0.843	0.856(+1.3%)	0.898(+5.5%)	**0.908(+6.5%)**	0.806	0.816(+1.0%)	0.839(+3.3%)	**0.867(+6.1%)**
Wicket	0.573	0.638(+6.5%)	0.695(+12.2%)	**0.744(+17.1%)**	0.831	0.850(+1.9%)	0.897(+6.6%)	**0.899(+6.8%)**	0.789	0.793(+0.4%)	0.837(+4.8%)	**0.856(+6.7%)**
Zookeeper	0.528	0.609(+8.1%)	0.696(+16.8%)	**0.746(+21.8%)**	0.796	0.848(+5.2%)	0.903(+10.7%)	**0.924(+12.8%)**	0.788	0.820(+3.2%)	0.858(+7.0%)	**0.887(+9.9%)**
Average	0.540	0.611(+7.0%)	0.678(+13.7%)	**0.710(+17.0%)**	0.808	0.837(+2.9%)	0.893(+8.5%)	**0.908(+10.0%)**	0.793	0.808(+1.5%)	0.833(+3.9%)	**0.854(+6.0%)**

Note: DL(Syn), DL(Comb), TL(Mul) and TL(Comb) stand for DeepLV with syntactic contexts, DeepLV with both syntactic contexts and log messages, TELL with multi-level block information, and TELL with multi-level block information and log messages.

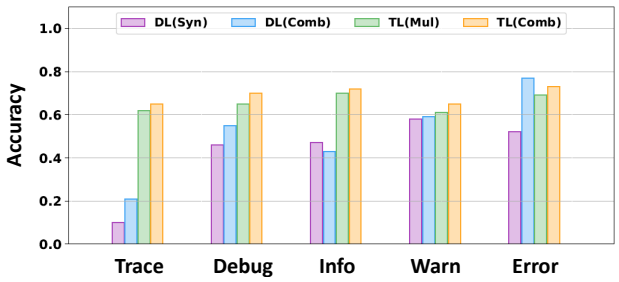


Figure 6: Comparison between TELL and DeepLV in individual log levels.

logging statements. Even though a log level is falsely predicted by TELL, it would be closer to the actual log level than DeepLV.

By further comparing TL(Mul) and TL(Comb) in Table 2, we discover that TELL can suggest log levels with higher accuracy if incorporating the semantics of log messages into the latent representation of logged blocks. This is expected as developers always insert unique words (e.g., *pthread_create fail*) into logging statements to briefly explain interesting behaviors or states of programs (e.g., software failures), which provide additional supervision to facilitate log level predictions. Another interesting observation is that even without considering log messages, TELL, namely TL(Mul), still outperforms DeepLV with log messages, namely DL(Comb), on our studied systems. In particular, TL(Mul) improves the average Accuracy, AUC, and AOD of DL(Comb) by 6.7%, 5.6%, and 2.4%. We attribute such an improvement to the fact that TELL adopts HBGN to explicitly exploit multiple levels of code block information (both ASTs and ICFGs) from source code. However, DeepLV only focuses on syntactic structures from ASTs. Therefore, even with the help of log messages, DeepLV still has a lower accuracy due to the lack of inter-block information.

4.3.2 Performance on different levels. In addition to the overall performance, a separate accuracy evaluation for individual log levels is also of importance. This is because various stakeholders typically focus on logging statements at different levels. For example, software developers care more about logs at the *debug* level for program debugging tasks. On the contrary, software maintainers are more interested in logs at the *warn* or *error* levels as their daily tasks are to locate unexpected software execution, e.g., faults and terminations. As a result, it is necessary to explore the effectiveness of TELL for individual log levels to better understand its applicability in real-world DevOps scenarios.

Table 3: Average performance of different TELL variants on log level suggestions over nine datasets.

Model	Encoding	Accuracy	AUC	AOD	Time(s)
GNN-NONE	Ordinal	0.517	0.840	0.733	3,459
NONE-GNN	Ordinal	0.602	0.875	0.796	1,780
ASTNN-GNN	Ordinal	0.618	0.878	0.802	8,908
GNN-GNN	One-hot	0.652	0.848	0.810	4,688
GNN-GNN(HBGN)	**Ordinal**	**0.678**	**0.893**	**0.833**	**4,713**

Figure 6 shows the suggestion results on different log levels. As we can see, TELL brings substantial improvements over DeepLV, especially for low log levels, e.g., *trace*. Specifically, for the *trace*, *debug*, *info* and *warn* level, TELL's accuracy are 44.3%, 14.9%, 29.4%, and 5.0% higher than those of DeepLV, respectively. From Figure 6, we also observe that both TL(Mul) and DL(Comb) achieve high accuracy on the *error* level, and DL(Comb) even achieves slightly higher accuracy. One possible reason is that static messages in logging statements play a major role in suggesting the *error* level. Therefore, by incorporating additional channels of code features (i.e., inter-block information), we do not obtain higher performance necessarily. For example, we can easily identify the *error* level for a logging statement if finding certain texts in its log messages, e.g., *crashes*, *failures*, and *errors*. However, we would like to point out that without log messages, TL(Mul) greatly outperforms DL(Syn), which is credited to the benefits of multi-level block information in log level suggestions.

Result 1: **Compared with the state-of-the-art approach, TELL significantly improves the accuracy for log level suggestions, especially for low levels, such as *trace (44.3%)*, *debug (14.9%)*, and *info (29.4%)*.**

4.4 Impacts of Different Design Choices

To answer RQ2, we explore the impact of different design choices in HBGN on the performance of TELL. Especially, we try different models to distill intra-block and inter-block information and investigate two encoding schemas of log levels. To explain the internals of HBGN, we further visualize code block embeddings.

4.4.1 Ablation study. The performance of different TELL variants are listed in Table 3. For a fair comparison, we report the results without considering log messages in the code block representation

```
1 public Set<Watcher> materialize(Watcher.Event.KeeperState state,
2                 Watcher.Event.EventType type,String clientPath) {
3     ...
4     switch (type) {
5     case None: ...
6     case NodeDataChanged:
7     case NodeCreated: ...
8     case NodeChildrenChanged: ...
9     case NodeDeleted: ...
10    default:                                          Logged Block: error
11        String msg = "Unhandled watch event type " + type
12            + " with state " + state + " on path " + clientPath;
13        LOG.error(msg);
14        throw new RuntimeException(msg);
15    }
16    return result;
17}
```

```
1 protected void syncWithLeader(long newLeaderZxid) throws Exception {
2     ...
3     switch(qp.getType()) {
4     case Leader.PROPOSAL: ...
5     case Leader.COMMIT:
6     case Leader.COMMITANDACTIVATE: ...
7     case Leader.INFORM:
8     case Leader.INFORMANDACTIVATE: ...
9     case Leader.UPTODATE:                             Logged Block: info
10        LOG.info("Learner received UPTODATE message");
11        ...
12}
```

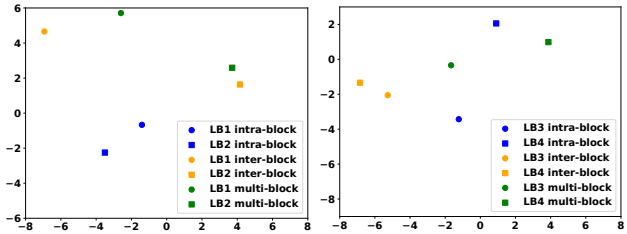
Figure 7: Two code fragments with logged blocks containing different intra-block information and similar inter-block information in Zookeeper.

learning. Generally speaking, all our model decisions contribute to the final performance of log level suggestions, more or less.

More specifically, to estimate the effectiveness of HBGN's intra-block information distillation, we compare our GNN model with two baselines, i.e., NONE (without intra-block information) and ASTNN [53]. In particular, by no intra-block information, we refer to initializing the embeddings of ASTs using Kaiming [13] without any supervision signals. Instead of using a GNN, ASTNN learns code representations upon ASTs by a recursive neural network. In Table 3, GNN-GNN (HBGN), ASTNN-GNN, and NONE-GNN report the results of distilling intra-block information with GNN, ASTNN, and without intra-block information, respectively. We see that HBGN performs the best results w.r.t. Accuracy, AUC, and AOD. NONE-GNN, however, underperforms both ASTNN-GNN and HBGN. This matches our expectations as it completely ignores syntax and semantics included in ASTs. While ASTNN captures the tree structures in ASTs, it does not distill helpful information from neighboring AST nodes like GNNs. Instead, ASTNN receives all information propagated from neighbors, introducing noisy information and generating suboptimal block representations, which might affect its performance on log level suggestions.

For inter-block information, we compare HBGN with and without our GNN-based modeling on ICFGs. The results are reported by GNN-NONE and GNN-GNN (HBGN) in Table 3. Again, we observe that HBGN achieves much better Accuracy/AUC/AOD, improving 16.1%, 5.3%, and 10.0%, respectively. This well justifies our claim that inter-block information (i.e., block contexts) is essential for suggesting log levels. Additionally, we discover that NONE-GNN outperforms GNN-NONE. We hypothesize that this is because contextual features in ICFGs are more powerful than syntactic features in ASTs for characterizing log principles.

Next, we investigate the effectiveness of two basic log level encoding schemas, i.e., *one-hot encoding* and *ordinal encoding*. By comparing GNN-GNN with one-hot and ordinal encodings in Table 3, we find that HBGN achieves better results using ordinal encoding.



(a) LB1 and LB2 embeddings. **(b) LB3 and LB4 embeddings.**

Figure 8: Visualization of logged block embeddings using t-SNE. Best view in color.

This is predictable as ordinal encoding preserves the ordinal nature of log levels. That said, TELL still reaches promising results with one-hot encoding.

To further compare the efficiency of different TELL variants, we measure the average time cost when training each model for 100 epochs on our studied system. From Table 3, we find that one-hot encoding and ordinal encoding consume similar training time, indicating that encoding schemas have a minor influence on TELL's efficiency. Compared with ASTNN, our TELL is two times faster. This is because ASTNN requires dynamic batch sizes adjustment during the model training. We refer interested readers to [53] for its detailed training procedure.

4.4.2 Explicability of multi-level information modeling. HBGN embeds code blocks into a high dimensional (128 dimensions in our case) vector space, where the distances among vectors encode block similarities. More specifically, logged blocks far from each other would like to be assigned different levels, otherwise the same level. To further understand the internals of HBGN, we visualize the embeddings of blocks in Figure 8 via the t-SNE technique [40], projecting high-dimensional embedding spaces into a two-dimensional plane. In particular, we plot the embeddings of two logged blocks in Figure 1 (henceforth called *LB1* and *LB2*) and another two logged blocks in Figure 7 (henceforth called *LB3* and *LB4*). For ease of presentation, we call embedding spaces derived from intra-block information, inter-block information, multi-level block information as intra-block, inter-block, and multi-block spaces.

As illustrated in Figure 8a, *LB1* and *LB2* are nearby in the intra-block space but far away from each other in the inter-block space. This mirrors our intuitive understanding as *LB1* and *LB2* have similar intra-block information but different inter-block information. Additionally, we see that *LB1* and *LB2* are well separated in the multi-block space, which suggests different log levels for these two logged blocks. To gain further insight, we investigate the embeddings of *LB3* and *LB4* that have different intra-block features but similar inter-block features, as shown in Figure 7. Figure 8b plots their embeddings, demonstrating that these two logged blocks should be assigned with different levels due to the considerable distance between them in the multi-block space.

Result 2: **All of our choices in the design of the neural architecture HBGN make contributions to the final performance of *TELL* in log level suggestions.**

4.5 Cross System Suggestion

In previous sections, we have validated TELL's advantage in within-system log level suggestions: predicting log levels based on logging practices learned from the same system. We further explore TELL's effectiveness on cross-system suggestions. Our motivation is that new software systems may not follow good logging practices or even do not contain enough logged blocks to train a GNN model. As such, within-system suggestion does not work in their cases. However, if the logging practices learned from well-maintained open-source systems can be transferred to other systems, new software can also benefit from TELL. Therefore, in this section, we explore the potentials of TELL for cross-system suggestions.

Towards this end, we train HBGN on eight arbitrary systems in Table 1 and test it with 20% logged blocks randomly sampled from the remaining system. Note that log level suggestion is a five-class classification problem, so the probability of an accurate random guess is 0.2. Figure 9 reports the accuracy of cross-system suggestions. We see that TELL produces a promising suggestion accuracy, significantly outperforming the random guess. This result reveals that although software systems are designed by different developers, they share implicit guiding principles in deciding log levels. We are not surprised to observe that cross-system suggestion exhibits a lower performance than within-system suggestion. This is because different software systems most likely follow varying explicit logging practices. For example, Jmeter rarely sets *trace* level for logging statements, while this log level is prevalent in specific systems such as Cassandra and ElasticSearch.

We further compare TELL with DeepLV as it also supports cross-system suggestions. In general, TELL achieves better accuracy than DeepLV on all experimental systems. The results demonstrate that TELL is able to infer more information shared by different software systems for log level prediction via explicitly exploring multi-level block information.

Result 3: **Benefiting from cross-system suggestions,** *TELL* **has the potential to predict log levels for new software systems that even follow limited logging practices. Besides, we find that different systems share moderate logging principles in determining log levels.**

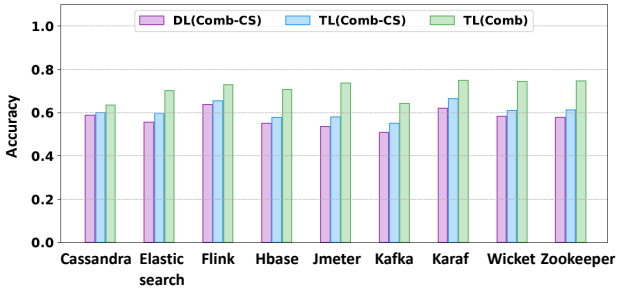


Figure 9: The accuracy of log level suggestions across systems. CS indicates Cross-System.

5 THREATS TO VALIDITY

There are five main threats to the validity. First, common to most learning-driven approaches, TELL's effectiveness relies on a high-quality training set. In particular, our approach presumes that the source code of a program for training follows good logging practices. However, since there exist no standard specifications for log level usages, the quality of logging statements cannot be always guaranteed. To reduce this threat, we evaluate TELL using nine large-scale and widely-used systems across various domains, which are also commonly adopted as experimental targets in prior log-related approaches [5, 21, 26, 27]. Second, we conducted our experiments based on only Java software systems, where the experimental results may not generalize to systems implemented in other programming languages. That said, the principle of TELL is not limited to a specific programming language. For example, both intra-block (i.e., AST) and inter-block (i.e., ICFG) information used in our approach are general code abstractions. Third, different hyper-parameters in the HBGN may affect TELL's effectiveness. Following advanced practices from existing studies [16, 43], we apply the grid search to find the optimal hyper-parameters that achieve the best accuracy. For the reproduction of our evaluation results, we present all the hyper-parameter settings in our experiments in Section 4.2. Fourth, TELL does not guarantee to generate sound and complete call graphs by matching function names and arguments. For example, caller-callee relations may be missed due to dynamic dispatches. Nevertheless, call graph construction itself is an open research problem [37] and beyond the scope of this study. We also would like to point out that modularity is one of the guiding principles in the design of TELL, which enables advanced uses to integrate more sophisticated techniques to build more accurate call graphs. Finally, we use the statistical metrics (i.e., Accuracy, AUC, and AOD) to evaluate and compare the performance of TELL with the state-of-the-art solutions. Unfortunately, whether a suggested log level is helpful for developers in practice remains unknown, and we leave it as our future work.

6 RELATED WORK

Application Log Analysis. Application logs have recently attracted increasing attention in software enhancement tasks such as anomaly detection [15, 28, 36, 54] and fault location [32, 52, 56]. For example, Zhang et al. [54] propose *LogRobust* to detect system anomalies based on unstable log sequences. Lu et al. [32] hunt concurrency bugs in the cloud via log-mining. Zhao et al. [56] develop a non-intrusive tool to profile the performance of distributed systems using the unstructured logs. The primary idea of these approaches is to leverage application logs to facilitate "after-the-fact" analysis. As a result, their performances heavily rely on the quality of logging statements in source code. This work aims to identify suitable levels for logging statements to improve the overall log quality, which benefits various log-based software engineering tasks.

Studies on Logging Practices. Due to the advances of logs in software engineering, plenty of research efforts [5–7, 11, 26, 38, 50, 57] have been made to study logging practices. Yuan et al. [50] and Chen et al. [7] first characterize the importance of logs in software debugging and maintenance through quantitative analysis. Chen et al. [5, 6] summarize five categories of common logging mistakes and analyze the usage of logging utilities in Java code. Inspired by these fundamental studies, extensive literature exists on improving

logging practices. Current logging improving approaches can be roughly categorized into two directions, where to log [25, 55, 58] and what to log [14, 27, 31, 51]. Research on where to log aims to suggest the most suitable locations for logging statements. Specifically, Zhao et al. [55] measure the entropy of a program to determine the optimal log positions. Elsewhere in the literature, Li et al. [25] and Zhu et al. [58] leverage intra-block information of logged blocks to predict log positions. In the other direction, studying what to log, existing work analyzes what message, variable, and level to log, based on the components of a logging statement. In specific, He et al. [14] explore the potential of natural language processing in predicting log messages. Yuan et al. [51] and Liu et al. [31] propose to determine which variable to log. Besides, Li et al. [22, 27] aim to suggest log levels with intra-block information. In this paper, we propose TELL to automatically incorporate multi-level code block information to predict log levels, significantly outperforming the state-of-the-art approach [27].

GNNs in Software Engineering. Graph Neural Networks (GNNs) are widely used in many problem domains due to their superiority in representation learning of graph data [12, 19, 23, 43, 46]. Recently, several studies propose exploiting GNNs in software engineering tasks, such as code clone detection [42], code summarization [20, 30], variable name prediction [2], and program representation [24, 29, 44]. In particular, Wang et al. [42] applies GNNs to capture syntax and semantic structures in ASTs and then perform code clone detection on a pair of code fragments. Allamanis et al. [2] utilize GNNs to learn program representations for variable naming and misuse. LeClair et al. [20] adopt GNNs to automatically generate natural language descriptions of source code. To the best of our knowledge, we are the first to explore the potential of GNNs in modeling multi-level code block information for log practices.

7 CONCLUSION

In this work, we explore the potential of different levels of code block information in log level suggestions. Towards this end, we first extract intra-block and inter-block block information from the AST and ICFG as a joint graph structure termed Flow of AST (FAST). Then, we design a new neural architecture, Hierarchical Block Graph Network, upon the FAST to model multi-level block information in a cooperative fashion. In this way, both intra-block and inter-block information have been explicitly incorporated into characterizing blocks with logging statements and predicting their levels. We conduct extensive experiments on nine large-scale software systems. The experimental results show that our approach outperforms the state-of-the-art approaches by large margins regarding log level suggestion accuracy.

To facilitate follow-up research, we release the source code of TELL at https://github.com/ljiahao/TeLL.

ACKNOWLEDGMENTS

We thank Yuancheng Jiang, Yinfang Chen and the anonymous reviewers for their valuable comments. This research is supported by the National Research Foundation, Singapore under its Industry Alignment Fund - Pre-positioning (IAF-PP) Funding Initiative and CCCD Key Lab of Ministry of Culture and Tourism, China. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation (OSDI)*. 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=BJOFETxR-

[3] Han Anu, Jie Chen, Wenchang Shi, Jianwei Hou, Bin Liang, and Bo Qin. 2019. An approach to recommendation of verbosity log levels based on logging intention. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 125–134. https://doi.org/10.1109/ICSME.2019.00022

[4] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (ICSM)*. 368–377. https://doi.org/10.1109/ICSM.1998.738528

[5] Boyuan Chen and Zhen Ming Jiang. 2017. Characterizing and detecting anti-patterns in the logging code. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 71–81. https://doi.org/10.1109/ICSE.2017.15

[6] Boyuan Chen and Zhen Ming Jiang. 2020. Studying the use of Java logging utilities in the wild. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 397–408. https://doi.org/10.1145/3377811.3380408

[7] 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* (2017), 330–374. https://doi.org/10.1007/s10664-016-9429-5

[8] Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming Jiang. 2018. An automated approach to estimating code coverage measures via execution logs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 305–316. https://doi.org/10.1145/3238147.3238214

[9] 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 *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. https://doi.org/10.1109/ICSE-SEIP.2017.26

[10] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 516–527. https://doi.org/10.1145/3395363.3397362

[11] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE)*. 24–33. https://doi.org/10.1145/2591062.2591175

[12] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS)*. 1025–1035. https://proceedings.neurips.cc/paper/2017/file/5dd9db5e033da9c6fb5ba83c7a7ebea9-Paper.pdf

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision (ICCV)*. 1026–1034. https://doi.org/10.1109/ICCV.2015.123

[14] Pinjia He, Zhuangbin Chen, Shilin He, and Michael R Lyu. 2018. Characterizing the natural language descriptions in software logging statements. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 178–189. https://doi.org/10.1145/3238147.3238193

[15] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. 2018. Identifying impactful service system problems via log analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 60–70. https://doi.org/10.1145/3236024.3236083

[16] Xiangnan He and Tat-Seng Chua. 2017. Neural factorization machines for sparse predictive analytics. In *Proceedings of the 40th international ACM SIGIR conference on Research and development in Information Retrieval (SIGIR)*. https://doi.org/10.1145/3077136.3080777

[17] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE)*. 96–105. https://doi.org/10.1109/ICSE.2007.30

[18] Diederik P Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR)*. http://arxiv.org/abs/1412.6980

[19] Thomas N Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=SJU4ayYgl

[20] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC)*. 184–195. https://doi.org/10.1145/3387904.3389268

[21] 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). https://doi.org/10.1109/TSE.2020.2970422

[22] Heng Li, Weiyi Shang, and Ahmed E Hassan. 2017. Which log level should developers choose for a new logging statement? *Empirical Software Engineering* (2017), 1684–1716. https://doi.org/10.1007/s10664-016-9456-2

[23] Jia Li, Yu Rong, Hong Cheng, Helen Meng, Wenbing Huang, and Junzhou Huang. 2019. Semi-supervised graph classification: A hierarchical graph perspective. In *The World Wide Web Conference (WWW)*. 972–982. https://doi.org/10.1145/3308558.3313461

[24] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2016. Gated graph sequence neural networks. In *International Conference on Learning Representations (ICLR)*. http://arxiv.org/abs/1511.05493

[25] Zhenhao Li, Tse-Hsun Chen, and Weiyi Shang. 2020. Where shall we log? studying and suggesting logging locations in code blocks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 361–372. https://doi.org/10.1145/3324884.3416636

[26] Zhenhao Li, Tse-Hsun Chen, Jinqiu Yang, and Weiyi Shang. 2019. DLFinder: characterizing and detecting duplicate logging code smells. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 152–163. https://doi.org/10.1109/ICSE.2019.00032

[27] Zhenhao Li, Heng Li, Tse-Hsun Peter Chen, and Weiyi Shang. 2021. DeepLV: Suggesting Log Levels Using Ordinal Based Neural Networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1461–1472. https://doi.org/10.1109/ICSE43902.2021.00131

[28] Fucheng Liu, Yu Wen, Dongxue Zhang, Xihe Jiang, Xinyu Xing, and Dan Meng. 2019. Log2vec: a heterogeneous graph embedding based approach for detecting cyber threats within enterprise. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1777–1794. https://doi.org/10.1145/3319535.3363224

[29] Shangqing Liu. 2020. A Unified Framework to Learn Program Semantics with Graph Neural Networks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1145/3324884.3418924

[30] Shangqing Liu, Yu Chen, Xiaofei Xie, Jingkai Siow, and Yang Liu. 2021. Retrieval-augmented generation for code summarization via hybrid {gnn}. In *International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=zv-typ1gPxA

[31] Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2019. Which variables should i log? *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2941943

[32] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. 2018. Cloudraid: hunting concurrency bugs in the cloud via log-mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 3–14. https://doi.org/10.1145/3236024.3236071

[33] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems (NIPS)*. 3111–3119. https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf

[34] Meiyappan Nagappan, Kesheng Wu, and Mladen A Vouk. 2009. Efficiently extracting operational profiles from execution logs using suffix arrays. In *2009 20th International Symposium on Software Reliability Engineering (ISSRE)*. 41–50. https://doi.org/10.1109/ISSRE.2009.23

[35] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured comparative analysis of systems logs to diagnose performance problems. In *9th {USENIX} Symposium on Networked Systems Design and Implementation (NSDI)*. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/nagaraj

[36] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi B Dasgupta, and Subhrajit Bhattacharya. 2016. Anomaly detection using program control flow graph mining from execution logs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 215–224. https://doi.org/10.1145/2939672.2939712

[37] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Modular call graph construction for security scanning of Node. js applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 29–41. https://doi.org/10.1145/3460319.3464836

[38] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. 2015. Industry practices and event logging: Assessment of a critical software development process. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*. 169–178. https://doi.org/10.1109/ICSE.2015.145

[39] Heidar Pirzadeh, Sara Shanian, Abdelwahab Hamou-Lhadj, and Ali Mehrabian. 2011. The concept of stratified sampling of execution traces. In *2011 IEEE 19th International Conference on Program Comprehension (ICPC)*. 225–226. https://doi.org/10.1109/ICPC.2011.17

[40] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing Data using t-SNE. *Journal of Machine Learning Research* (2008), 2579–2605. http://jmlr.org/papers/v9/vandermaaten08a.html

[41] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=rJXMpikCZ

[42] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 261–271. https://doi.org/10.1109/SANER48275.2020.9054857

[43] Xiang Wang, Xiangnan He, Meng Wang, Fuli Feng, and Tat-Seng Chua. 2019. Neural graph collaborative filtering. In *Proceedings of the 42nd international ACM SIGIR conference on Research and development in Information Retrieval (SIGIR)*. 165–174. https://doi.org/10.1145/3331184.3331267

[44] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning semantic program embeddings with graph interval neural network. *Proceedings of the ACM on Programming Languages* (2020), 1–27. https://doi.org/10.1145/3428205

[45] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering (ICSE)*. 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[46] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=ryGs6iA5Km

[47] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2018. Representation learning on graphs with jumping knowledge networks. In *International Conference on Machine Learning (ICML)*. 5453–5462. https://proceedings.mlr.press/v80/xu18c.html

[48] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy (S&P)*. 590–604. https://doi.org/10.1109/SP.2014.44

[49] 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 fifteenth International Conference on Architectural support for programming languages and operating systems (ASPLOS)*. 143–154. https://doi.org/10.1145/1736020.1736038

[50] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *2012 34th International Conference on Software Engineering (ICSE)*. 102–112. https://doi.org/10.1109/ICSE.2012.6227202

[51] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)* (2012), 1–28. https://doi.org/10.1145/2110356.2110360

[52] Tarannum Shaila Zaman, Xue Han, and Tingting Yu. 2019. SCMiner: localizing system-level concurrency faults from large system call traces. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 515–526. https://doi.org/10.1109/ASE.2019.00055

[53] 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 *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 783–794. https://doi.org/10.1109/ICSE.2019.00086

[54] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. 2019. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 807–817. https://doi.org/10.1145/3338906.3338931

[55] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. https://doi.org/10.1145/3132747.3132778

[56] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *12th {USENIX} Symposium on Operating Systems Design and Implementation (OSDI)*. 603–618. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhao

[57] 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)*. 459–469. https://doi.org/10.1109/ICSME.2019.00079

[58] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. 2015. Learning to log: Helping developers make informed logging decisions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*. 415–425. https://doi.org/10.1109/ICSE.2015.60