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

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

Jun Zeng  
National University of 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  
kaihang@comp.nus.edu.sg

Zhenkai Liang∗  
National University of 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.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ISSSTA ’22, July 18–22, 2022, Virtual, South Korea  
© 2022 Copyright held by the owner/author(s).  
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, 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].
protected QuorumServer findLeader() {
    QuorumServer leaderServer = null;
    Vote current = self.getCurrentVote();
    for (QuorumServer s : self.getView().values()) {
        if (s.id == current.getId()) {
            leaderServer = s;
            break;
        }
    }
    if (leaderServer == null) {
        LOG.warn("Couldn’t find the leader with id = 
            + current.getId());
    }
    return leaderServer;
}

We implement a prototype system, TeLL (Tell Log Levels), 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.

1https://github.com/apache/zookeeper/releases/tag/release-3.5.6
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 getCurrentVote, especially when dealing with the logic of getPeerMatching 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
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 parametrizes 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
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, recreateSocketAddressAddresses, in the FAST. Since most of logging statements in recreateSocketAddressAddresses 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.

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 $W_v \in \mathbb{R}^{S \times d}$ 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 $e_s$ by:

$$e_s = W_v^r x,$$

where $x$ is the one-hot encoding of the symbol $s$. As such, $e_s \in \mathbb{R}^d$ 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 $\rightarrow$ type $\rightarrow$ 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, TrLL 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:

$$e_s^{(l)} = \sigma((e_s^{(l-1)}||e_{N_s}^{(l-1)})W_e^r),$$

where $N_s$ is the set of neighbors of $s$.
where $e_s^{(l)} \in \mathbb{R}^{d^{(l)}}$ is the embedding of $s$ after stacking $l$ layers, $d^{(l)}$ presents the embedding size at $l$-th layer, and $e_s^{(0)}$ denotes the word2vec-initialized embedding, $W_s^{(l)} \in \mathbb{R}^{d^{(l-1)} \times d^{(l)}}$ is a trainable weight matrix to distill useful information from neighboring symbols; $||$ is the concatenation operation, and $\sigma$ is an activation function set as LeakyReLU [41]. $e_{s'}^{(l-1)}$ stands for the neural messages aggregated from $s$’s neighbors $N_s$, which is formulated as:

$$e_{s'}^{(l-1)} = \sum_{s' \in N_s} \frac{e_s^{(l-1)}}{\sqrt{|N_s|/|N_s|}},$$

(3)

where $1/\sqrt{|N_s|/|N_s|}$ 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 $s$, $\{e_s^{(0)}, \ldots, e_s^{(L)}\}$, which encode different-hops of relation information in the AST. Hence we adopt the layer-aggregation mechanism [47] to concatenate the representations of $s$ into a single vector, as follows:

$$e_s^l = e_s^{(0)} || \cdots || e_s^{(L)}.$$  

(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 $h_t$. Here we employ a pooling function over these symbol representations:

$$h_t = \rho([e_{i_1}^1, e_{i_2}^1, \ldots, e_{i_s}^k]),$$  

(5)

where $\rho$ is set as the average pooling function to delineate important features from all symbol representations; $e_{i_1}^1, e_{i_2}^1, \ldots, e_{i_s}^k$ are the representations of symbols $i_1, i_2, \ldots, i_s$, 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 $h_t$ of an AST $t$, TeLL updates its representation by combining neural messages from its neighboring blocks:

$$h_t^{(k)} = \sigma((h_t^{(k-1)} || \sum_{t' \in N_t} \frac{h_t^{(k-1)} W_{t'}}{|N_t|/|N_t|}) W_{p}^{(k)}),$$

(6)

where $h_t^{(k)} \in \mathbb{R}^{d^{(k)}}$ is the updated representation of $t$ at the $k$-th inter-block propagation layer, and $h_t^{(0)}$ is initialized as $h_t$ in Equation (5); $N_t$ is the set of blocks directly connecting to $t$ in a FAST, and $W_{p}^{(k)} \in \mathbb{R}^{d^{(k-1)} \times d^{(k)}}$ 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 $t - \{h_t^{(0)}, \ldots, h_t^{(K)}\}$. 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 $t$:

$$h_t^* = h_t^{(0)} || \cdots || h_t^{(K)}.$$  

(7)

Note that $h_t^*$ represents the embedding of not only an AST $t$ but a code block. As a result, our hierarchical GNNs (HGBN) 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:

$$q_m = \rho([z_{i_1}, z_{i_2}, \ldots, z_{i_N}]),$$  

(8)

where $z_{i_1}, z_{i_2}, \ldots, z_{i_N}$ are the word2vec-initialized embeddings of tokens $i_1, i_2, \ldots, i_N$ in log message $m$ and $N$ is the number of unique tokens; $\rho$ 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:

$$x_b = (h_t^* W_t || q_m W_p) W_f + b_f,$$  

(9)

where $W_t \in \mathbb{R}^{\sum_{k=0}^{K} d^{(k)} \times d}$, $W_p \in \mathbb{R}^{d \times d}$ and $W_f \in \mathbb{R}^{d \times 5}$ are the transformation matrices, and $b_f$ is the bias term.

To learn model parameters in HGBN, 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 HGBN’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 HGBN’s objective function.

In summary, TeLL adopts HGBN 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], and the rest of the FAST building (e.g., ICFG construction) is developed due to the different granularity of code blocks — we analyze the most fine-grained basic blocks. As the random sampling [39] is applied to split logged blocks to ensure that our statistics are not necessarily the same as existing studies been commonly adopted in recent log-related studies [5, 25–27].

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.

We collect nine widely used Java software systems from Datasets.

### Table 1: Log level distribution of nine software systems.

<table>
<thead>
<tr>
<th>Systems</th>
<th>Version</th>
<th>LB</th>
<th>TB</th>
<th>DB</th>
<th>WB</th>
<th>EB</th>
<th>FB</th>
<th>MIX</th>
<th>Hack (1.3%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra</td>
<td>3.11.4</td>
<td>325</td>
<td>202</td>
<td>280</td>
<td>228</td>
<td>280</td>
<td>0</td>
<td>100 (0.8%)</td>
<td></td>
</tr>
<tr>
<td>Elasticsearch</td>
<td>7.40</td>
<td>5,363</td>
<td>902</td>
<td>1,195</td>
<td>2,048</td>
<td>1,473</td>
<td>473</td>
<td>5</td>
<td>27 (1.3%)</td>
</tr>
<tr>
<td>Flink</td>
<td>1.8.2</td>
<td>2,475</td>
<td>35</td>
<td>793</td>
<td>815</td>
<td>455</td>
<td>352</td>
<td>6</td>
<td>190 (0.8%)</td>
</tr>
<tr>
<td>HBase</td>
<td>22.2</td>
<td>5,146</td>
<td>461</td>
<td>1,153</td>
<td>1,454</td>
<td>1,286</td>
<td>765</td>
<td>0</td>
<td>270 (0.5%)</td>
</tr>
<tr>
<td>Jmeter</td>
<td>5.30</td>
<td>1,762</td>
<td>1</td>
<td>654</td>
<td>295</td>
<td>406</td>
<td>401</td>
<td>0</td>
<td>50 (3.3%)</td>
</tr>
<tr>
<td>Kafka</td>
<td>2.30</td>
<td>1,426</td>
<td>199</td>
<td>414</td>
<td>335</td>
<td>171</td>
<td>293</td>
<td>0</td>
<td>14 (1.0%)</td>
</tr>
<tr>
<td>Karaf</td>
<td>4.2.9</td>
<td>698</td>
<td>15</td>
<td>152</td>
<td>192</td>
<td>151</td>
<td>185</td>
<td>6</td>
<td>30 (4.1%)</td>
</tr>
<tr>
<td>Wicket</td>
<td>8.6.1</td>
<td>408</td>
<td>11</td>
<td>167</td>
<td>45</td>
<td>88</td>
<td>96</td>
<td>0</td>
<td>10 (2.3%)</td>
</tr>
<tr>
<td>Zookeeper</td>
<td>3.6.6</td>
<td>1,496</td>
<td>33</td>
<td>225</td>
<td>599</td>
<td>372</td>
<td>262</td>
<td>0</td>
<td>5 (0.3%)</td>
</tr>
</tbody>
</table>

Average: 2,232 229 551 674 425 345 1 170 (0.8%)

Note: LB is the number of logged blocks. TB, DB, WB, FB, MIX refers 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: $\text{AOD} = \frac{\sum_{i=1}^{N} (\text{Dis}(a, s_i))/\text{MaxDis}(a_i))}{N}$, where $N$ is the number of logged blocks predicted by TeLL, $\text{Dis}(a, s_i)$ 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 $\text{MaxDis}(a_i)$ 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

---

\(^2\)https://tree-sitter.github.io/tree-sitter/
### Table 2: Comparison between TeLL and DeepLV in terms of Accuracy, AUC and AOD.

<table>
<thead>
<tr>
<th>Systems</th>
<th>Accuracy</th>
<th>AUC</th>
<th>AOD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DL(Syn)</td>
<td>DL(Comb)</td>
<td>TL(Mul)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cassandra</td>
<td>0.559</td>
<td>0.603 (+4.4%)</td>
<td>0.702 (+14.3%)</td>
</tr>
<tr>
<td>Elastisearch</td>
<td>0.528</td>
<td>0.609 (+8.1%)</td>
<td>0.696 (+16.8%)</td>
</tr>
<tr>
<td>Hbase</td>
<td>0.525</td>
<td>0.652 (+12.7%)</td>
<td>0.687 (+16.2%)</td>
</tr>
<tr>
<td>Jmeter</td>
<td>0.511</td>
<td>0.652 (+12.2%)</td>
<td>0.686 (+16.1%)</td>
</tr>
<tr>
<td>Kafka</td>
<td>0.507</td>
<td>0.621 (+8.4%)</td>
<td>0.664 (+15.5%)</td>
</tr>
<tr>
<td>Karaf</td>
<td>0.565</td>
<td>0.721 (+10.7%)</td>
<td>0.750 (+18.5%)</td>
</tr>
<tr>
<td>Wicket</td>
<td>0.573</td>
<td>0.638 (+6.5%)</td>
<td>0.692 (+17.1%)</td>
</tr>
<tr>
<td>Zookeeper</td>
<td>0.528</td>
<td>0.699 (+8.3%)</td>
<td>0.746 (+21.8%)</td>
</tr>
<tr>
<td>Average</td>
<td>0.540</td>
<td>0.611 (+7.9%)</td>
<td>0.670 (+17.0%)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Model</th>
<th>Encoding</th>
<th>Accuracy</th>
<th>AUC</th>
<th>AOD</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNN-NONE</td>
<td>Ordinal</td>
<td>0.517</td>
<td>0.840</td>
<td>0.733</td>
<td>3.459</td>
</tr>
<tr>
<td>NONE-GNN</td>
<td>Ordinal</td>
<td>0.602</td>
<td>0.875</td>
<td>0.796</td>
<td>1.780</td>
</tr>
<tr>
<td>ASTN-GNN</td>
<td>Ordinal</td>
<td>0.618</td>
<td>0.878</td>
<td>0.802</td>
<td>8.908</td>
</tr>
<tr>
<td>GNN-GNN</td>
<td>One-hot</td>
<td>0.652</td>
<td>0.848</td>
<td>0.810</td>
<td>4.688</td>
</tr>
<tr>
<td>GNN-GNN(HBGN)</td>
<td>Ordinal</td>
<td>0.678</td>
<td>0.893</td>
<td>0.833</td>
<td>4.713</td>
</tr>
</tbody>
</table>

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 schemes 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.
    ...
    switch (type) {
        case None: ...
        case NodeCreated: ...
        case NodeChildrenChanged: ...
        case NodeDataChanged: ...
    }
    ...
    switch (qp.getType()) {
        case NodeChildrenChanged: ...
        case NodeDataChanged: ...
        case NodeCreated: ...
        case NodeDeleted: ...
    }
    ...
}

protected void syncWithLeader(long newLeaderZxid) throws Exception {
    switch (type) {
        case Leader.COMMIT: ...
        case Leader.COMMITANDACTIVATE: ...
        case Leader.PROPOSAL: ...
        case NodeCreated: ...
        case NodeChildrenChanged: ...
        default: ...
    }
    ...
    switch (qp.getType()) {
        case NodeChildrenChanged: ...
        case NodeDataChanged: ...
        case NodeCreated: ...
        case NodeDeleted: ...
    }
    ...
}

String msg = "Unhandled watch event type " + type + " on path " + clientPath; ...

LOG.error(msg);
throw new RuntimeException(msg);
return result;

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

lem 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 discovery 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.

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.

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 CCDC 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**


