DeepLog: Deep-Learning-Based Log Recommendation

Yang Zhang, Xiaosong Chang, Lining Fang, Yifan Lu
School of Information Science and Engineering,
Hebei University of Science and Technology
Shijiazhuang, Hebei, China
zhangyang@hebust.edu.cn,cxiaosong@foxmail.com,{fanglining1228,hn_luyifan}@163.com

ABSTRACT

Log recommendation plays a vital role in analyzing run-time issues including anomaly detection, performance monitoring, and security evaluation. However, existing deep-learning-based approaches for log recommendation suffer from insufficient features and low F_1. To this end, this paper proposes a prototype called *DeepLog* to recommend log location based on a deep learning model. *DeepLog* parses the source code into an abstract syntax tree and then converts each method into a block hierarchical tree in which *DeepLog* extracts both semantic and syntactic features. By doing this, we construct a dataset with more than 110K samples. *DeepLog* employs a double-branched neural network model to recommend log locations. We evaluate the effectiveness of *DeepLog* by answering four research questions. The experimental results demonstrate that it can recommend 8,725 logs for 23 projects and the F_1 of *DeepLog* is 28.17% higher than that of the existing approaches, which improves state-of-the-art.

CCS CONCEPTS

• **Software and its engineering → Software evolution**.

KEYWORDS

Log location, Deep learning, Recommendation, Similarity analysis

ACM Reference Format:
Yang Zhang, Xiaosong Chang, Lining Fang, Yifan Lu. 2023. DeepLog: Deep-Learning-Based Log Recommendation . In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Logging is a popular practice to generate run-time information in software systems. Practitioners rely on logging to analyze some running issues including error tracing, anomaly detection, performance monitoring, and security evaluation. In a recent study by Li et al. [13], developers require to handle hundreds of GB of logging records generated by real-world applications every day. Therefore, it is necessary to determine where to insert logging information, especially for large-scale industrial software.

How to log efficiently gains increasing concerns. On one hand, although the absence of a log has no impact on the correctness of programs, inefficient logs hinder developers to monitor the run-time issues. Consequently, they probably miss the opportunities to conduct some optimization on critical problems. On the other hand, redundant logs burden software maintenance and challenge the management task of a large amount of logging data.

Recommending logs has become mainstream. In a recent survey by Gholamian et al. [1], they identify the top 5 popular research topics of log recommendation including log location, log printing automation, log mining, log maintenance, and log parsing. Many logging tasks will benefit from reasonable and accurate log recommendations. Consequently, it becomes more important to find an appropriate log location compared to those downstream research.

More and more researchers are investigating how to recommend logs. Most of them leverage rule-based approach [2][16][17], heuristic-based approach [3][18][19], statistic-based approach [4][5][20][21][22]. Specifically, the rule-based approach identifies log locations by relying on static analysis to summarize the code patterns. However, choosing such code patterns often requires expertise. Ignoring those important code patterns may harm the accuracy of log location recommendations. The heuristic-based approach leverages heuristics and static analysis to recommend log locations. However, for a specific task, it is hard to recommend logs with multiple objectives since making logging decisions needs lots of consideration. In recent years, machine learning and deep learning methods produce efficient results in various domains[14][15], and many researchers begin to recommend log location by leveraging machine learning or deep learning techniques (such as decision-tree[4], LSTM[5]). The statistic-based approach first maps features of the source code into a latent vector space and then builds a dataset for training. This approach produces efficient results. However, due to the difficulty of combining multiple features, and the logging practices being different in different projects, the trained model always has bias.

Although these lately-proposed deep learning approaches are effective in log recommendation, several problems remain open and need to be improved. Firstly, existing works are prone to leveraging single features. Few works extract semantic, syntactic, and structural features from the source code. Secondly, considering a scenario of cross-project logging prediction, there is still a lack of guidance to decide which project is similar to the target project and which should be selected to build the training dataset. As a result, the logging customization learned by the deep models can be biased. Thirdly, most log recommendation tools employ machine learning models. The works based on the deep learning model need to be further developed.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

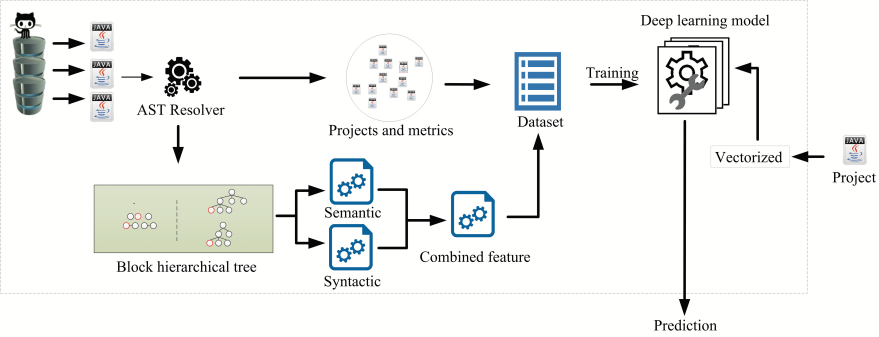

Figure 1: Overview of DeepLog

To address these problems, this paper proposes a novel approach called *DeepLog* to recommend log location based on the deep learning model. *DeepLog* parses the source code into an abstract syntax tree (AST) and then converts each method into a block hierarchical tree (BHT) in which *DeepLog* extracts both semantic and syntactic features. By doing this, we construct a dataset with more than 110K samples. *DeepLog* employs a double-branched neural network model to recommend log locations. We evaluate the effectiveness of *DeepLog* by answering four research questions. The experimental results demonstrate that it can recommend 8,725 logs for 23 projects and the F_1 of *DeepLog* is 28.17% higher than that of the existing approaches, which improves state-of-the-art. The main contribution of this paper can be summarized as follows:

- We propose a method to extract both semantic and syntactic features from block hierarchical tree.
- Similarity analysis is leveraged to build the training dataset under a cross-project scenario.
- We propose a double-branched deep learning model and implement a prototype *DeepLog* as an Eclipse plug-in.
- We evaluate *DeepLog* on 23 projects and compare it with existing approaches, demonstrating its effectiveness.

2 DESIGN

The design of *DeepLog* is depicted in Figure 1. To generate a training dataset, we select 23 projects from GitHub. By traversing AST generated from each project, a block hierarchical tree is built for each method. Both syntactic and semantic features are extracted from BHT and then combined. By using similarity analysis, we select samples to build the dataset employed to train a neural network-based classifier. The output of this classifier indicates whether a code block should be logged or not. The main components of *DeepLog* are presented in the following sections. We evaluate *DeepLog* on 23 projects and compare it with existing approaches.

2.1 Sample generation

In this section, we first introduce the projects used to generate the samples and the features generated from the block hierarchical tree. Then, we present the similarity analysis method and the final dataset.

2.1.1 *Project selection.*

To generate samples, we select 23 real-world projects that are commonly used in prior log-related studies[5][8]. These projects and their configurations are presented in Table 1 which presents the version (column 2), thousands of lines of code(column 3), the number of logging statements (column 4), the number of logged and non-logged blocks (columns 5-6), and the percentage of logged blocks over all the blocks (column 7).

Table 1: Projects and their configurations

Project	Version	KLOC	NOL	#LB	#NLB	%LB
Angel	3.2.0	154	1.1K	0.9K	30.3K	2.97%
Beam	2.39.0	210	1.6K	1.3K	42.2K	3.08%
Cassandra	4.1.0	205	2.3K	1.9K	47.8K	3.97%
Druid	4.0.0	265	2.9K	2.1K	45.1K	4.66%
Dubbo	4.0.0	66	0.9K	0.8K	19.7K	4.06%
Elasticsearch	8.5.0	630	5.5K	4.2K	146.4K	2.87%
Flink	4.0.0	458	3.4K	2.9K	92.2K	3.15%
Groovy	5.0.0	126	0.1K	0.1K	36.9K	0.27%
Hadoop	3.4.0	619	12.9K	11.3K	142.6K	7.92%
Hbase	3.0.0	321	6.9K	5.4K	85.4K	6.32%
Incubator	2.4.3	25	0.5K	0.4K	8.0K	5.00%
Jmeter	5.5.0	69	1.8K	1.6K	16.6K	9.64%
Kafka	3.3.0	132	2.8K	2.2K	30.3K	7.26%
Pulsar	2.11.0	161	4.5K	3.4K	28.3K	12.01%
Rocketmq	4.9.4	54	1.5K	1.2K	12.7K	9.45%
Elasticjob	3.1.0	6	0.1K	0.1K	2.0K	5.00%
Shardingsphere	5.1.1	53	0.3K	0.3K	18.2K	1.65%
Skywalking	9.1.0	35	0.5K	0.4K	7.1K	5.63%
Storm	2.5.0	164	3.3K	2.5K	46.6K	5.36%
Tomcat	10.1.0	200	2.6K	2.3K	48.1K	4.78%
Wicket	10.0.0	87	0.4K	0.4K	18.7K	2.14%
Zeppelin	0.11.0	83	2.2K	1.8K	21.8K	8.26%
Zookeeper	3.9.0	39	1.3K	1.0K	8.6K	11.63%

2.1.2 *Feature selection.*

Prior works [4][10] indicate that making decisions where to log is impacted by several factors including the type of code block, the complexity of code, and the number of nested blocks in methods, etc. To reflect the depth of the code block, we build a BHT based on inclusion relationships between blocks. We extract the semantic and syntactic features from BHT.

The syntactic feature is extracted from methods by using Javaparser[6]. The node type of every statement is recorded and is

called a *token*. We generate a list of words according to the node type.

Semantic features are extracted from the textual information of code blocks. Prior studies revealed that code information (such as variable name) may capture the semantic information from the source code [5]. Therefore, we handle variable names and invoking methods in the block as plain text. To get the semantic features and reduce noising information, we split words by following camel case or underscore naming convention[9], then filter stop words. Finally, we count repetitive words in a *token* list.

Both syntactic and semantic features are merged into a fixed-length continuous vector by using Keras embedding[11]. When handling syntactic features using the *Keras tokenizer*, we decide to take the top 5,000 words and construct vectors of the maximum length of 300. The maximum length is changed to 500 for semantic features since the number of semantic features is more than that of syntactic features. After extracting the features from nodes in a BHT, we can combine the feature information from leaf to root to add the structure information. Note that we limit the max depth from leaf to root to 10. The combined syntactic and semantic features are taken as the final vector to build the dataset.

2.1.3 *Similarity analysis.*

To recommend log locations in a cross-project scenario, similarity analysis (SA) is employed to find a similar project to generate the training dataset. SA calculates the similarity between projects by evaluating the metrics extracted by CK[7]. We select those important metrics (6 out of 46 metrics) for similarity analysis. Cosine distance with these metrics is employed to calculate the project similarity score, and then we choose the best similarity project to build the training dataset and execute model training.

2.1.4 *Label identification.*

All samples are labeled by judging that a code block contains at least one logging statement, such as `trace`, `debug`, `info`, `warn`, and `error`. The identification of the logging statement is automatically conducted by leveraging Javaparser[6].

2.1.5 *Dataset.*

The final dataset is composed of 118,800 samples after extracting features from 23 projects and then applying the under-sampling technique to mitigate the impact of data imbalance. We split the sample into three parts: training(60%), validation(20%), and testing(20%).

2.2 Deep learning model

We build the deep learning model under Keras framework[11]. The structure of the model is presented in Figure 2. Both semantic and syntactic features are separately fed into each branch of this model. The structure of DeepLog's model can prevent these features from interweaving with each other and reduce the dimension. The layers `Convolution1D`, `MaxPooling1D`, `Flatten`, `Embedding`, and `GlobalMaxPool1D` are utilized to reduce noising information and feature dimension. The semantic and the syntactic outputs are merged by the layer `concatenate`. The layer `dropout` with a 0.4 dropout rate reduces over-fitting and immoderate reliance on the trained projects. The following layers `Dense` with 128 neurons and *output* with 2 neurons will map the combined vectors into a single output that indicates whether a code block should be logged or not. We use the search strategy to find the optimal hyperparameters[14].

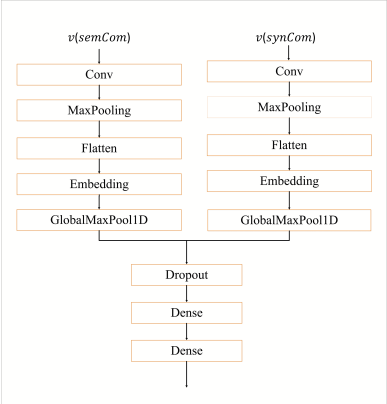

Figure 2: The neural network architecture of *DeepLog*

2.3 Implementation

We implement a prototype *DeepLog* leveraging Keras[11], and the prototype is integrated as a plug-in of Eclipse. *DeepLog* generates the prediction result after feeding vectored features into DeepLog's neural network.

3 EVALUATION

All experiments are conducted on a workstation with a 3.6GHz Intel Core i7 CPU and 8GB RAM. The machine runs Ubuntu 16.04.7 and has Python 3.8 and TensorFlow 2.7.0 installed.

We evaluate the effectiveness of *DeepLog* to address the following research questions. For the reproducibility of the evaluation, all models, datasets, and plug-in are available at *https://uzhangyang.github.io/research/deeplog.html*

RQ1 How applicable is *DeepLog* in log recommendation?
RQ2 How effective is SA in *DeepLog*?
RQ3 Is *DeepLog* with both syntactic and semantic features better than that with single feature?
RQ4 Does *DeepLog* outperform existing deep-learning-based approaches in log recommendation?

3.1 Results for RQ1

To answer RQ1, we present the number of logs recommended by DeepLog and the performance of each project. The experimental results are summarized in Table 2 which presents the numbers of positive samples in the test dataset (column 2) and the number of logs predicted by *DeepLog* (column 3). As shown in Table 2, F_1 ranges from 72.73% to 89.91% for different projects. A total of 8725 out of 9562 logs is predicted, which shows that *DeepLog* is applicable in log recommendation.

3.2 Results for RQ2

To answer RQ2, we compare the accuracy of *DeepLog* with SA against that without SA under a cross-project scenario. We replicate five times on each project and get the average accuracy. This

Table 2: Evaluation results of each project

Project	ALB	CLB	Accuracy	Precision	Recall	F_1
Angel	175	158	89.18%	88.27%	90.29%	89.27%
Beam	267	238	84.22%	81.51%	89.14%	85.15%
Cassandra	375	326	83.59%	80.69%	86.93%	83.70%
Druid	415	391	88.40%	83.55%	94.22%	88.56%
Dubbo	159	131	86.25%	88.51%	82.39%	85.34%
Elasticsearch	833	793	88.91%	84.91%	95.18%	89.75%
Flink	550	491	85.65%	82.24%	89.27%	85.61%
Groovy	21	19	87.83%	82.61%	90.48%	86.36%
Hadoop	2186	1962	80.74%	76.16%	89.75%	82.40%
Hbase	1086	1025	85.58%	80.20%	94.38%	86.72%
Incubator	75	66	88.44%	86.84%	88.00%	87.42%
Jmeter	319	282	82.75%	79.21%	88.40%	83.56%
Kafka	422	393	79.79%	71.85%	93.13%	81.11%
Pulsar	695	634	82.41%	78.56%	91.22%	84.42%
Rocketmq	235	211	85.11%	82.10%	89.79%	85.77%
Elasticjob	21	16	75.13%	69.57%	76.19%	72.73%
Shardingsphere	49	49	90.52%	81.67%	100.00%	89.91%
Skywalking	86	78	79.83%	74.29%	90.70%	81.68%
Storm	523	496	86.97%	82.94%	94.84%	88.49%
Tomcat	450	408	85.64%	82.09%	90.67%	86.17%
Wicket	64	55	78.03%	70.51%	85.94%	77.46%
Zeppelin	344	322	88.38%	84.29%	93.60%	88.71%
Zookeeper	212	181	76.02%	72.69%	85.38%	78.52%
Average	416	379	84.32%	80.23%	90.00%	84.73%

Table 3: Evaluation of similarity analysis

Project	Without SA	With SA	Diff
Angel	71.24%	77.52%	6.28%
Beam	72.96%	72.79%	-0.17%
Cassandra	74.80%	78.62%	3.82%
Druid	71.97%	73.79%	1.83%
Dubbo	77.10%	76.88%	-0.22%
Elasticsearch	69.26%	80.27%	11.02%
Flink	75.67%	73.49%	-2.18%
Groovy	70.91%	69.59%	-1.32%
Hadoop	70.59%	73.53%	2.94%
Hbase	76.96%	79.40%	2.44%
Incubator	76.56%	76.71%	0.15%
Jmeter	72.67%	75.22%	2.55%
Kafka	76.62%	76.58%	-0.04%
Pulsar	70.19%	73.90%	3.70%
Rocketmq	71.10%	73.03%	1.93%
Elasticjob	71.16%	75.45%	4.30%
Shardingsphere	71.01%	77.05%	6.04%
Skywalking	69.24%	71.81%	2.57%
Storm	76.41%	82.34%	5.93%
Tomcat	71.23%	75.87%	4.64%
Wicket	75.00%	76.39%	1.39%
Zeppelin	77.29%	83.74%	6.45%
Zookeeper	69.55%	72.81%	3.26%
Average	73.02%	75.95%	2.93%

method can remission the error caused by the randomness in the training phrase. The experimental results are tabulated in Table 3. The accuracy of DeepLog with SA is 2.93% higher than that of DeepLog without SA. We should note that the accuracy has slightly decreased in *Beam*, *Dubbo*, *Flink*, *Groovy*, and *Kafka*, this is due to the project found by SA not always being the best. Thus we will investigate more efficient metrics set in future work. Furthermore, we perform a Wilcoxon rank sum test and effect size (Cliff's Delta) to evaluate the significance of the difference[12]. Our results indicate that the similarity analysis method can provide a statistically significant improvement in the ability of the model at cross-project predict scenarios with a Wilcoxon rank value of p=1.2779e-04 and with effect size of as large as 0.4896.

3.3 Results for RQ3

To answer RQ3, we compare the performance of DeepLog with both syntactic and semantic features against that with a single feature. Figure 3 presents the results. The F_1 based on both syntactic and semantic features is 7.49% and 4.09% higher than that based on syntactic features and that based on semantic features, respectively. The results show that *DeepLog* with both features is better than that with a single feature.

3.4 Results for RQ4

To answer RQ4, we compare DeepLog against Li et al.'s work [5]. Evaluation results are presented in Table 4. DeepLog obtains an average of 83.27% F_1 while the average F_1 of Li et al. [5] is 55.10%, which improves F_1 by 28.17%. The experimental results show that DeepLog outperforms existing approaches.

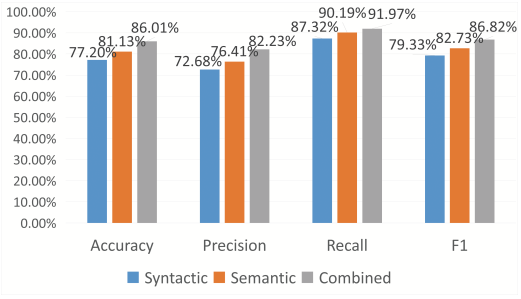

Figure 3: DeepLog with different features

Table 4: Comparison between DeepLog with Li et al.'s work[5]

Project	Li et al.[5]				DeepLog			
	Accuracy	Precision	Recall	F_1	Accuracy	Precision	Recall	F_1
Cassandra	83.00%	51.70%	56.60%	54.00%	**83.59%**	**80.69%**	**86.93%**	**83.70%**
Elasticsearch	81.90%	52.00%	55.60%	52.90%	**88.91%**	**84.91%**	**95.18%**	**89.75%**
Flink	83.00%	58.90%	70.90%	64.30%	**85.65%**	**82.24%**	**89.27%**	**85.61%**
Hbase	80.50%	56.10%	63.40%	59.50%	**85.58%**	**80.20%**	**94.38%**	**86.72%**
Kafka	74.40%	41.50%	58.20%	47.30%	**79.79%**	**71.85%**	**93.13%**	**81.11%**
Wicket	**84.70%**	45.70%	72.30%	56.00%	78.03%	**70.51%**	**85.94%**	**77.46%**
Zookeeper	72.90%	48.30%	55.60%	51.70%	**76.02%**	**72.69%**	**85.38%**	**78.52%**
Average	80.06%	50.60%	61.80%	55.10%	**82.51%**	**77.58%**	**90.03%**	**83.27%**

4 CONCLUSIONS

In this paper, we propose a prototype called *DeepLog* to recommend log location based on the deep learning model. We construct a dataset with more than 110K samples and employ a double-branched neural network model to recommend log locations. The experimental results demonstrate that *DeepLog* can recommend 8,725 logs for 23 projects and the F_1 of *DeepLog* is 28.17% higher than that of the existing approaches, which improves state-of-the-art.

REFERENCES

[1] S. Gholamian and P. A. J. a. p. a. Ward, "A Comprehensive Survey of Logging in Software: From Logging Statements Automation to Log Mining and Analysis," 2021.

[2] Z. Jia, S. Li, X. Liu, X. Liao, and Y. Liu, "SMARTLOG: Place error log statement by deep understanding of log intention," in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 61-71: IEEE.

[3] K. Yao, G. B. de Pádua, W. Shang, S. Sporea, A. Toma, and S. Sajedi, "Log4perf: Suggesting logging locations for web-based systems' performance monitoring," in Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, 2018, pp. 127-138.

[4] Q. Fu et al., "Where do developers log? an empirical study on logging practices in industry," presented at the Companion Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 2014. Available: https://doi.org/10.1145/2591062.2591175

[5] Z. Li, T.-H. Chen, and W. Shang, "Where shall we log? studying and suggesting logging locations in code blocks," Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Virtual Event, Australia, 2020. Available: https://doi.org/10.1145/3324884.3416636

[6] JavaParser, https://javaparser.org/, 2019, last accessed July 1 2020.

[7] M. Aniche, 'Java code metrics calculator (CK)', 2015. https://github.com/mauricioaniche/ck

[8] Z. Li, H. Li, T.-H. P. Chen, and W. Shang, "Deeplv: Suggesting log levels using ordinal based neural networks," in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 1461-1472: IEEE.

[9] K. Liu et al., "Learning to Spot and Refactor Inconsistent Method Names," presented at the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019.

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

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

[12] S. Locke, H. Li, T.-H. Chen, W. Shang, and W. Liu, "LogAssist: Assisting Log Analysis Through Log Summarization," IEEE Transactions on Software Engineering, vol. 48, no. 9, pp. 3227-3241, 2022.

[13] H. Li, W. Shang, B. Adams, M. Sayagh, and A. E. Hassan, "A Qualitative Study of the Benefits and Costs of Logging From Developers' Perspectives," IEEE Transactions on Software Engineering, vol. 47, no. 12, pp. 2858-2873, 2021.

[14] Yang Zhang, Chuyan Ge, Shuai Hong, Ruili Tian, Chunhao Dong, Jingjing Liu. DeleSmell: Code Smell Detection Based on Deep Learning and Latent Semantic Analysis. Knowledge-Based Systems. 2022. 255: 1-12

[15] Yang Zhang, Chunhao Dong. MARS: Detecting Brain Class/Method Code Smell Based on Metric-Attention Mechanism and Residual Network. Journal of Software: Evolution and Process. 2021. E2403. 1-15

[16] D. Yuan et al., "Be conservative: Enhancing failure diagnosis with proactive logging," in 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), 2012, pp. 293-306.

[17] M. Cinque, D. Cotroneo, and A. Pecchia, "Event logs for the analysis of software failures: A rule-based approach," IEEE Transactions on Software Engineering, vol. 39, no. 6, pp. 806-821, 2012.

[18] R. Ding et al., "Log2: A Cost-Aware Logging Mechanism for Performance Diagnosis," in 2015 USENIX annual technical conference (USENIX ATC 15), 2015, pp. 139-150.

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

[20] S. Gholamian and P. A. J. a. p. a. Ward, "Borrowing from Similar Code: A Deep Learning NLP-Based Approach for Log Statement Automation," 2021.

[21] J. Cândido, J. Haesen, M. Aniche, and A. van Deursen, "An Exploratory Study of Log Placement Recommendation in an Enterprise System," in 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), 2021, pp. 143-154: IEEE.

[22] A. Mastropaolo, L. Pascarella, and G. Bavota, "Using deep learning to generate complete log statements," in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 2279-2290.