LoGenText: Automatically Generating Logging Texts Using Neural Machine Translation

Zishuo Ding
Concordia University
Montreal, Canada
zi_ding@encs.concordia.ca

Heng Li
Polytechnique Montréal
Montreal, Canada
heng.li@polymtl.ca

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

***Abstract*—The textual descriptions in logging statements (i.e., logging texts) are printed during system executions and exposed to multiple stakeholders including developers, operators, users, and regulatory authorities. Writing proper logging texts is an important but often challenging task for developers. However, despite extensive research on automated logging suggestions, research on suggesting logging texts rarely exists. In this paper, we present *LoGenText*, an automated approach that generates logging texts by translating the related source code into short textual descriptions. *LoGenText* takes the preceding source code of a logging text as the input and considers other context information such as the location of the logging statement, to automatically generate the logging text using neural machine translation models. We evaluate *LoGenText* on 10 open-source projects, and compare the automatically generated logging texts with the developer-inserted logging texts in the source code. We find that *LoGenText* generates logging texts that achieve BLEU scores of 23.3 to 41.8 and ROUGE-L scores of 42.1 to 53.9, which outperforms the state-of-the-art approach by a large margin. In addition, we perform a human evaluation involving 42 participants, which further demonstrates the quality of the logging texts generated by *LoGenText*. Our work is an important step towards automated generation of logging statements, which can potentially save developers' efforts and improve the quality of software logging.**

I. Introduction

Developers insert logging statements in the source code to collect valuable runtime information of software systems. Logging statements produce execution logs at runtime, which play important roles in the daily tasks of developers and other software practitioners [1], [2]. An example logging statement from HBase, *LOG.warn("Failed to create dir {}", dst)*, has a verbosity level of *warn*, a dynamic variable *dst* whose value will be dynamically determined, and a logging text *Failed to create dir* which will be directly outputted during software execution. Prior work has leveraged the rich information in logs to support different software engineering activities, including system comprehension [3], [4], anomaly detection [5], [6], [7], [8], [9], and failure diagnosis [10], [11]. In particular, logs are usually the only available resource for diagnosing field failures [12].

Extensive prior research has shown that writing proper logging statements is an important and challenging task [13], [14], [15], [16]. Besides the typical challenges of deciding where to log [17] and how to choose verbosity levels [18], deciding the textual information in the logging statement is even more challenging [19]. Prior studies find that developers spend significant efforts modifying the textual information in their logging statements [13], [20], [14], [15], [16]. A recent study has shown that developers rely heavily on reading the text in the logging statement while misleading textual information often makes the use of logs counterproductive [2].

Despite the importance of logging texts, there exists rare research effort that devotes to assisting developers in writing logging texts. A recent study by He et al. [19] proposes an approach that reuses the texts in the logging statements from similar code snippets. However, since only existing logging texts are directly reused, the texts generated by the prior approach may still require significant revisions by practitioners. Nevertheless, prior work [19] has demonstrated the potential possibility of automatically generating logging texts.

In order to help developers address the challenges of writing logging texts, we propose *LoGenText*, a neural-machine-translation based approach. *LoGenText* automatically generates the textual description of a logging statement by translating the related source code into logging texts. Specifically, we adopt a Transformer-based Sequence-to-Sequence model which leverages an encoder-decoder architecture to automate translations and uses the attention mechanism to boost its performance [21]. In *LoGenText*, the target sequence of the Transformer-based model is a logging text, and the source sequence is its related source code. We consider the source code preceding the logging text as the source input. We also consider incorporating other contexts that may provide relevant information about the logging texts to be generated, including the location of the logging statement, the succeeding source code, and the logging texts in similar code snippets. To incorporate such contexts, we further extend the Transformer by adding additional encoders that integrate the context information into the model [22]. The outputs of these encoders are then formed as a new input to the decoder which generates the logging text as the final output of *LoGenText*.

We evaluate *LoGenText* on 10 open-source Java projects from different domains. We first evaluate the automatically generated logging texts by comparing them with the original logging texts inserted by developers using quantitative metrics such as BLEU and ROUGE-L. *LoGenText* achieves BLEU scores of 23.3 to 41.8 and ROUGE-L scores of 42.1 to 53.9, which outperforms the baseline approach from prior research [19] by a large margin. On the other hand, our evaluation results show that incorporating other context information (e.g., the location of the logging statement) can further improve *LoGenText*. In order to further understand the effectiveness of *LoGenText*, we conduct a human-based evaluation that involved 42 participants. The results confirm that *LoGenText* can provide high-quality logging texts and it significantly outperforms the baseline approach in generating logging texts.

The contributions of this paper include:

- Our automated approach *LoGenText* significantly improves the state-of-the-art in generating logging texts.
- Our work suggests that automated approaches for logging text generation should not only focus on the preceding code of a logging statement, but also consider other context information to further improve the performance.
- Our work demonstrates the promising direction of leveraging advances in neural machine translation techniques to generate logging texts.

Our work is an important step towards automated generation of logging statements. Our findings shed light on future research opportunities that apply up-to-date neural machine translation techniques in automated generation and suggestion of logging statements. We share our extracted datasets from the 10 open-source projects and the configurations used for training our models[1].

Paper Organization. Section II presents the details of our approach *LoGenText*. Section III presents the setup of the experiment for evaluating *LoGenText*. Section IV and Section V present the results of evaluating *LoGenText* through quantitative metrics and human evaluation. Section VI discusses threats to the validity. Section VII presents the related work. Finally, Section VIII concludes the paper.

II. Approach

In this section, we describe the details of *LoGenText* that leverages neural machine translation (NMT) to automatically generate logging texts.

A. Approach Overview

LoGenText is a NMT-based approach that uses deep neural networks to translate source code into logging texts. Figure 1 illustrates the overall approach of *LoGenText*. First, for each logging statement in the source code, *LoGenText* extracts its logging text, the source code preceding the logging text (i.e., the pre-log code), and the context information from the source code (i.e., **data preparation**). Then, *LoGenText* feeds the extracted logging text, the pre-log code (i.e., the source), and the context information into a Transformer-based Sequence-to-Sequence (Seq2Seq) model [21] that consists of embedding layers, encoders, and decoders (i.e., **model training**). Finally, the trained model takes the source (the pre-log code) and the context information as input and translates it into the corresponding logging text (i.e., **model inference**). We detail each of these three steps in the rest of this section.

In the base form of *LoGenText*, we use the pre-log code of a logging statement to generate its logging text. We evaluate the base form of *LoGenText* in RQ1 (Section IV-RQ1). In RQ2 (Section IV-RQ2) and RQ3 (Section IV-RQ3), we propose a context-aware form of *LoGenText* and discuss the impact of adding the context information, including the location of the logging statement in the abstract syntax tree (AST) (i.e., the structural (AST) context), the source code succeeding the logging statement (i.e., the post-log code), and the logging text in the most similar code snippet, on the performance of *LoGenText*. The pre-log code is fed as the *source*, while other context information is fed as the *context* to the model.

[1]Data package: https://github.com/conf-202x/experimental-result

B. Data Preparation

LoGenText takes three parts of information about a logging statement to train the model: 1) *logging text*, which refers to the static plain text in the logging statement, 2) *source*, which contains the pre-log code, and 3) *context*, which includes the structural (AST) context, the post-log code, and the logging text in similar code.

1) Extracting the logging text: We first extract the complete logging message (including the logging text and variables) from the logging statement. Since our focus is on the logging text, we then replace the variables with a wildcard (<*vid*>). For example, given the following logging statement from Hadoop, the extracted logging text is "Removed child queue: <vid>".

```
// Original logging statement:
   LOG.debug("Removed child queue: {}",
        cs.getQueueName())
// Extracted logging text:
   "Removed child queue: <vid>"
```

2) Extracting the source data: We use the pre-log code as the main input (i.e., the *source* data) for our neural translation model. Specifically, the *source* data includes the code from the method start point to the location right before the logging text of the logging statement. We consider the pre-log code as our main input for logging text generation because a logging statement usually communicates the runtime behavior of the system before the execution of the logging statement [23], [2].

3) Extracting the context data: We consider three types of data as the *context* input of our neural translation model, including the structural (AST) context, the post-log code context, and the logging text in similar code. We discuss the details of extracting the structural context and the post-log code context in RQ2 where we discuss the impact of such contexts. Similarly, we discuss the details of extracting the logging text in similar code in RQ3.

4) Pre-processing the logging text and source data: Following the previous approaches for pre-processing the input text data [19], [24], [25], we convert the logging text and source code text into lower cases and tokenize them into token units. We also remove all the non-identifiers (e.g., quotation marks).

A potential challenge is the out-of-vocabulary (OOV) tokens of the source code and logging texts [26], [27]. At testing time, there would be tokens that have never occurred in the training data, which may lead to the poor translation of the NMT systems [28]. One way to alleviate the OOV problem is to enlarge the dictionary size to include more rare tokens. However, due to the fact that user-defined identifiers

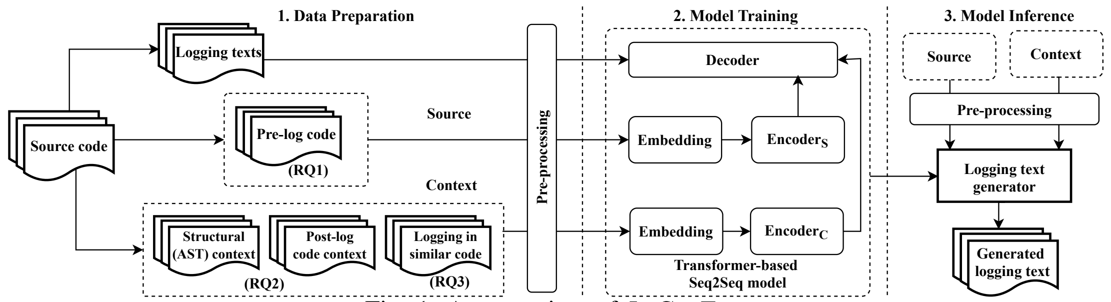


Fig. 1: An overview of *LoGenText*.

(i.e., not reserved by the programming language) take up the majority of code tokens, they have a non-negligible influence on the vocabulary of translation dictionary [27]. Thus, using a large dictionary to cover the user-defined tokens would increase the difficulty of training the translation model, as it requires more training data and hardware resources [27]. To address this problem, we employ byte pair encoding (BPE), a data compression technique, to segment the code tokens into subword units [29], [30]. This is based on the intuition that users often define identifiers via combining smaller word units. For example, the token "getQueueName" is a combination of three subwords, i.e., "get", "queue" and "name". In this way, our approach can encode all tokens as sequences of subword units.

We set the maximum length of both the logging text sequences and the source code sequences to 1,024 (the default value of our Transformer-based model). The tokens of the sequences beyond the maximum length will be truncated; the sequences shorter than the maximum length are padded. 0% of the logging text sequences are truncated and 3.7% to 3.8% of the source code sequences are truncated in the studied projects.

C. NMT-based Log Generation

In our approach, we consider the logging text generation task as a machine translation task, i.e., translating a code snippet into logging text that communicates the internal behavior of the code snippet. Thus, we can apply neural machine translation (NMT) techniques to solve the logging text generation problem. Formally, given a source sequence $X = (x_1, x_2, \ldots, x_S)$, our goal is to predict tokens in the target logging text $Y = (y_1, y_2, \ldots, y_T)$. Most NMT models use an encoder-decoder architecture. The input to the encoder is the source sequence X, and the output of the encoder is a sequence of distributed representations. The generated representations are then fed into the decoder part, where the tokens in the target sequence are generated one by one [31]. Hence, the objective of the models is to approximate the conditional distribution $\log P\left(Y|X;\theta\right)$ over the given source target pairs and model parameters θ.

Our model is also based on an encoder-decoder model, in particular, the Transformer model proposed by Vaswani et al. [21], which has shown outstanding performance in many software engineering tasks (e.g., source code summarization [32] and code completion [33]). Figure 2 illustrates the structure of the Transformer translation model that is implemented in *LoGenText*. Like many other sequence to sequence models, the Transformer utilizes an encoder-decoder structure, which is explained in detail in the rest of this section.

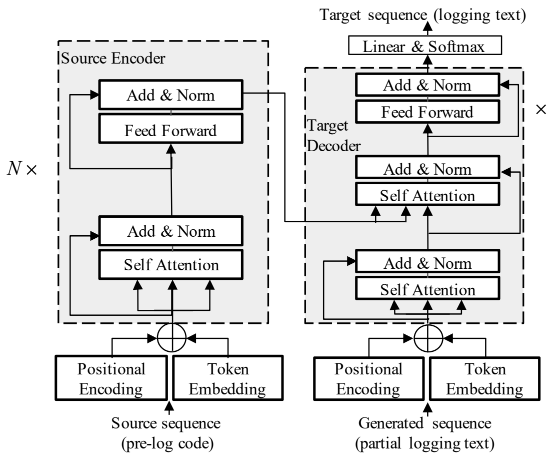


Fig. 2: An overview of the Transformer translation model.

Source encoder: As Figure 2 shows, the source encoder component makes use of N stacked layers. Each layer is broken down into two sub-layers. The first sub-layer is a self attention layer:

$$Attention\left(Q,K,V\right) = softmax(\frac{QK^T}{\sqrt{d_k}})V \qquad (1)$$

where Q, K, V are the query, key, and value vectors, $\sqrt{d_k}$ is a normalization factor where d_k is the dimension of the key/query vector, $Attention$ is the output of the attention layer. The self attention mechanism allows the model to look at other positions for extra information while encoding the current position.

The residual connection and layer normalization are then applied on the output of the attention layer:

$$LayerNorm\left(Attention + X\right) \qquad (2)$$

where X is the vector representation of the input token after positional encoding (explained in the next paragraph). The output is then fed to the second sub-layer, a fully connected feed forward network. Note that the feed forward network is point-wise, which means the network is applied independently to individual vectors generated by the attention layer.

Positional encoding: The orders of the tokens in the source sequence are important for a machine translation model. To address this, unlike RNN and its variances, Transformer adopts positional encoding to inject the relative positional information into the token representations. Specially, a positional vector is added to the input embedding, where the positional vector pe for tth token is calculated as follows:

$$pe_t^i = \begin{cases} \sin(w_k \cdot t) & \text{if } i = 2k \\ \cos(w_k \cdot t) & \text{if } i = 2k+1 \end{cases} \quad (3)$$

where k is used for determining whether i is an odd or even number, $i \in \{0, \ldots, d-1\}$ is the encoding index, d is the dimensionality of the input embedding, and $w_k = \frac{1}{10000^{2k/d}}$. The final token representation fed into the self attention layer is a sum of the token embedding and the positional encoding.

Context encoder: The structure of the context encoder is the same as the source encoder. As the *context* inputs (i.e., the structural context, the post-log code context, and logging text in similar code) are only discussed in RQ2 and RQ3, we describe the details about how we integrate the *context* into our model in RQ2.

Target decoder: The decoder in Transformer has a similar structure with the encoder. It also consists of N stacked layers, with three sub-layers in each layer (slightly different to the two sub-layers in the source encoder). The additional second sub-layer takes the source encoder's output and the decoder's states which are generated by the first self attention sub-layer. Besides, an attention masking is applied to the first self attention sub-layer. This masking prevents the future information from being leaked to the decoder before the prediction and ensures that the predictions only rely on the previous outputs.

Given a source code and logging text corpus D, the goal of training the Transformer model is to find parameters θ that maximize the log likelihood of the training data:

$$\widehat{\theta} = \arg\max_{\theta} \sum_{\langle X,Y \rangle \in D} \log P(Y|X;\theta) \quad (4)$$

where P is the conditional probability of the target sequence Y (i.e., the logging text) given the source sequence X (i.e., the source code).

III. Evaluation Setup

A. Subject projects

We evaluate *LoGenText* on 10 open-source Java projects. We choose the same subject projects that are used in prior work [19] which studies the characteristics of logging texts. The details of the studied versions of these projects are listed in Table I. The source lines of code of the studied projects ranges from 330K to 1.7M. These projects have about 2K to 12K logging statements, among which 76.2% to 95.8% have logging texts. Similar to prior work [19], we evaluate *LoGenText* on the logging statements with logging texts.

TABLE I: Details of the studied projects.

Project	Version	SLOC	# of logging statements	# of logging statements with text
ActiveMQ	5.16.0	415k	2,185	2,093 (95.8%)
Ambari	2.7.5	490K	4,150	3,651 (88.0%)
Brooklyn	1.0.0	339K	2,937	2,813 (95.8%)
Camel	3.4.2	1.4M	7,046	6,366 (90.3%)
CloudStack	4.14.0	645K	12,015	10,613 (90.3%)
Hadoop	3.3.0	1.7M	12,471	11,270 (88.3%)
HBase	2.3.0	778K	5,534	5,071 (90.4%)
Hive	3.1.2	1.7M	6,845	6,290 (91.6%)
Ignite	2.8.1	1.1M	3,366	3,048 (90.6%)
Synapse	3.0.1	330K	1,978	1,508 (76.2%)
Avg.		890K	5853	5272 (90.1%)

B. Experimental settings

1) Model training settings: The goal of *LoGenText* is to use the Transformer-based model to automatically generate logging texts with the source code as the input. Our *LoGenText* is implemented based on Fairseq [22], [34], a sequence-to-sequence modeling toolkit. We use the same model structure as in the original Transformer model: six stacked layers (i.e., $N = 6$), 512 embedding dimensions for both the source encoder and the target decoder, and 2,048 feed-forward embedding dimensions. We use the Adam optimizer to optimize the model parameters (same as the original Transformer model). To prevent overfitting, we use a dropout rate of 0.1. More details about the configuration of hyperparameters can be found in our replication package.

For each subject project, we split all the instances into 80%/10%/10% training/validation/testing sub datasets[2]. As the number of instances in each subject project is relatively small (i.e., about 1.5K to 11K), it is challenging to fit a Transformer model with more than forty million parameters. To overcome this problem, we adopt a two-stage training strategy (*a.k.a.,*transfer learning (TL)) [35], [36], [37]: for each subject project, 1) we first pre-train a model using all the training sets from 10 projects for 50 epochs, and 2) we then continue to fine-tune the pre-trained model parameters using the target project's training set for another 50 epochs. The validation set is used to monitor the performance of the model during training to avoid overfiting.

For inference, we use the beam search with a width of eight, which means at each step, the top eight candidate tokens with the highest scores are kept for the next step. However, the beam search algorithm favors shorter sequences [38], [39]. To address this problem, we adopt the length penalty, which gives favors to longer sequences [40]. In our experiments, we set the value of length penalty to 2.5. In addition, we set the maximum length and minimum length of the generated logging text to be 100 and 3, respectively, as we find that the lengths of 92.4% to 98.4% of the logging texts in the studied projects fall in this range.

The training of our models are conducted in a cluster of machines each with a NVIDIA V100 Tensor Core GPU.

2) Model evaluation approaches: We evaluate the performance of *LoGenText* using a combination of quantitative evaluation and human evaluation.

Quantitative evaluation: We use two widely used machine translation evaluation metrics, BLEU [41] and ROUGE [42], to evaluate the quality of the generated logging text sequences in terms of their similarity to the original logging texts inserted by the developers. The details of these evaluation metrics are described in the research questions that apply these metrics.

[2]The sizes of training datasets range from 1K to 9k.

Human evaluation: In order to evaluate how developers perceive the generated logging texts, we also performed a human evaluation, which is detailed in Section V.

C. Baseline approach

We compare our approach with prior work by He et al. [19], which is by far the state-of-the-art approach for generating logging texts. Their method assumes that similar code snippets tend to have similar logging texts. To generate the logging text for a given code snippet, He et al. [19] perform a search in the training corpus to retrieve the most similar code snippet based on Levenshtein distance [43]. The logging text of the most similar code snippet is used as the logging text for the given code snippet. We re-implement their method as a baseline to compare with our approach.

IV. Evaluation Results

In this section, we discuss the results of evaluating *LoGenText* through answering three research questions.

RQ1: How well can the base form of *LoGenText* automatically generate logging text?

Motivation. Prior research [19] has observed that logging texts are predictable and proposes a simple approach (the baseline approach in Section III) based on the intuition that similar code snippets contains similar logging texts. Such a simple approach has demonstrated a promising result. Therefore, in this RQ, we would like to explore whether our NMT-based solution (i.e., *LoGenText*) can automatically generate logging texts with a better performance than the baseline approach.

Approach. We evaluate the base form of *LoGenText*, i.e., using only the source input (pre-log code) to generate the logging texts, and compare it with the baseline approach [19]. Following prior work [19], we evaluate the quality of the generated logging texts using two widely used metrics for machine translation evaluation, i.e., BLEU[3] [41], [44] and ROUGE[4] [42]. Both BLEU and ROUGE take the automatically generated logging texts and the reference logging texts (i.e., the original logging texts written by developers) as input and calculate the similarity between them, which outputs a percentage score between 0 and 1. The higher the score, the better the generated logging texts in terms of their similarity to the reference logging texts.

BLEU (Bilingual Evaluation Understudy) is used to evaluate the match between a generated text and a reference text, which is calculated as follows:

$$\text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{5}$$

$$BP = \begin{cases} 1 & if\ c > r \\ e^{(1-r/c)} & if\ c \leq r \end{cases} \tag{6}$$

where p_n is the modified n-gram precision (i.e., the maximum number of n-grams co-occurring in the automatically generated logging text and the reference logging text divided by the the total number of n-grams in the generated logging text), w_n are positive weights that can be configured, BP is a brevity penalty, c is the length of the generated logging text and r is the length of the reference logging text. In our evaluation, we choose $N = 4$ and uniform weights $w_n = 1/N$, same as prior work [19]. In addition to the overall BLUE score, we also consider the specific BLEU-n (n = 1, 2, 3 ,4) scores, which are the BLUE scores considering only one gram size.

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is a set of metrics for evaluating automated generated texts in text summarization and translations. ROUGE is calculated as follows:

$$\text{ROUGE-n} = \frac{\sum_{gram_n \in Ref} Count_{match}(gram_n)}{\sum_{gram_n \in Ref} Count(gram_n)} \tag{7}$$

where n is the length of the n-gram ($gram_n$), and $Count_{match}(gram_n)$ is the number of n-grams co-occurring in the automatically generated logging text and the reference logging text, Ref. We calculate ROUGE-1, ROUGE-2 and ROUGE-L. ROUGE-L measures the longest matching sequence of tokens using LCS (Longest Common Subsequence).

***Results.* Our base form of *LoGenText* generally outperforms the baseline approach.** Our experimental results of comparing *LoGenText* with the baseline on the 10 studied projects are presented in Table II. The best results are highlighted in the **bold** font. We can see that the base form of *LoGenText* provides a ROUGE-L score of 41.1 to 52.3 and a BLEU score of 21.8 to 39.0 for the studied projects. As shown in Table II, *LoGenText* outperforms the baseline approach for all the projects in terms of ROUGE-L by 5.7% to 22.8% and has a higher BLEU score than the baseline approach by 2.9% to 18.5% in seven out 10 projects. In addition, besides the overall BLEU and ROUGE-L, *LoGenText* performs better than the baseline approach in almost all different gram sizes (i.e., BLEU-n and ROUGE-n). Our results indicate the promising research direction of using neural translation techniques in automated generation of logging text.

On the other hand, we also observe that the base form of *LoGenText* may not always provide a better performance in terms of BLEU scores (e.g, BLEU-4). As shown in Table II, *LoGenText* performs better than the baseline approach for seven out 10 projects in terms of BLEU but worse for the other three projects (Brooklyn, Synapse and Hive). By examining the BLEU scores of different gram size (i.e., BLEU-n), we realized that the base form of *LoGenText* always outperforms the baseline in terms of smaller gram sizes (i.e., BLEU-1 and BLEU-2); in some cases (e.g., for the projects Brooklyn, Synapse, and Hive) , the base form of *LoGenText* may not perform better than the baseline approach in terms of larger gram sizes (i.e., BLEU-3 and BLEU-4). This phenomenon can be explained by the different working mechanisms of these two different approaches. The baseline approach simply reuses logging texts from other code snippets [14], thus it tends to produce long sequence of identical tokens between code snippets, which can result in relatively high larger-gram BLEU scores, especially when there are many duplications of logging texts [45]. In contrast, *LoGenText* automatically generates new logging texts token by token, thus it may not

[3]https://github.com/mjpost/sacrebleu

[4]https://github.com/pltrdy/rouge

TABLE II: Evaluation results of using *LoGenText* and the baseline approach to generate logging texts (RQ1).

Projects	Methods	BLEU(%)	BLEU-1(%)	BLEU-2(%)	BLEU-3(%)	BLEU-4(%)	ROUGE-L(%)	ROUGE-1(%)	ROUGE-2(%)
ActiveMQ	Baseline	21.0	37.0	22.9	18.4	**16.0**	36.1	36.0	21.6
	LoGenText	**23.0(+9.5%)**	**44.6**	**26.0**	**19.6**	**16.0**	**43.4(+20.4%)**	**43.1**	**25.1**
Ambari	Baseline	19.9	36.8	22.0	17.0	**14.1**	36.8	37.5	22.4
	LoGenText	**22.8(+14.6%)**	**44.0**	**25.6**	**17.8**	13.4	**42.9(+16.5%)**	**44.1**	**24.7**
Brooklyn	Baseline	**26.0**	41.4	25.5	**21.8**	**19.7**	38.1	40.9	23.0
	LoGenText	25.4(-2.1%)	**48.7**	**28.4**	20.8	16.8	**43.6(+14.4%)**	**47.1**	**26.2**
Camel	Baseline	37.9	51.5	39.2	35.6	33.8	47.5	47.9	33.0
	LoGenText	**39.0(+2.9%)**	**58.3**	**43.3**	**38.1**	**35.9**	**52.3(+10.2%)**	**52.5**	**35.3**
CloudStack	Baseline	30.1	46.6	33.5	28.4	25.4	43.9	44.5	30.0
	LoGenText	**34.6(+14.7%)**	**52.4**	**37.3**	**30.0**	**25.6**	**50.1(+14.0%)**	**50.8**	**35.2**
Hadoop	Baseline	19.6	37.2	22.8	18.7	**16.8**	34.1	34.9	20.1
	LoGenText	**21.8(+11.1%)**	**44.4**	**25.4**	**19.1**	16.5	**41.1(+20.5%)**	**42.3**	**23.0**
HBase	Baseline	19.5	38.4	24.2	19.4	15.9	38.4	38.9	26.1
	LoGenText	**23.1(+18.5%)**	**46.1**	**28.2**	**21.6**	**17.2**	**46.5(+21.2%)**	**47.0**	**30.6**
Hive	Baseline	**28.2**	42.9	29.8	**26.2**	**24.0**	42.4	42.9	28.9
	LoGenText	28.0(-0.6%)	**47.4**	**30.8**	25.2	21.7	**46.7(+10.2%)**	**47.2**	**29.8**
Ignite	Baseline	21.5	38.5	23.4	18.4	14.8	37.1	38.0	22.9
	LoGenText	**24.9(+15.6%)**	**50.9**	**30.7**	**23.3**	**18.3**	**45.5(+22.8%)**	**47.2**	**27.1**
Synapse	Baseline	**34.1**	46.7	**36.7**	**31.7**	**27.2**	46.9	46.8	**36.9**
	LoGenText	28.9(-15.3%)	**53.3**	34.7	26.7	21.5	**49.5(+5.7%)**	**50.2**	32.0
Avg.	Baseline	25.8	41.7	28.0	23.6	**20.8**	40.1	40.8	26.5
	LoGenText	**27.1(+5.0%)**	**49.0**	**31.1**	**24.2**	20.3	**46.1(+15.0%)**	**47.2**	**28.9**

Note: The numbers in the brackets indicate the relative change of *LoGenText* to the baseline approach.

always produce long sequences of tokens that are identical to the ones written by developers, even though the generated ones may have similar semantic meanings with the written ones, as discussed in our user study in Section V.

The base form of our NMT-based approach *LoGenText* generally outperforms the baseline approach that leverages the existing logging texts in similar code snippets. Our results illustrate the promising future research opportunity of formulating automated logging text generation as neural machine translation tasks.

RQ2: Can incorporating context information improve the base form of *LoGenText* in generating logging texts?

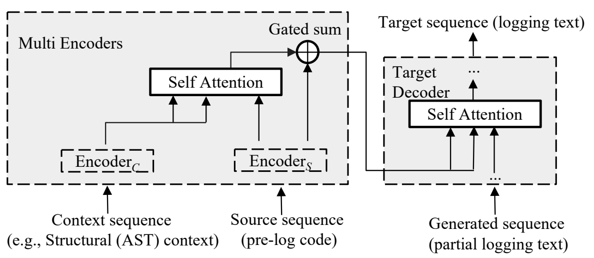
Motivation. Prior studies [46], [31], [47], [48], [49], [50], [51], [52] on NMT show that incorporating the context information (e.g., surrounding text) of the source input may provide promising results in generating better translations. In addition, the context information (e.g., surrounding source code, AST structure of source code) of a particular source code of interest has shown benefits in some software engineering (SE) tasks that rely on neural network-based techniques [53], [54], [55], [56], [27]. Therefore, in this research question, we aim to understand whether the context information (e.g., the post-log code and the structural (AST) information of a logging statement) can help further improve *LoGenText* in automatically generating logging texts.

Approach. We propose a context-aware form of *LoGenText* and consider two types of context information in this research question: the post-log code and the structural (AST) information related to a logging statement. Below we discuss how we extract such information and incorporate it in *LoGenText*.

Extracting context information. *Extracting the structural (AST) context:* We use AST extracted by *srcML* [57] to represent the location of a logging statement. The structural information represented by the AST has been applied successfully in many SE tasks, including suggesting *where to log* [17] and *how to choose log levels* [18]. First, we extract the AST of the method containing the logging statement. Then, we convert the AST into a sequence of AST node types (e.g., if statement) following a depth-first traversal. We only keep the sequence of AST node types prior to the logging statements.

Extracting the post-log code context: Although a logging statement is usually not directly related to the subsequent code (i.e., post-log code), prior research [19] shows the post-log code may provide some extra information relevant to the logging text. Therefore, we consider the post-log code as the context input instead of the source input in our NMT-based model. Specifically, the post-log code contains the code from the location that immediately follows the logging statement to the end of the containing method. We use the same approach as the pre-log code (cf., Section II) to convert the post-log code into a sequence of code tokens.

Integrating context information in our models. There are mainly two approaches for integrating the context information in NMT-based models: (1) simply concatenating the context and the source as a new input sequence [58], [59], and (2) utilizing a multi-encoder model, where additional neural networks are used to encode the context [22], [31], [46]. Prior work [31], [22] shows that the multi-encoder approach is more effective for incorporating context information in NMT tasks. We experimented with both approaches and we also found that the latter approach shows better performance in our context. Therefore, we use the multi-encoder approach in this paper.



Multi Encoders
Gated sum
Self Attention
Encoder_C
Encoder_S
Context sequence (e.g., Structural (AST) context)
Source sequence (pre-log code)
Target sequence (logging text)
Target Decoder
Self Attention
Generated sequence (partial logging text)

Fig. 3: An overview of the multi-encoder Transformer.

TABLE III: Evaluation results of incorporating contexts (AST, post-log code) in *LoGenText* for logging text generation (RQ2).

						BLEU(%)						
		ActiveMQ	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**
	Baseline	21.0	19.8	26.0	37.9	30.1	19.6	19.5	28.2	21.5	34.1	25.8
	Base *LoGenText* (RQ1)	23.0	22.8	25.4	39.0	**34.6**	21.8	23.1	28.0	24.9	28.8	27.1
With context	AST	**24.1**	23.8	27.8	**41.8**	**34.6**	**23.3**	23.5	**29.6**	**28.8**	**37.2**	**29.5**
	Post-log code	**24.1**	**24.5**	**28.4**	39.9	34.3	23.1	**24.3**	**29.6**	28.2	34.8	29.1
						ROUGE-L(%)						
		ActiveMQ	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**
	Baseline	36.1	36.8	38.1	47.4	43.9	34.1	38.4	42.4	37.1	46.9	40.1
	Base *LoGenText* (RQ1)	**43.4**	42.9	43.6	52.3	50.1	41.1	**46.5**	46.7	45.5	49.5	46.2
With context	AST	42.5	43.4	44.0	**53.9**	**50.8**	**42.1**	46.4	**48.2**	**47.6**	**53.6**	**47.3**
	Post-log code	42.8	**43.5**	**44.7**	53.6	50.4	41.5	46.3	48.0	46.0	53.4	47.0

Note: Values in bold font indicate the best performing models.

The structure of our context integration approach is illustrated in Figure 3. The context encoder replicates the original Transformer encoder and takes one type of context information (e.g., AST context, post-log code context) as input. The output of the context encoder together with the output of the source encoder are then fed into a self-attention layer. Then, the outputs of the attention layer and the source encoder are fused by a gated sum. Formally, let S be the output of the source encoder and C be the output of the attention layer, the output of the gated sum G is

$$G = \lambda \odot C + (1 - \lambda) \odot S \tag{8}$$

where the gating weight λ is calculated by

$$\lambda = \sigma\left(W\left[C, S\right] + b\right) \tag{9}$$

where $\sigma\left(\cdot\right)$ is the sigmoid function, W is the weight parameters of the model, and b is the bias.

In order to understand the impact of different types of context information, we evaluate the performance of the models using each type of context. We use the same metrics used in RQ1 (i.e., BLEU and ROUGE-L) to evaluate the quality of the generated logging texts.

Results. **Incorporating context information can improve the performance of the base form of *LoGenText* and outperforms the baseline approach in all the studied projects.** Table III shows the results of incorporating different context information. By comparing the context-aware form of *LoGenText* with the base form, we find that by incorporating the context information using multi-encoders models, we can obtain a performance improvement on almost all the projects. For example, by encoding the structural (AST) context into our *LoGenText*, we obtain an 29.2% relative (8.4% absolute) increase in terms of BLEU score in project Synapse over the base form of *LoGenText*. Overall, as shown in Table III, the context-aware form of *LoGenText* that incorporates the AST context provides a BLEU score of 23.3 to 41.8 and a ROUGE-L score of 42.1 to 53.9 for the studied projects, which are 5.0% to 34.0% and 13.7% to 28.3% higher than the baseline approach, respectively. In addition, unlike the base form of *LoGenText* which may underperform the baseline approach for certain projects (e.g, Brooklyn and Synapse) in terms of BLEU scores, our context-aware form of *LoGenText* can provide better BLEU scores than the baseline approach for all the studied projects. The results demonstrate that *LoGenText* can benefit from the extracted context information.

Meanwhile, we observe that for some projects (e.g., Synapse and Camel), different types of context can result in diverse performance. In particular, for the Synapse project, incorporating AST and post-log code results in BLEU scores of 37.2 and 34.8, respectively. This finding suggests that practitioners should be careful with the selection of contexts for different projects, as they may produce diverse results. On the other hand, we also observe that leveraging the AST context performs better than post-log context in seven out of the 10 projects and has the largest improvement over the base form of *LoGenText* on average. This observation further confirms the success of applying AST information in suggesting logging activities [17], [18].

We also find that incorporating additional context may not always improve the performance of *LoGenText* significantly. As shown in Table III, by adding context using the multi-encoders model, the performance on the project CloudStack (using AST context) remains the same as that without the context. This may be due to the fact that CloudStack has a much higher number of pre-log code tokens for each generated logging text (information used in the base form of *LoGenText*) than other projects, leading to less value of adding the context information.

Incorporating context information (AST and post-log code) can improve the performance of the base form of *LoGenText* for generating logging texts, and different context information may have diverse impact on the studied projects.

RQ3: Can incorporating logging text from similar code improve the base form of *LoGenText* in generating logging texts?

Motivation. Prior work [19] proposes a preliminary logging text generation approach that simply reuses the logging text from the most similar code snippet (i.e., our baseline approach) and achieves promising results. Their results suggest that the logging in similar code may provide additional information about the logging text to be generated. Although we demonstrate better performance of *LoGenText* than the baseline, it may be the case that the information captured by *LoGenText* and that captured by the baseline approach do not overlap. Therefore, including the information provided by the baseline may further improve the results. Therefore, in this research question, we aim to explore the impact of incorporating logging text in similar code on automated logging text generation

TABLE IV: Evaluation results of incorporating logging text from similar code in *LoGenText* for logging text generation (RQ3).

		BLEU(%)										
		ActiveMQ	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**
	Baseline	21.0	19.8	26.0	37.9	30.1	19.6	19.5	28.2	21.5	**34.1**	25.8
	Base *LoGenText* (RQ1)	23.0	22.8	25.4	39.0	**34.6**	21.8	23.1	28.0	24.9	28.8	27.1
With context	Logging text from similar code	**25.8**	**25.3**	**27.5**	**41.6**	34.4	**22.8**	**24.0**	**29.2**	**26.6**	34.0	**29.1**

		ROUGE-L(%)										
		ActiveMQ	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**
	Baseline	36.1	36.8	38.1	47.4	43.9	34.1	38.4	42.4	37.1	46.9	40.1
	Base *LoGenText* (RQ1)	43.4	42.9	43.6	52.3	50.1	41.1	46.5	46.7	45.5	49.5	46.2
With context	Logging text from similar code	**44.8**	**44.1**	**43.9**	**53.9**	**50.7**	**41.8**	**46.6**	**47.5**	**46.4**	**53.1**	**47.3**

Note: Values in bold font indicate the best performing models.

and examine whether we can improve the base form of *LoGenText* by utilizing such logging information.

Approach. Similar to prior work [19], we leverage the logging texts from similar code in the generation of logging texts.

Extracting logging text from similar code. For each logging statement, we extract its pre-log code and search for the most similar code snippet in the training dataset. Specifically, for a given pre-log code snippet, we follow prior work [19] and use the Levenshtein distance [43] to calculate the similarity between it and all the other code snippets in the training dataset. We then extract the logging text in the most similar code snippet.

Incorporating logging text from similar code. We adopt the same multi-encoder approach as in RQ2 to incorporate the retrieved logging text from similar code. In particular, the logging text in the similar code snippet is encoded using a context encoder, and then a gated sum is applied on the outputs of the context encoder and the source encoder, the output of the gated sum is then fed to the target decoder.

Similar to RQ1 and RQ2, we evaluate the performance of *LoGenText* that incorporates the logging text from the similar code using the BLEU and ROUGE-L metrics.

Results. **Incorporating logging text from similar code can improve the performance of the base form of *LoGenText*.** As shown in Table IV, we find that by incorporating the retrieved logging text from similar code using a context encoder, the performance of the base form of *LoGenText* can be increased in nine out of the ten studied projects (e.g., the average BLEU score increases from 27.1 to 29.1). The results indicate that the logging in similar code may contain useful knowledge for the logging text to be generated in the NMT model. However, incorporating the logging text from similar code (with an average BLEU of 29.1) is less effective than the *LoGenText* that incorporates the AST context (with an average BLEU of 29.5, cf. RQ2).

Similar to our results in RQ2, incorporating logging text from similar does not improve the performance on the CloudStack project over the base form of *LoGenText*. Similarly, this result may be due to the fact that CloudStack has a large number of pre-log code tokens for each generated logging text (information used in the base form of *LoGenText*), which may lead to less value of incorporating the additional logging information from similar code.

Incorporating logging text from similar code can provide additional information to the base form of *LoGenText*. However, it cannot further improve the best performing version of *LoGenText* that incorporates the AST context.

V. Human Evaluation

Our approach *LoGenText* is evaluated in the last section based on quantitative metrics (i.e., BLEU and ROUGE scores) that measure the similarity between the original and the generated logging texts. However, the quantitative metrics may not directly reflect how developers perceive the quality of the generated logging texts. Therefore, in this section, we conduct a human evaluation to further evaluate *LoGenText*.

We invited 42 participants in our human evaluation. The participants include a mix of 23 graduate students who major in computer science or software engineering and 19 software developers who are employed in the software industry across the globe. All the participants have at least five years of experience in software development.

Our human evaluation contains two tasks: **task 1)** evaluating the *similarity* between the automatically generated logging texts and the original logging texts extracted from source code. **task 2)** evaluating the logging texts separately from three aspects [60], i.e., *relevance*, *usefulness* and *adequacy* based on the given source code. For task 1, each participant was given 15 logging statements that were randomly sampled from the 10 projects to evaluate. We presented the participants with the original logging texts, the logging texts generated by the baseline, and the logging texts generated by *LoGenText*. Since our results in Section IV show that the context-aware form of *LoGenText* incorporating the AST context has the best overall performance, we used it to generate logging texts used in our human evaluation. We named the logging text from the original logging statement as *log-ref* and the two generated logging texts as *log-1* and *log-2*. We asked the participants to rate the similarity between the generated logging texts (*log-1* and *log-2*) and the original logging texts (*log-ref*). In order to avoid the bias caused by the order of the two generated logging texts, we randomly assigned the one generated by *LoGenText* or by the baseline as *log-1* or *log-2*. Each generated logging text is evaluated based on a scale from 0 to 4 where 0 means no similarity and 4 means perfect similarity. For task 2, each participant was randomly given three logging statements to evaluate. We presented each participant with the original logging text, the logging text generated by the baseline, the logging text generated by *LoGenText*, and the surrounding method of the logging statement that highlights the location of the logging statement. We randomly assigned the three logging texts as *log-a*, *log-b* and *log-c*. We asked the participant to rate the three logging texts based on the given code snippet from three aspects, i.e., *relevance*, *usefulness* and *adequacy*. *Relevance* refers to how relevant the logging text is to the given source code. *Usefulness* refers to how useful the logging text is for collecting valuable runtime information of the source code. *Adequacy* refers to how the logging text is acceptable in quality or quantity with regard to the given source code. Each logging text is evaluated based on a scale from 0 to 4 where 0 means irrelevant/useless/unacceptable and 4 means perfect relevance/usefulness/adequacy.

***LoGenText* generates logging texts that are significantly more similar to the original logging texts than that generated by the baseline approach.** Figure 4 presents the distribution of the user ratings in our evaluation. We find that *LoGenText* generates more logging texts with the ratings of 3 and 4 while fewer logging texts with the ratings of 0 and 1 than the baseline approach. We conducted a Wilcoxon signed-rank

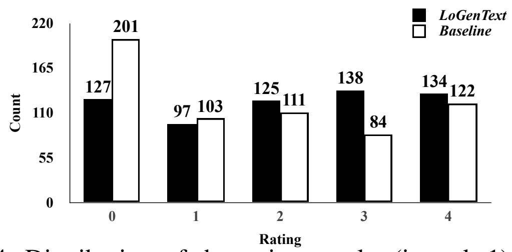


Fig. 4: Distribution of the rating results (in task 1) in terms of the similarity between the generated logging texts and the reference logging texts.

test [61] to compare the ratings of the logging texts generated by *LoGenText* and the baseline approach. With a p-value $\ll 0.00001$, we can confirm that the difference between the ratings of the logging texts generated by the two approaches is statistically significant. On the other hand, despite the significant improvement over the baseline approach, we still observe that more than one third of the automatically generated logging texts by *LoGenText* receive a rating of 0 or 1. The results suggest opportunities for future research that further improves the automated logging generation.

TABLE V: Comparing the human ratings (in task 1) and the BLEU and ROUGE scores of the logging texts generated by *LoGenText*.

Rating	**0**	**1**	**2**	**3**	**4**
BLEU	14.3	20.6	27.4	36.4	78.5
ROUGE-L	21.4	29.7	37.4	46.4	87.3

In order to reflect on the results of our research questions that leverage quantitative metrics BLEU and ROUGE to evaluate *LoGenText* (cf., Section IV), we analyze the relationship between the results of the quantitative measurement and the human evaluation. Specifically, we group the logging texts generated by *LoGenText* by each rate, then evaluate the BLEU and ROUGE score of the logging texts in each group. As shown in Table V, higher BLEU and ROUGE scores are both associated with higher user ratings. Such results confirm the validity of our findings in our research questions that leverage the quantitative metrics.

We manually examine the generated logging texts for which the participants assigned a very high rating (i.e., 3 or 4) while the BLEU and ROUGE values are relatively low (i.e., lower than median), in order to further understand the quality of the generated logging texts. In particular, there are 79 (12.5%) cases where the human ratings are the highest (i.e., 3 or 4) while the BLEU scores are lower than median. We find two main reasons contributing to such inconsistency 1) **Using shorter words.** In the generated logging texts, the generated words are often short and easy to follow. For example, in a logging statement from CloudStack,

```
// Original logging statement:
LOG.info("copying localfile := " + sourceFilepath +
    " to hdfsPath := " + destFilePath)
// Extracted logging text after preprocessing:
"copying localfile <vid> to hdfspath <vid>"
// Generated logging text:
"copying local file <vid> to <vid>"
```

the original logging text uses the term "*localfile*"; while our generated logging text uses the term "*local file*". Although these two terms have very low similarity in terms of BLEU and ROUGE, they have a very similar meaning. 2) **Using synonyms.** Another reason for the inconsistency is the use of synonyms. For example, a logging text from Hadoop says "*no beanstalks defined*" while our generated logging text says "*no beanstalk definitions found*". Both logging texts have similar meanings but with different choices of words, which results in a high human rating but low BLEU and ROUGE-L values.

```
// Original logging statement:
log.debug("No beanstalks defined for
    initialization.")
// Extracted logging text after preprocessing:
"no beanstalks defined for initialization"
// Generated logging text:
"no beanstalk definitions found for initialization"
```

***LoGenText* outperforms the baseline approach in all three aspects.** Table VI shows the mean and median of relevance, usefulness and adequacy scores of the reference logging texts and the logging texts generated by *LoGenText* and the baseline approach. We can see that *LoGenText* outperforms the baseline approach on all three aspects with an average score of 2.67, 2.41 and 2.15, respectively. Similar to task 1, we also conducted a Wilcoxon signed-rank test and the difference is statistically significant for each aspect. However,

TABLE VI: Comparing the mean and median ratings of the logging texts in task 2. The median ratings are in the brackets following the mean ratings.

	Relevance	**Usefulness**	**Adequacy**
Reference	3.37 (4)	3.19 (4)	3.02 (3)
Baseline	2.09 (2)	1.89 (2)	1.75 (2)
LoGenText	**2.67 (3)*****	**2.41 (3)*****	**2.15 (2)****

Note: ***: p-value<0.001; **: 0.001<p-value<0.01.

there is still a non-negligible margin between the logging texts generated by *LoGenText* and the reference logging texts.

The results call for future research that narrows down the gap between the logging texts written by developers and the automatically generated logging texts. On the other hand, the mean scores of the reference logging texts are 3.37, 3.19 and 3.02 respectively, which indicate that some logging texts inserted by the developers can still be further improved and call for high-quality logging texts to record the software execution information.

The logging texts generated by *LoGenText* have a higher quality than that generated by the baseline approach in terms of relevance, usefulness, adequacy, and their similarity to the logging texts written by developers. Our results also suggest future research opportunities for improving automated logging generation.

VI. Threats to Validity

External Validity. In this paper, we evaluate *LoGenText* based on 10 subject systems. All of the subject systems are open-source systems that are mainly written in Java. Evaluating *LoGenText* with a cross-project setting or on other systems that are written in other languages, with closed-source code, or running on mobile devices, may further demonstrate the effectiveness and limitations of our approach.

Internal Validity. In RQ2, we attempt to include two types of context information to further improve *LoGenText*. Similarly, we adopt the same approach to incorporate logging texts from similar code snippets in RQ3. There could exist other context information and other strategies for integrating the context information, while our findings do not in any way claim to generalize the usefulness of other types of context information nor other integration strategies. We evaluate the effectiveness of *LoGenText* based on both quantitative metrics (i.e., BLEU and ROUGE) and human ratings. The quantitative metrics may not reflect the actual quality of the generated logging from developers' perspective, while the human ratings may include subjective bias introduced by the individual participants. Future work should consider further evaluating *LoGenText* by using it in a real-life industrial setting.

Construct Validity. *LoGenText* requires several hyper-parameters for the training process, such as the dimensions, the number of layers, and the number of attention heads, which may impact the results of generating logging texts. To minimize the bias caused by the hyper-parameter configurations, we follow the practices from prior studies [21], [22] to configure the hyper-parameters. Performing further fine tuning on these hyper-parameters may even further improve the results from *LoGenText*. In our evaluation, the data from each project is randomly split into 80%/10%/10% training, validation and testing datasets, which may introduce the selection bias.

VII. Related Work

Automated logging suggestions. To address the challenge of logging, prior research has proposed automated approaches that provide different logging suggestions including the locations of logging statements [23], [17], [25], [62], [63], [64], the verbosity levels [18], [65], the variables to include in a logging statement [66], and the need to update an existing logging statement [20]. The most related work to our paper is from He et al. [19], who conduct an empirical study on the usage of natural language descriptions in logging statements and propose an automated logging text generation approach that leverages logging texts from similar code snippets. Their approach has been adopted in this paper as the baseline approach (cf., Section III). Other research aims to detect issues in logging statements. Chen et al. [67] and Hassani et al. [68] discovered anti-patterns of logging statements from prior log-related code changes and issue reports. Automated tools are designed and implemented to detect these anti-patterns in logging statements. Li et al. [45] discuss the issue of duplicate logging statements.

Despite the above research efforts, providing automated suggestions of logging texts is still challenging. Prior work has highlighted the great importance of the information in the logging texts [2], [69]. Therefore, our work aims to provide automated generation of logging texts to support developers' logging decisions.

Empirical studies on software logging. Empirical studies have been conducted on the practices of logging. The first empirical study on quantitatively characterizing the logging practices was performed by Yuan et al. [13]. Afterwards, follow-up studies by Chen et al. [14] and Zeng et al. [15] extend Yuan et al's study from C/C++ projects to Java projects and Android app projects, respectively. Similarly, Shang et al. [16] conduct a study focusing on the evolution of logging statements. Recently, Li et al. [2] conduct a qualitative study on the benefits and costs of logging based on surveying developers and studying logging-related issue reports. Besides those characteristic studies on logging, empirical studies are also carried out focusing on different aspects of logging practices. The studied topics include the stability of logging statements [70], logging utilities [71] and libraries[72], logging configurations [73], and the relationship between logging practices and software quality [74] and performance [75], [15].

All prior studies provide empirical evidences that show the challenges in software logging practices, which motivates our work towards automated generation of logging texts.

VIII. Conclusion

In this paper, we present our approach, *LoGenText*, which automatically generates the textual descriptions of logging statements based on neural machine translation models. By comparing the generated logging texts with the actual logging texts in the source code, we find that *LoGenText* shows promising results in the automated generation of logging texts. Our approach *LoGenText* outperforms the state-of-the-art baseline approach in terms of both quantitative metrics (BLEU and ROUGE) and human ratings. Our research sheds light on promising research opportunities that exploit and customize neural machine translation models for the automated generation of logging statements, which will reduce developer's efforts in logging development and maintenance and potentially improve the overall quality of software logging.

References

[1] T. Barik, R. DeLine, S. Drucker, and D. Fisher, "The bones of the system: A case study of logging and telemetry at microsoft," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion*, ser. ICSE Companion '16, 2016, pp. 92–101.

[2] 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*, pp. 1–1, 2020.

[3] Q. Fu, J.-G. Lou, Q. Lin, R. Ding, D. Zhang, and T. Xie, "Contextual analysis of program logs for understanding system behaviors," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13, 2013, pp. 397–400.

[4] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin, "Assisting developers of big data analytics applications when deploying on hadoop clouds," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 402–411.

[5] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 117–132. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629587

[6] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online system problem detection by mining patterns of console logs," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ser. ICDM '09, 2009, pp. 588–597.

[7] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Proceedings of the 9th IEEE International Conference on Data Mining*, ser. ICDM '09, 2009, pp. 149–158.

[8] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs," in *Proceedings of the 19th International Symposium on Software Reliability Engineering*, ser. ISSRE '08, 2008, pp. 117–126.

[9] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *Proceedings of the 2008 IEEE International Conference on Software Maintenance*, ser. ICSM '08, 2008, pp. 307–316.

[10] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, "Leveraging performance counters and execution logs to diagnose memory-related performance issues," in *Proceedings of the 29th IEEE International Conference on Software Maintenance*, ser. ICSM '13, 2013, pp. 110–119.

[11] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12, 2012, pp. 26–26.

[12] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: Error diagnosis by connecting clues from run-time logs," in *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '10, 2010, pp. 143–154.

[13] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE, 2012, pp. 102–112.

[14] B. Chen and Z. M. J. Jiang, "Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation," *Empirical Software Engineering*, vol. 22, no. 1, pp. 330–374, 2017.

[15] Y. Zeng, J. Chen, W. Shang, and T. P. Chen, "Studying the characteristics of logging practices in mobile apps: a case study on f-droid," *Empir. Softw. Eng.*, vol. 24, no. 6, pp. 3394–3434, 2019. [Online]. Available: https://doi.org/10.1007/s10664-019-09687-9

[16] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora, "An exploratory study of the evolution of communicated information about the execution of large software systems," *Journal of Software: Evolution and Process*, vol. 26, no. 1, pp. 3–26.

[17] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 415–425.

[18] H. Li, W. Shang, and A. E. Hassan, "Which log level should developers choose for a new logging statement?" *Empirical Software Engineering*, vol. 22, no. 4, pp. 1684–1716, 2017.

[19] P. He, Z. Chen, S. He, and M. R. Lyu, "Characterizing the natural language descriptions in software logging statements," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. ACM, 2018, pp. 178–189.

[20] H. Li, W. Shang, Y. Zou, and A. E. Hassan, "Towards just-in-time suggestions for log changes," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1831–1865, 2017.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, 2017, pp. 5998–6008. [Online]. Available: http://papers.nips.cc/paper/7181-attention-is-all-you-need

[22] B. Li, H. Liu, Z. Wang, Y. Jiang, T. Xiao, J. Zhu, T. Liu, and C. Li, "Does multi-encoder help? A case study on context-aware neural machine translation," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Association for Computational Linguistics, 2020, pp. 3512–3518. [Online]. Available: https://www.aclweb.org/anthology/2020.acl-main.322/

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

[24] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the naturalness of software," in *ICSE*. IEEE Computer Society, 2012, pp. 837–847.

[25] H. Li, T.-H. P. Chen, W. Shang, and A. E. Hassan, "Studying software logging using topic models," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2655–2694, Oct. 2018.

[26] V. J. Hellendoorn and P. T. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, 2017, pp. 763–773.

[27] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*. ACM, 2018, pp. 200–210.

[28] T. Luong, I. Sutskever, Q. Le, O. Vinyals, and W. Zaremba, "Addressing the rare word problem in neural machine translation," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Beijing, China: Association for Computational Linguistics, Jul. 2015, pp. 11–19. [Online]. Available: https://www.aclweb.org/anthology/P15-1002

[29] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016.

[30] P. Gage, "A new algorithm for data compression," *C Users Journal*, vol. 12, no. 2, pp. 23–38, 1994.

[31] E. Voita, P. Serdyukov, R. Sennrich, and I. Titov, "Context-aware neural machine translation learns anaphora resolution," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*. Association for Computational Linguistics, 2018, pp. 1264–1274. [Online]. Available: https://www.aclweb.org/anthology/P18-1117/

[32] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *arXiv preprint arXiv:2005.00653*, 2020.

[33] F. Liu, G. Li, B. Wei, X. Xia, M. Li, Z. Fu, and Z. Jin, "Characterizing logging practices in open-source software," in *Proceedings of the 28th International Conference on Program Comprehension*, ser. ICPC '20, 2020.

[34] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, "fairseq: A fast, extensible toolkit for sequence modeling," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Demonstrations*. Association for Computational Linguistics, 2019, pp. 48–53.

[35] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, 2014, pp. 3320–3328. [Online]. Available: http://papers.nips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks

[36] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[37] E. S. Olivas, J. D. M. Guerrero, M. M. Sober, J. R. M. Benedito, and A. J. S. Lopez, *Handbook Of Research On Machine Learning Applications and Trends: Algorithms, Methods and Techniques - 2 Volumes*. Hershey, PA: Information Science Reference - Imprint of: IGI Publishing, 2009.

[38] N. Boulanger-Lewandowski, Y. Bengio, and P. Vincent, "Audio chord recognition with recurrent neural networks," in *Proceedings of the 14th International Society for Music Information Retrieval Conference, ISMIR 2013, Curitiba, Brazil, November 4-8, 2013*, 2013, pp. 335–340.

[Online]. Available: http://www.ppgia.pucpr.br/ismir2013/wp-content/uploads/2013/09/243_Paper.pdf

[39] M. Neishi, J. Sakuma, S. Tohda, S. Ishiwatari, N. Yoshinaga, and M. Toyoda, "A bag of useful tricks for practical neural machine translation: Embedding layer initialization and large batch size," in *Proceedings of the 4th Workshop on Asian Translation, WAT@IJCNLP 2017, Taipei, Taiwan, November 27- December 1, 2017*. Asian Federation of Natural Language Processing, 2017, pp. 99–109. [Online]. Available: https://www.aclweb.org/anthology/W17-5708/

[40] S. Wu, D. Zhang, N. Yang, M. Li, and M. Zhou, "Sequence-to-dependency neural machine translation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, R. Barzilay and M. Kan, Eds. Association for Computational Linguistics, 2017, pp. 698–707.

[41] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 2002, pp. 311–318. [Online]. Available: https://www.aclweb.org/anthology/P02-1040/

[42] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[43] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[44] M. Post, "A call for clarity in reporting BLEU scores," in *Proceedings of the Third Conference on Machine Translation: Research Papers, WMT 2018, Belgium, Brussels, October 31 - November 1, 2018*. Association for Computational Linguistics, 2018, pp. 186–191.

[45] Z. Li, T. P. Chen, J. Yang, and W. Shang, "Dlfinder: characterizing and detecting duplicate logging code smells," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 152–163. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00032

[46] J. Zhang, H. Luan, M. Sun, F. Zhai, J. Xu, M. Zhang, and Y. Liu, "Improving the transformer translation model with document-level context," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*. Association for Computational Linguistics, 2018, pp. 533–542.

[47] Y. Kim, D. T. Tran, and H. Ney, "When and why is document-level context useful in neural machine translation?" in *Proceedings of the Fourth Workshop on Discourse in Machine Translation, DiscoMT@EMNLP 2019, Hong Kong, China, November 3, 2019*. Association for Computational Linguistics, 2019, pp. 24–34.

[48] E. Voita, R. Sennrich, and I. Titov, "When a good translation is wrong in context: Context-aware machine translation improves on deixis, ellipsis, and lexical cohesion," in *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, A. Korhonen, D. R. Traum, and L. Màrquez, Eds. Association for Computational Linguistics, 2019, pp. 1198–1212.

[49] R. Bawden, R. Sennrich, A. Birch, and B. Haddow, "Evaluating discourse phenomena in neural machine translation," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*. Association for Computational Linguistics, 2018, pp. 1304–1313.

[50] L. M. Werlen, D. Ram, N. Pappas, and J. Henderson, "Document-level neural machine translation with hierarchical attention networks," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*. Association for Computational Linguistics, 2018, pp. 2947–2954.

[51] S. Maruf and G. Haffari, "Document context neural machine translation with memory networks," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*. Association for Computational Linguistics, 2018, pp. 1275–1284.

[52] S. Maruf, A. F. T. Martins, and G. Haffari, "Selective attention for context-aware neural machine translation," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 2019, pp. 3092–3102.

[53] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. ACM, 2018, pp. 542–553.

[54] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 783–794.

[55] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, 2019.

[56] L. Büch and A. Andrzejak, "Learning-based recursive aggregation of abstract syntax trees for code clone detection," in *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*. IEEE, 2019, pp. 95–104.

[57] M. L. Collard, M. J. Decker, and J. I. Maletic, "Lightweight transformation and fact extraction with the srcml toolkit," in *11th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM 2011, Williamsburg, VA, USA, September 25-26, 2011*. IEEE Computer Society, 2011, pp. 173–184.

[58] R. Agrawal, M. Turchi, and M. Negri, *Contextual Handling in Neural Machine Translation: Look Behind, Ahead and on Both Sides*. European Association for Machine Translation, EAMT. [Online]. Available: http://rua.ua.es/dspace/handle/10045/76016

[59] J. Tiedemann and Y. Scherrer, "Neural machine translation with extended context," in *Proceedings of the Third Workshop on Discourse in Machine Translation, DiscoMT@EMNLP 2017, Copenhagen, Denmark, September 8, 2017*. Association for Computational Linguistics, 2017, pp. 82–92.

[60] B. Xu, Z. Xing, X. Xia, and D. Lo, "Answerbot: Automated generation of answer summary to developersundefined technical questions," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. IEEE Press, 2017, p. 706–716.

[61] F. Wilcoxon, *Individual Comparisons by Ranking Methods*. New York, NY: Springer New York, 1992, pp. 196–202. [Online]. Available: https://doi.org/10.1007/978-1-4612-4380-9_16

[62] 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*. ACM, 2017, pp. 565–581.

[63] K. Yao, G. B. de Pádua, W. Shang, C. Sporea, A. Toma, and S. Sajedi, "Log4perf: suggesting and updating logging locations for web-based systems' performance monitoring," *Empir. Softw. Eng.*, vol. 25, no. 1, pp. 488–531, 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09748-z

[64] Z. Li, T.-H. Chen, and W. Shang, "Where shall we log? studying and suggesting logging locations in code blocks," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 361–372.

[65] Z. Li, H. Li, T. Chen, and W. Shang, "DeepLV: Suggesting log levels using ordinal based neural networks," in *Proceedings of the 43rd International Conference on Software Engineering, ICSE 2021*, 2021, pp. 1–12.

[66] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Which variables should i log?" *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[67] B. Chen and Z. M. J. Jiang, "Characterizing and detecting anti-patterns in the logging code," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, p. 71–81. [Online]. Available: https://doi.org/10.1109/ICSE.2017.15

[68] M. Hassani, W. Shang, E. Shihab, and N. Tsantalis, "Studying and detecting log-related issues," *Empirical Software Engineering*, vol. 23, no. 6, pp. 3248–3280, 2018.

[69] D. Yuan, S. Park, P. Huang, Y. Liu, M. M.-J. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: Enhancing failure diagnosis with proactive logging." in *OSDI*, vol. 12, 2012, pp. 293–306.

[70] S. Kabinna, C.-P. Bezemer, W. Shang, M. D. Syer, and A. E. Hassan, "Examining the stability of logging statements," *Empirical Software Engineering*, vol. 23, no. 1, pp. 290–333, Feb. 2018.

[71] B. Chen and Z. M. J. Jiang, "Studying the use of java logging utilities in the wild," in *Proceedings of the 42th International Conference on Software Engineering*, ser. ICSE '20, 2020.

[72] S. Kabinna, C. Bezemer, W. Shang, and A. E. Hassan, "Logging library migrations: a case study for the apache software foundation projects," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, M. Kim, R. Robbes, and C. Bird, Eds. ACM, 2016, pp. 154–164. [Online]. Available: https://doi.org/10.1145/2901739.2901769

[73] C. Zhi, J. Yin, S. Deng, M. Ye, M. Fu, and T. Xie, "An exploratory study of logging configuration practice in java," *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 459–469, 2019.

[74] W. Shang, M. Nagappan, and A. E. Hassan, "Studying the relationship between logging characteristics and the code quality of platform software," *Empirical Software Engineering*, vol. 20, no. 1, pp. 1–27, Feb. 2015. [Online]. Available: https://doi.org/10.1007/s10664-013-9274-8

[75] S. Chowdhury, S. D. Nardo, A. Hindle, and Z. M. Jiang, "An exploratory study on assessing the energy impact of logging on android applications," *Empirical Software Engineering*, vol. 23, pp. 1422–1456, 2017.