LoGenText-Plus: Improving Neural Machine Translation-based Logging Texts Generation with Syntactic Templates

ZISHUO DING, University of Waterloo, Canada
YIMING TANG, Rochester Institute of Technology, USA
XIAOYU CHENG, University of Waterloo, Canada
HENG LI, Polytechnique Montréal, Canada
WEIYI SHANG, University of Waterloo, Canada

Developers insert logging statements in the source code to collect important runtime information about software systems. 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. Prior studies find that developers spend significant efforts modifying their logging texts. However, despite extensive research on automated logging suggestions, research on suggesting logging texts rarely exists. To fill this knowledge gap, we first propose *LoGenText*, reported in our conference paper (Ding et al., 2022), an automated approach that uses neural machine translation models to generate 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. The *LoGenText*'s evaluation on 10 open-source projects indicates that the approach is promising for automatic logging text generation and significantly outperforms the state-of-the-art approach. Furthermore, we extend *LoGenText* to *LoGenText-Plus* by incorporating the syntactic templates of the logging texts. Different from *LoGenText*, *LoGenText-Plus* decomposes the logging text generation process into two stages. *LoGenText-Plus* first adopts a neural machine translation model to generate the syntactic template of the target logging text. Then *LoGenText-Plus* feeds the source code and the generated template as the input to another neural machine translation model for logging text generation. We also evaluate *LoGenText-Plus* on the same 10 projects and observe that it outperforms *LoGenText* on nine of them. According to a human evaluation from developers' perspectives, the logging texts generated by *LoGenText-Plus* have a higher quality than those generated by *LoGenText* and the prior baseline approach. By manually examining the generated logging texts, we then identify five aspects that can serve as guidance for writing or generating good logging texts. Our work is an important step towards the automated generation of logging statements, which can potentially save developers' efforts and improve the quality of software logging. Our findings shed light on research opportunities that leverage advances in neural machine translation techniques for automated generation and suggestion of logging statements.

CCS Concepts: • **Software and its engineering → Software development techniques**.

Additional Key Words and Phrases: software logging, logging text, neural machine translation

Authors' addresses: Zishuo Ding, zishuo.ding@uwaterloo.ca, University of Waterloo, Waterloo, ON, Canada; Yiming Tang, yxtvse@rit.edu, Rochester Institute of Technology, Rochester, NY, USA; Xiaoyu Cheng, x24cheng@uwaterloo.ca, University of Waterloo, Waterloo, ON, Canada; Heng Li, heng.li@polymtl.ca, Polytechnique Montréal, Montreal, QC, Canada; Weiyi Shang, wshang@uwaterloo.ca, University of Waterloo, Waterloo, ON, Canada.

1049-331X/2023/9-ART $15.00
https://doi.org/10.1145/3624740

1 INTRODUCTION

Developers insert logging statements in the source code to collect valuable runtime information about software systems. Logging statements produce execution logs at runtime, which play important roles in the daily tasks of developers and other software practitioners [4, 40]. 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 at runtime, 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 [20, 71], anomaly detection [21, 33, 48, 55, 88, 89], and failure diagnosis [61, 73]. In particular, logs are usually the only available resource for diagnosing field failures [94].

Extensive prior research has shown that writing proper logging statements is an important and challenging task [9, 70, 96, 97]. Besides the typical challenges of deciding where to log [15, 16, 44, 104] and how to choose verbosity levels [41, 74], deciding the textual information in the logging statement is even more challenging [29]. Prior studies find that developers spend significant efforts modifying the textual information in their logging statements [9, 42, 70, 96, 97]. 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 [40].

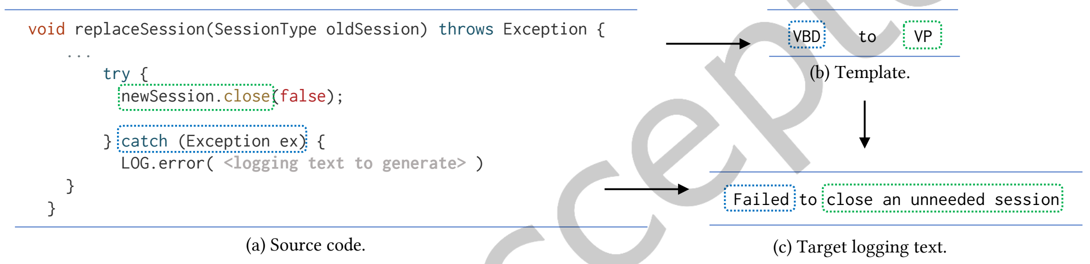
Despite the importance of logging texts, there exists a rare research effort that devotes to assisting developers in writing logging texts. A recent study by He et al. [29] proposes an approach that reuses the texts in the logging statements from similar code snippets. However, since only existing logging texts are directly reused, the texts generated by the prior approach may still require significant revisions by practitioners. Nevertheless, prior work [29] 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* [14], 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 [77]. 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 [38]. 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 [29] 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.

Although *LoGenText* has achieved superior performance over the baseline approach, it still has limitations. For example, in our previous work [14], we find that there exists a non-negligible gap between the automatically generated logging texts and that written by developers, as *LoGenText* may generate logging texts that have similar meanings but different syntactic structures from the developer-written logging texts. Meanwhile, recent studies [18, 27, 81, 85, 90] on text generation tasks (e.g., text summarization, sentence generation) show that incorporating the templates of the target sentences can produce promising results in generating better translations, as templates can provide a positive impact for guiding the translation process [90].

Therefore, we further extend our previous work, *LoGenText*, which was published in SANER 2022 [14], to *LoGenText-Plus* to incorporate the template information (i.e., syntactic structures) of the logging texts, which may guide the generation of the logging text. In this work, we use constituency-based parse trees of logging texts from the training set to train an NMT-based model to generate templates. Then, the generated templates are used to guide the generation of the logging texts. Figure 1 illustrates the process of generating logging texts based on the source code and templates. In this example, the source code (i.e., Figure 1(a)) is used to generate the coarse syntactic template (i.e., Figure 1(b)) which contains different levels of information, including the syntactic symbols (i.e., "VBD", verb with past tense and "VP", verb phrase) and tokens (i.e., "to") of the target logging text. Then, the template together with the source code is fed into a Transformer-based model for generating the target logging text. We assume that by using the templates, we are breaking down the task of logging text generation into several stages in a coarse-to-fine manner and the coarse syntactic templates can guide the generation of the target logging texts.



```
void replaceSession(SessionType oldSession) throws Exception {
    ...
        try {
          newSession.close(false);

        } catch (Exception ex) {
          LOG.error( <logging text to generate> )
      }
    }
```

Fig. 1. An example of the process of generating logging texts based on the source code and templates. "VBD" represents a verb with the past tense, and "VP" represents a verb phrase.

To assess the performance of our newly proposed *LoGenText-Plus*, we evaluate it on the same 10 projects that were previously used to evaluate *LoGenText*. We first compare the logging texts generated by *LoGenText-Plus* with that generated by *LoGenText* as well as the baseline approach [29] using quantitative metrics. Experiments show that *LoGenText-Plus* outperforms the baseline approach as well as the best performing version of *LoGenText* in nine out of the 10 projects. Besides, we conduct another human evaluation to qualitatively evaluate the quality of the generated logging texts. The results further confirm that *LoGenText-Plus* can provide higher-quality logging texts.

The contributions of this paper include (the new contributions compared to that of previous work [14] are highlighted with *.) :

- Our automated approach *LoGenText* significantly outperforms the baseline approach in generating logging texts.
- The newly extended approach, *LoGenText-Plus*, which incorporates the syntactic templates of logging texts further advances the state-of-the-art.*
- Our work suggests that automated approaches for logging text generation should not only focus on the preceding code of a logging statement (as done in prior work) 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.

- Based on the manual evaluation results, we identify five aspects that can be used to generate better logging texts.*

Our work is an important step towards the 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 source code used for training our models[1].

Paper Organization. Section 2 presents the details of our approach: *LoGenText* and its extension *LoGenText-Plus*. Section 3 presents the setup of the experiment for evaluating *LoGenText* and *LoGenText-Plus*. Section 4 and Section 5 present the results of evaluating *LoGenText* and *LoGenText-Plus* through quantitative metrics and human evaluation. Section 6 discusses threats to the validity. Section 7 presents the related work. Finally, Section 8 concludes the paper.

2 APPROACH

In this section, we describe the details of *LoGenText* and its extension *LoGenText-Plus* that leverage neural machine translation (NMT) to automatically generate logging texts.

2.1 Approach Overview

2.1.1 LoGenText. *LoGenText* is an NMT-based approach that uses deep neural networks to translate source code into logging texts. The bottom half of Figure 2 (i.e., the part delimited by the black dashed lines) illustrates the overall approach of *LoGenText* which consists of three phases. 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 [77] 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**).

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 4-RQ1). In RQ2 (Section 4-RQ2) and RQ3 (Section 4-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.

2.1.2 LoGenText-Plus. Besides, we extend *LoGenText* to *LoGenText-Plus* by incorporating the template of the target logging text. Different from *LoGenText*, *LoGenText-Plus* contains two stages:

Stage 1: template generation, where *LoGenText-Plus* adopts a Transformer-based model to predict the templates. As shown in Figure 2, during **data preparation**, for each logging statement in the source code, *LoGenText-Plus* first extracts four types of information (i.e., the target logging text, the pre-log code, the structural (AST) context information and the logging text in the most similar code (cf. Section 4-RQ3)) [14] which are also used in *LoGenText*. Then *LoGenText-Plus* extracts the syntactic template from the logging text as the target sequence for training and from the logging text in similar code which is concatenated with pre-log code as the source sequence. During **model training**, *LoGenText-Plus* feeds the template from the target logging text (i.e., target sequence), the pre-log code together with the template from the logging text in the similar code (i.e.,

[1]Replication package: https://tinyurl.com/4njsbu9m

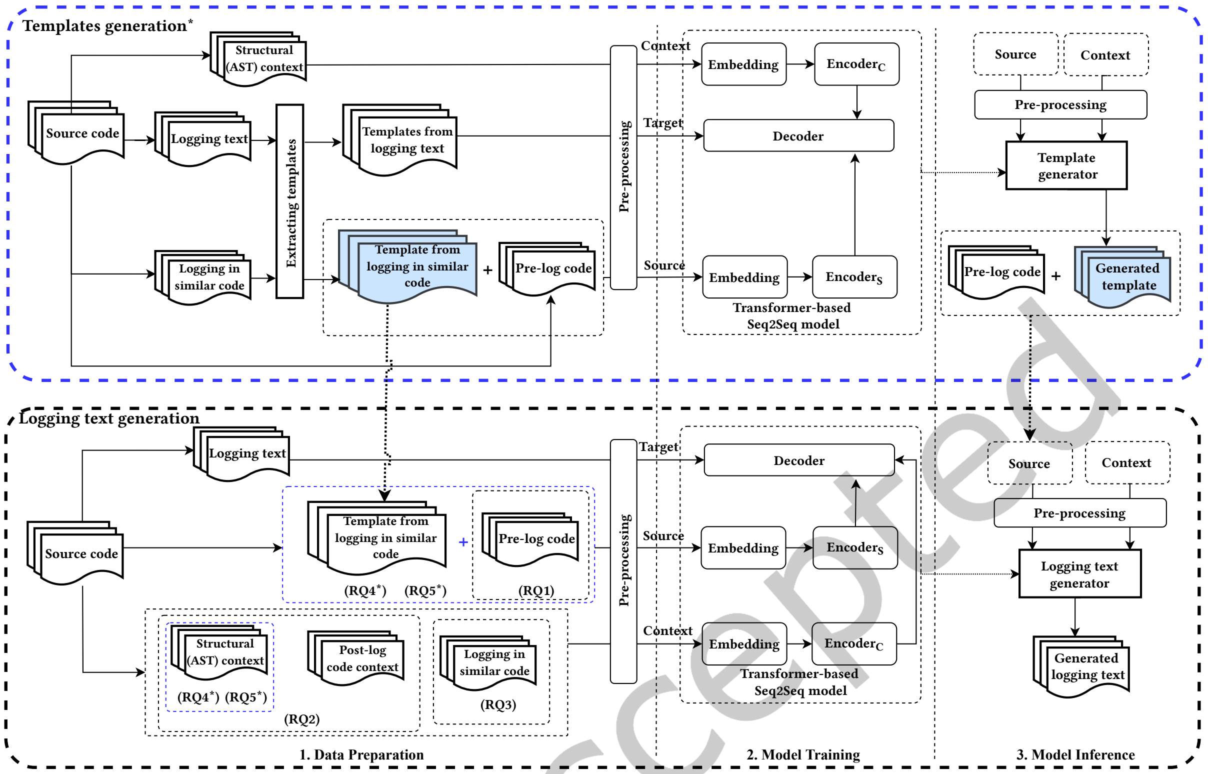


Fig. 2. An overview of *LoGenText* and its extension (*LoGenText-Plus*) which is delimited by the blue dashed lines and highlighted with *. In *LoGenText-Plus*, to train the template generation model, the templates extracted logging texts are used as the target sequence, the templates extracted from logging texts in similar code (highlighted in blue) are concatenated with the pre-log code as the source, and the structural (AST) context is used as the context; To train the logging generation model, the logging texts are used as the target sequence, the templates extracted from logging texts in similar code (highlighted in blue) are concatenated with the pre-log code as the source, and the structural (AST) context is used as the context. During inference, the generated templates are concatenated with the pre-log code as the source, and the structural (AST) context is used as the context.

the source sequence) and the structural (AST) context into a Transformer-based model. Finally, during **model inference**, the trained model (i.e., Template generator) takes the source sequence (i.e., the pre-log code together with the template from the logging text in the similar code) and the structural (AST) context information as input and translates it into the corresponding template (i.e., Generated template).

Stage 2: template-based logging text generation, where *LoGenText-Plus* adopts another Transformer-based model to predict the final logging text based on the generated template in stage 1. As shown in Figure 2, during **data preparation**, similar to that of template generation stage, for each logging statement in the source code, *LoGenText-Plus* first extracts four types of information (i.e., the target logging text, the pre-log code, the structural (AST) context information and the logging text in the most similar code). Then *LoGenText-Plus* extracts the syntactic template from the logging text in similar code which is concatenated with pre-log code as the source sequence. During **model training**, *LoGenText-Plus* feeds the target logging text (i.e., target sequence), the pre-log code together with the template from the logging text in the similar code (i.e., the source sequence) and the structural (AST) context into the Transformer-based model. Finally, during **model inference**, the trained model (i.e., Logging text generator) takes the source sequence (i.e., the pre-log code together with the generated template in the template generation stage) and the structural (AST) context information as input and translates it into the corresponding logging text (i.e., Generated logging text).

In RQ4 (Section 4-RQ4) and RQ5 (Section 4-RQ4), we detail *LoGenText-Plus* and discuss the impact of incorporating different templates on its performance.

2.2 Data Preparation

In this part, we describe the steps for preparing data that are required by both *LoGenText* and *LoGenText-Plus*.

Data preparation involves three types of information: 1) *logging text*, which refers to the static plain text in the logging statement, 2) *(part) 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.

The steps for preparing data that are required by both *LoGenText* and *LoGenText-Plus* are as follows. Details for step **template extraction** specific to *LoGenText-Plus* can be found in Section 4-RQ4, in which we evaluate *LoGenText-Plus*.

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())
```

$\longrightarrow$

```
// Extracted logging text:
   "Removed child queue: <vid>"
```

Extracting (part of) the source data. We use the pre-log code as the main input (i.e., the *source* data) for *LoGenText*. 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 [22, 40].

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.

Pre-processing the logging text and source data. Following the previous approaches for pre-processing the input text data [29, 31, 39], 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 [30, 32]. 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 [54]. 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 (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 [32]. 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 [32]. To address this problem, we employ byte pair encoding (BPE), a data compression technique, to segment the code tokens into subword units [23, 69]. 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.

Note that sometimes, preserving the original case of the source code provides more information, which can be useful for certain code-related tasks. For example, in the task of logging variables recommendation (i.e., which variable in the source code should be logged), the capitalization information can be a strong signal for identifying the variables. However, the authors of BERT also note that typically, the uncased model is better[2] for a range of downstream tasks. In our experiments, our goal is to generate the textual description in the logging statements. Although using BPE tokenization allows us to have relatively good coverage with small vocabularies, unknown tokens still exist. Therefore, we lowercase the source code and logging text to further reduce the out-of-vocabulary (OOV) tokens, especially considering the fact that the capitalization might be inconsistent between the source code and logging text. For example, in the project ActiveMQ[3], the source code contains a variable named "sendShutdown", but in the logging statement the token "shutdown" is used.

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.

2.3 NMT-based log generation

In this work, 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 [79]. Hence, the objective of the models is to approximate the conditional distribution $\log P\,(Y|X;\theta)$ 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. [77], which has shown outstanding performance in many software engineering tasks (e.g., source code summarization [2] and code completion [50]). Figure 3 illustrates the structure of the Transformer translation model that is implemented in *LoGenText* and *LoGenText-Plus*. Note that the two models used in the different stages of *LoGenText-Plus* are both based on the Transformer model. 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.

Source encoder: As Figure 3 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:

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

where Q, K, V are the query, key, and value vectors, $\sqrt{d_k}$ is a normalization factor and 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.

[2]https://github.com/google-research/bert#pre-trained-models

[3]https://github.com/apache/activemq/blob/905f00c843b96996b25017e1b8646de15d703398/activemq-broker/src/main/java/org/apache/activemq/network/DemandForwardingBridgeSupport.java#L324

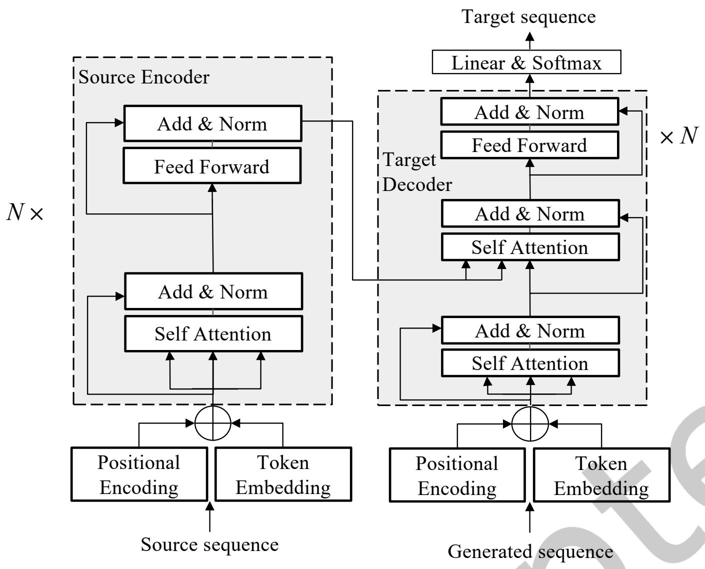


Fig. 3. An overview of the Transformer translation model.

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

$$LayerNorm\left(Attention + X\right) \tag{2}$$

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. Specifically, a positional vector is added to the input embedding, where the positional vector pe for tth token is calculated as follows:

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

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 that is 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 to the encoder. It also consists of N stacked layers, with three sub-layers in each layer (slightly different from 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 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:

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

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).

Note that in *LoGenText-Plus*, we also consider the template generation as a machine translation task, i.e., translating a code snippet into a template that provide the syntactical information of the logging text. The model shares the same structure as in Section 2.3, where the target sequence is changed to the template. As the template information is only used in RQ4 and RQ5, we describe the details about how to generate templates in RQ4 and RQ5.

3 EVALUATION SETUP

3.1 Subject projects

We evaluate our proposed *LoGenText* and *LoGenText-Plus* on 10 open-source Java projects. We choose the same subject projects that are used in prior work [29] which studies the characteristics of logging texts. The details of the studied versions of these projects are listed in Table 1. The source lines of code of the studied projects range 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 [29], we evaluate *LoGenText* and *LoGenText-Plus* on the logging statements with logging texts.

Table 1. 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%)

3.2 Experimental settings

3.2.1 Model training settings. The goal of *LoGenText* and *LoGenText-Plus* is to use the Transformer-based model to automatically generate logging texts with the source code as the input. Our *LoGenText* and *LoGenText-Plus* are implemented based on Fairseq [38, 64], 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 [19, 25, 58–60]. More details about the configuration of hyperparameters can be found in our replication package[4].

For each subject project, we split all the instances into 80%/10%/10% training/validation/testing sub datasets[5]. 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)) [24, 63, 92]: 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 idea is inspired by the work of He et al. [29], where the authors have shown that the logging practices are quite different across different projects, and the n-gram patterns in different projects vary a lot. Meanwhile, a large project usually has a long development history (e.g., years). By fine-tuning a model for a specific project using its existing data, we can leverage the model to suggest logging text for its future development activities. Therefore, we intentionally train separate models for each project, aiming to accurately capture the in-project language patterns while avoiding the (negative) impact of other projects. The validation set is used to monitor the performance of the model during training to avoid overfitting.

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 [6, 62]. To address this problem, we adopt the length penalty, which gives favor to longer sequences [86]. In our experiments, we set the value of the 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 is conducted in a cluster of machines each with an NVIDIA V100 Tensor Core GPU.

3.2.2 Model evaluation approaches. We evaluate the performance of *LoGenText* and *LoGenText-Plus* using a combination of quantitative evaluation and human evaluation.

Quantitative evaluation: We use two widely used machine translation evaluation metrics, BLEU [65] and ROUGE [49], 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.

Human evaluation: In order to evaluate how developers perceive the generated logging texts, we also performed a human evaluation, which is detailed in Section 5.

3.3 Baseline approach

We compare our approach with prior work by He et al. [29], 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. [29] perform a search in the training corpus to retrieve the most similar code snippet based on Levenshtein distance [37]. 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.

[4]https://github.com/conf-202x/experimental-result
[5]The sizes of training datasets range from 1K to 9k.

4 EVALUATION RESULTS

In this section, we discuss the results of evaluating *LoGenText* and *LoGenText-Plus* through answering six research questions. More specifically, the first three research questions (i.e., RQ1-RQ3) are related to *LoGenText*, and RQ4-RQ6 are newly proposed research questions and related to our extension, *LoGenText-Plus*.

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

Motivation.

Prior research [29] has observed that logging texts are predictable and proposes a simple approach (the baseline approach in Section 3) 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 [29]. Following prior work [29], we evaluate the quality of the generated logging texts using two widely used metrics for machine translation evaluation, i.e., BLEU[6] [65, 66] and ROUGE[7] [49]. 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:

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

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

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 [29]. 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.

[6]https://github.com/mjpost/sacrebleu
[7]https://github.com/pltrdy/rouge

Table 2. Evaluation results of using *LoGenText* and the baseline approach to generate logging texts in the studied projects (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.

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 2. 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 2, *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 2, *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 [9], 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 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 5.

Summary

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 [5, 36, 56, 57, 78, 79, 83, 98] 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 [3, 7, 32, 76, 99]. 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.

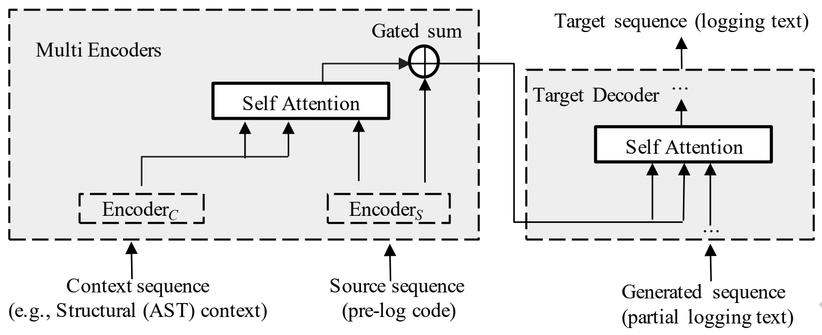
We propose a context-aware form of *LoGenText* and consider two types of context information in this research question: the post-log code and the structural (AST) information related to a logging statement. Below we discuss how we extract such context information and incorporate it in *LoGenText*.

Extracting context information. *Extracting the structural (AST) context:* We use AST extracted by *srcML* [12] 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* [104] and *how to choose log levels* [41]. 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 preorder 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 [29] 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 2) 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 [1, 75], and (2) utilizing a multi-encoder model, where additional neural networks are used to encode the context [38, 79, 98]. Prior work [38, 79] 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 multi-encoder approach shows better performance in our context. Therefore, we use the multi-encoder approach in this paper.

The structure of our context integration approach is illustrated in Figure 4. 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



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

Fig. 4. An overview of the multi-encoder Transformer.

Table 3. 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.

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 $\sigma(\cdot)$ 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 3 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 a 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 3, 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, sizes (i.e., BLEU-3 and BLEU-4), 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 [41, 104].

We also find that incorporating additional context may not always improve the performance of *LoGenText* significantly. As shown in Table 3, 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.

Additionally, to gain a deeper understanding of why utilizing AST context is more beneficial, we conduct a comprehensive manual analysis. First, we sort the cases in descending order based on the BLEU score gap between utilizing the AST context and utilizing the post-log code. This sorting allows us to identify the cases where the utilization of AST context has the most significant impact on performance improvement. Subsequently, we select the top 10 cases from each project, resulting in a total of 100 cases.

We find that most (i.e., 90%) of the logging statements, where utilizing AST outperforms relying on the post-log code, are describing the preceding source code (i.e., pre-log code). As a result, the succeeding code (i.e., post-log code) would be a source of noise and has a negative impact on the generation of the logging text.

Moreover, to understand how the post-log code would be a source of noise during the logging text generation process, we further manually analyze the characteristics of the post-log code. We find that the noise mainly comes from two aspects: 1) For a testing case, there exist training cases that share exactly the same post-log code, but a different pre-log code. Therefore, utilizing the post-log code would cause *LoGenText* to (partly) copy from such existing logging texts. It is intuitive that the post-log codes are similar, as developers may put return or exception statements at the end of a method; 2) The post-log code contains irrelevant tokens and thus misleads the generation of the logging text.

In short, all the noisy information can be summarized as the introduction of irrelevant code to the source input. As a result, *LoGenText* cannot effectively focus on the most important source code to generate the logging text. Moreover, we find that even though in some cases where the logging texts are describing the succeeding source code, they are only related to one or two lines of post-log code. Therefore, using all the post-log code as the context in *LoGenText* would sometimes decrease the performance of our approach.

Summary

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?

Table 4. 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.

Motivation.
Prior work [29] 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. Given that including the information provided by the baseline may further improve the results, in this research question, we aim to explore the impact of incorporating logging text in similar code on automated logging text generation and examine whether we can improve the base form of *LoGenText* by utilizing such logging information.
Approach.
Similar to prior work [29], we leverage the logging texts from similar code snippets 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 [29] and use the Levenshtein distance [37] to calculate the similarity between it and 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 4, 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.

Summary

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.

RQ4: Can incorporating the template information into *LoGenText-Plus* improve *LoGenText* in generating logging texts?

Motivation.

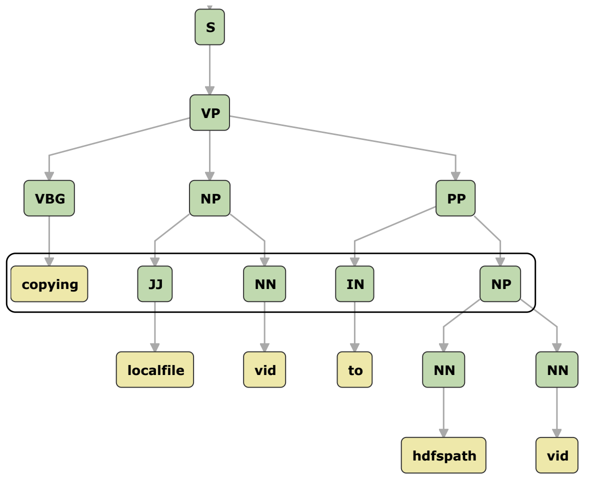
Prior studies [18, 27, 81, 85, 90] on text generation tasks (e.g., text summation, sentences generation) show that incorporating the template information of the target sentences can provide promising results in generating better texts. For example, Yang et al. [90] use syntax-based templates to guide the translation procedure and outperform the baseline models in the task of neural machine translation. In addition, based on the manual inspection of the two generated logging texts in the section of human evaluation provided by Ding et al. [14], we find that the syntactic structures of the two logging texts are different from each other and carry distinct information for each logging text. Such variety and specificity raise the question of whether we can extract syntactic templates from the syntactic structures and adopt templates to guide the automatic logging text generation process. Figure 5 are two different constituency-based parse trees produced by Stanza [67] for two logging texts. Based on these two syntactic trees, we may construct two syntactic templates (the construction process is elaborated in the following section ***Approach***.), "copying JJ NN IN NP" and "no beanstalks defined IN NP", where JJ NN IN NP are non-terminal symbols of the parse tree, representing different token syntactic abstractions (e.g., NP refers to noun phrase and NN means noun). As illustrated in Figure 5, templates are abstract representations of logging texts that encompass the syntactic characteristics of logging texts and may serve as a guide when generating logging texts. Therefore, in this research question, we aim to understand whether the syntactic template information can help further improve *LoGenText* in automatically generating logging texts.

Approach.

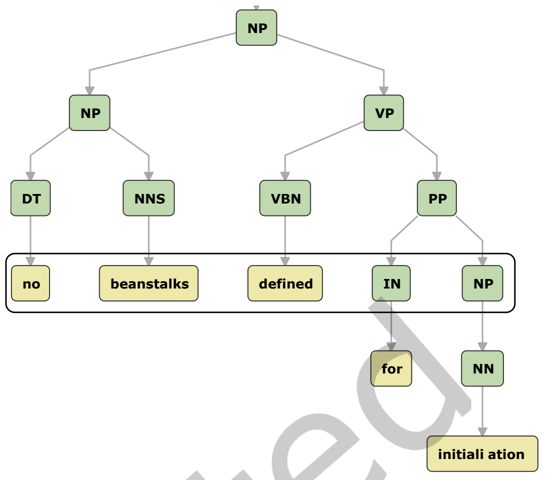
To answer our research question, we propose *LoGenText-Plus*, which uses the pre-log code and the logging template as the source and AST as the context to generate the logging text. Unlike *LoGenText*, *LoGenText-Plus* not only extracts the three types of information (i.e., logging text, the pre-log code and the context information) used in *LoGenText* but also considers the syntactic template of logging texts. As stated in Section 2.1.2, *LoGenText-Plus* contains two stages, (1) template generation and (2) template-based logging text generation.

Stage 1: template generation. *LoGenText-Plus* uses a Transformer-based Seq2Seq model to generate the templates for the given source input. To effectively incorporate the template and AST information, *LoGenText-Plus* considers concatenating the template from the logging text in the similar code with the pre-log code (i.e., the source input in *LoGenText*) as the new input to the source encoder, and AST as the context to the context encoder. Below, we describe how we extract the template and incorporate it into *LoGenText-Plus*.

Extracting the template from the logging text in the similar code. Similar to RQ3, in this step, we assume that similar code snippets would have similar logging templates, and incorporating the logging template in similar code would ease the process of predicting the target templates. For example, "failed to unregister vid" is a logging text from the project ActiveMQ, and the logging text in its similar code is "failed to dispose of vid". Both can have the same syntactic template "failed TO VP". In RQ3, we have extracted logging text from similar code based



(a) The constituency-based parse tree of the logging text, "copying localfile vid to hdfspath vid".



(b) The constituency-based parse tree of the logging text, "no beanstalks defined for initialization".

Fig. 5. Two constituency-based parse trees for the logging texts. The no-terminal (i.e., syntactic tags) and terminal (i.e., tokens in logging text) symbols delimited by the black lines can be selected to construct the templates.

on the given pre-log code snippet. In this step, we first use *Stanza* [67], an open-source NLP library, to perform constituency parsing for each logging text from similar code. Constituency parsing is the task of analyzing phrase structures (e.g., simple declarative clauses, verb phrases, noun phrases, etc.) for a given sentence. Figure 5 shows two parsed trees, where the terminal symbols (or, leaf nodes in the tree) are tokens in the logging text, and non-terminal symbols are syntactic categories (e.g., "S" for simple declarative clause, "NP" for a noun phrase and "VP" for a verb phrase).

After having the consistency-based parse tree, we choose a certain depth of the constituency-based parse tree to construct the template. Then, all the symbols (including both terminal and non-terminal symbols) at the pre-defined depth are collected as the template. For example, assuming the depth is four, then the template for the logging text, "copying localfile vid to hdfspath vid", is "copying JJ NN IN NP". The depth ranges from one to the maximum depth of the generated tree (e.g., six for both examples in Figure 5), resulting in different templates for the logging text. Among all the templates, one special case is that we set the depth to a number larger than the maximum depth of the tree, the templates are exactly the same as the logging texts. With the decrease of the depth, the complexity of the templates (e.g., the length of the template and the number of unique tokens in the template) is also reduced.

In this research question, we start with the depth of one due to the following reasons: 1) each template contains only one token and should be easier to predict, and 2) even though the template contains only one token, it still can convey different syntactic information (e.g., "S" for simple declarative clause and "NP" for noun phrase as shown in Figure 5). Note that this may be different from the text generation tasks (e.g., machine translation) in NLP, where the target text is usually a complete sentence (that is if we set the depth to one, the template may always be "S".). On the contrary, for the logging text in the source code, some developers prefer to use noun phrases while others may use complete sentences to monitor the status of the software. Moreover, the experiment in RQ5 also demonstrates that our choice is optimal for automatic logging text generation.

Concatenating the pre-log code and the template. In previous research questions (i.e., RQ1-RQ3), the source is the pre-log code. However, in this step, we consider concatenating the pre-log code and the template from the logging text in a similar code as the new source input to the source encoder of the Transformer-based model. We adopt this incorporation strategy due to the following considerations: 1) by doing the input concatenation, the newly formed input sequence is almost a complete code structure (i.e., the pre-log code and the template of the logging text from similar code). In other words, in the newly formed input sequence, the template acts as a placeholder, which needs to be refined or replaced by the output of the Transformer-based model. 2) Besides, under this setting, we can still integrate the AST information using our multi-encoder strategy which has been proven to be useful for generating the logging text [14].

Extracting the template from the target logging text. The goal of this stage is to predict the template based on the source input, thus, we need to construct a new target sequence, which is the template of each logging text. As we have already collected the logging text from each logging statement in Section 2.2, in this step, we share the same way of extracting the template from the logging text in the similar code to extract the template from the logging text.

By now, we have the source sequence (i.e., the concatenated pre-log code and the template from the logging text in similar code), target sequence (i.e., the template from the target logging text) and context information (i.e., AST information extracted in RQ2). Next, we describe how we integrate the context information into the model used for generating the template.

Integrating context information in our models. In RQ2, *LoGenText* utilizes another encoder to encode the context information. As shown in Figure 4, the outputs of the context encoder and source encoder are converted into a new representation by an attention layer, which is finally fused with the output of the source encoder by a gated sum. The input of the target decoder, G, is a deep hybrid of both the source and the context inputs. However, this design may have one limitation, that is the deep hybrid happens at the encoder part, as a result, the context information may not be passed into the decoder part effectively and the influence of the context may vanish after the attention and gated sum operations. Hence, in this step, we adopt another design to incorporate the context information at the decoder part [38, 98].

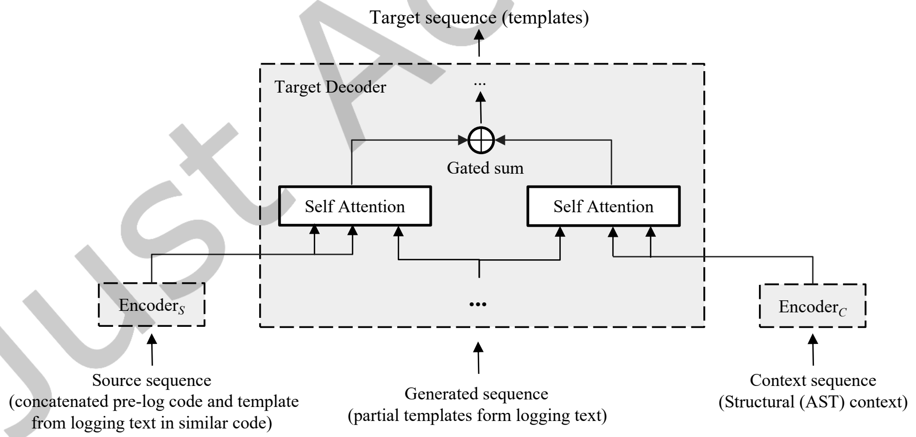

Target sequence (templates)
Target Decoder
...
Gated sum
Self Attention
Self Attention
Encoder$_S$
...
Encoder$_C$
Source sequence (concatenated pre-log code and template from logging text in similar code)
Generated sequence (partial templates form logging text)
Context sequence (Structural (AST) context)

Fig. 6. An overview of the new multi-encoder Transformer used for template generation. The left "Self Attention" is the source attention layer and the right "Self Attention" is the context attention layer.

The structure of our new context integration approach is illustrated in Figure 6. Similar to the context encoder in the multi-encoder Transformer used in RQ2, the context encoder replicates the original encoder Transformer and takes the AST as the input. The output of the context encoder and the output of the source encoder are then passed to the target decoder separately, where the two outputs together with the previously generated template sequences are fed into self-attention layers (i.e., context attention layer and source attention layer as shown in Figure 6), respectively. Then, the outputs of the attention layers are fused by a gated sum. Formally, let S' be the output of the source attention layer and C' be the output of the context attention layer, the output of the gated sum G is

$$G = \lambda \odot C' + (1 - \lambda) \odot S' \tag{10}$$

where the gating weight λ is calculated by

$$\lambda = \sigma \left(W \left[C', S' \right] + b \right) \tag{11}$$

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

Finally, the template generation stage produces a template for each of the newly constructed source input (i.e., Generated templates in Figure 2), which is later used for the template-based logging text generation (i.e., the model inference in Figure 2).

Stage 2: template-based logging text generation. *LoGenText-Plus* adopts the same model structure as the model used in template generation for template-based logging text generation. During the model training of logging text generation, *LoGenText-Plus* concatenates the template from the logging text in the similar code with the pre-log code (i.e., the source input in *LoGenText*) as the new input to the source encoder, and AST as the context to the context encoder, while the target sequence is the logging text instead of the template. During the model inference, *LoGenText-Plus* concatenates the pre-log code with the template produced in stage 1 as the new input to the source encoder. The output is the final prediction of the logging text. Note that during model training, we intentionally use the template from the logging text in the similar code instead of the template from the corresponding logging text, because the former contains noise that can be used to simulate the prediction errors in the generated template. Otherwise, during model training, if we use the template from the corresponding logging text, the model would pay more attention to the template part. However, during inference, the generated template may contain errors, which would mislead the model, thus, resulting in a poor quality of the generated logging text. Our experiment results also confirm our assumption.

In order to understand the impact of the syntactic template information, similar to previous RQs, we evaluate the performance of *LoGenText-Plus* on all the subject projects, where we train separate models for each project (cf. Section 3.2.1).

Besides, we conduct another experiment to study how the diversity across different projects would impact our approach. We first train a single model using the combined ten training datasets and evaluate its performance on each of the ten individual projects. Furthermore, we extend the evaluation to include a new dataset from the project Cassandra, allowing us to assess the generalizability of our approach to unseen data. We select Cassandra as it is widely studied in the literature [45, 47, 52, 53, 93, 101].

We use the same metrics used in RQ1 (i.e., BLEU and ROUGE-L) to evaluate the quality of the generated logging texts.

Results.

Overall, our newly proposed approach *LoGenText-Plus* outperforms the baseline approach as well as the best performing version of *LoGenText* that incorporates the AST context. The experimental results on the 10 studied projects are provided in Table 5 with the best results highlighted in **bold**. In particular, *LoGenText-Plus* outperforms *LoGenText* in eight out of 10 projects in terms of BLEU score. For example, we obtain over 10% relative increase (i.e., 12.3%) for the project Brooklyn in terms of BLEU score and a 5.9% increase in ROUGE-L score. Our results indicate the effectiveness of incorporating templates in guiding the automated generation of logging text.

In addition to the overall BLEU and ROUGE-L, *LoGenText-Plus* can provide relatively good performance for relatively small projects. As shown in Table 5, *LoGenText-Plus* achieves a BLEU score of 25.5 to 37.9 for the ActiveMQ, Ambari, Brooklyn and Synapse projects, which are the smallest projects with less than 500K SLOC.

Table 5. Evaluation results of using *LoGenText-Plus*, *LoGenText* and the baseline approach to generate logging texts in the studied projects (RQ4).

	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
LoGenText with AST (RQ2)	24.1	23.8	27.8	**41.8**	34.6	23.3	23.5	29.6	**28.8**	37.2	29.5
LoGenText-Plus	**26.6**	**25.5**	**31.2**	40.1	**35.0**	**23.8**	**23.7**	**30.3**	**28.8**	**37.9**	**30.3**
	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
LoGenText with AST (RQ2)	42.5	43.4	44.0	53.9	**50.8**	42.1	46.4	**48.2**	**47.6**	53.6	47.3
LoGenText-Plus	**44.4**	**44.0**	**46.6**	**54.0**	50.1	**42.5**	46.2	47.6	47.3	**54.3**	**47.7**

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

Table 6. Evaluation results of incorporating templates with different multi-encoder Transformers for logging text generation (RQ4).

	BLEU(%)										
	ActiveMQ	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**
Integrating context at decoder	**26.6**	**25.5**	**31.2**	40.1	35.0	**23.8**	**23.7**	30.3	28.8	**37.9**	**30.3**
Integrating context at encoder	26.0	23.3	31.0	**42.3**	**35.1**	23.6	23.3	**30.6**	**29.4**	36.0	30.1
	ROUGE-L(%)										
	ActiveMQ	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**
Integrating context at decoder	**44.4**	**44.0**	**46.6**	54.0	50.1	**42.5**	46.2	47.6	47.3	**54.3**	47.7
Integrating context at encoder	43.5	43.3	46.5	**54.5**	**50.7**	41.7	**46.4**	**48.8**	**48.2**	53.5	47.7

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

LoGenText-Plus has the largest BLEU improvements on three of these four smallest projects (i.e., 10.3%, 7.1%, 12.3% relative increases on projects ActiveMQ, Ambari, Brooklyn respectively) over the best performing form of *LoGenText*. It is widely recognized that deep neural networks usually require larger training data to generalize better [24, 43]. However, our results indicate that our *LoGenText-Plus* could alleviate the (negative) impact of limited training data and effectively generate logging texts for smaller projects.

However, we also observe that *LoGenText-Plus* may not always improve the performance significantly. For example, as Table 5 shows, the improvement of BLEU score on some relatively larger projects (e.g., HBase, Hadoop and Ignite) is limited, and *LoGenText-Plus* performs even worse than *LoGenText* on the project Camel. This phenomenon may be explained from two aspects: 1) The size of the project: the project contains more lines of source code, which provides more source information and training sets for learning a good model, as a result, the impact of the template can be mitigated, and 2) The different context integration strategies: as we stated in RQ4-***Approach***, we adopt another integration strategy, where we move the context integration from the encoder part to the decoder, trying to enlarge the impact of the context. On the other hand, it should be noted that the context may contain noise, as a result, the impact of the noise is also increased, which makes the performance even worse. Thus, we further conduct another experiment on the 10 projects using the integration strategy proposed by *LoGenText*. The results are presented in Table 6. The results confirm that the integration strategy does have an impact on the performance of logging text generation. Specifically, we get a BLEU score of 42.3 on project Camel, which is higher than *LoGenText-Plus* and *LoGenText* (i.e., 40.1 and 41.8, respectively). This observation also matches previous studies that show that enlarging the context may lead to performance degradation of NMT models due to the noise introduced by the enlarged context [98, 102].

Table 7. Evaluation results of different training strategies for logging text generation in the studied projects and a new project (RQ4).

BLEU(%)

	ActiveMQ	Ambari	Brooklyn	Camel	CloudStack	Hadoop	HBase	Hive	Ignite	Synapse	Avg.	Cassandra
Separate models	**26.6**	**25.5**	**31.2**	**40.1**	**35.0**	**23.8**	**23.7**	**30.3**	**28.8**	**37.9**	**30.3**	**20.1**
Single model	22.1	21.9	27.2	36.9	32.8	20.9	22.0	27.5	26.5	34.1	27.2	11.3

ROUGE-L(%)

	ActiveMQ	Ambari	Brooklyn	Camel	CloudStack	Hadoop	HBase	Hive	Ignite	Synapse	Avg.	Cassandra
Separate models	**44.4**	**44.0**	**46.6**	**54.0**	**50.1**	**42.5**	**46.2**	**47.6**	**47.3**	**54.3**	**47.7**	**42.0**
Single model	41.5	42.1	42.9	50.1	49.1	40.2	45.4	46.4	46.5	51.5	45.6	31.2

Note: Values in bold font indicate the best performing models. The single model of Cassandra is trained on the 10 studied projects.

Additionally, we find that training separate models for each project (cf. Section 3.2.1) can benefit the performance of our approach. Table 7 shows the results of the different training strategies, as well as the performance of the model trained on the 10 studied projects, when applied to the new project, Cassandra. As the table shows, by training only one single model on all the training datasets, the performance of *LoGenText-Plus* decreases on almost all the projects and reaches an average BLEU score of 27.2, which is lower than that of the separate models (i.e., 30.3). The results may be due to the fact that the training data from other projects would bring some noise, and thus may negatively impact the performance of the model. Furthermore, when evaluating the single trained model on the unseen dataset (i.e., Cassandra), the model gives a BLEU score of 11.3. Meanwhile, we also evaluate the model that is trained on the project Casandra and the model has a BLEU score of 20.1. For comparison, we further evaluate the baseline approach under these two settings, which gives BLEU scores of 8.8 and 15.7 respectively. On the one hand, the results confirm the findings from previous work [29] that the language patterns in different projects vary a lot. On the other hand, the results reveal one of the limitations of our approach: although our approach has better performance than the baseline, there is still a non-negligible performance drop when the training and testing datasets are drawn from different distributions. The results call for future research that can alleviate such performance decreases across different distributions.

Although *LoGenText-Plus* exhibits improved performance compared to *LoGenText*, and both approaches surpass the baseline approach, it is important to acknowledge that there is still potential for further improvement in both approaches. To better understand the limitations and instances requiring enhancement, we conduct another manual analysis on instances where both *LoGenText* and *LoGenText-Plus* do not achieve satisfactory performance. By closely examining these cases, we aim to identify the specific challenges and factors contributing to suboptimal outputs.

In particular, we check the distribution of the BLEU scores for the generated logging texts and find that, on average, approximately 35 cases yield a BLEU or ROUGH score of zero. To gain further insights into these cases, we randomly selected 10 cases for each project, which we would thoroughly analyze.

We have categorized the characteristics of the cases into two main categories, each with several subcategories[8]:

Limited source input. Both *LoGenText* and *LoGenText-Plus* rely on the pre-log code as the source input, and in some cases, the pre-log code may lack sufficient information for logging text generation, resulting in unsatisfactory outputs. We have identified two common scenarios: 1) The logging statement is put at the beginning of the method of which the method name is very simple and common and does not provide meaningful information. For instance, Figure 7a presents an example where the corresponding code (i.e., lines 2 and 3) only contains a few tokens and does not provide much information. 2) The logging statement describes the post-log code, while in our approaches, we use the preceding code as the input, which would result in the wrong output. As shown in

[8]Note that these categories may not be strictly exclusive. For instance, in Example (a), the code (i.e., lines 2 and 3) is very common and there is a high possibility that it appears in the training set.

```
1  @Override
2  public void run() {
3    try {
4  --------------Candidate log start---------------
5  Original log: LOG.debug("Executing task #{}", taskId)
6  Generated log-ast: logsearchfilenamerequestrunnable starting
7  Generated log-plus: persisting metric metadata
8  --------------Candidate log end---------------
9  ...
```

(a)

```
1  ...
2  try {
3  --------------Candidate log start ---------------
4  Original log: LOG.debug("Setting subscriptions: {}", ...)
5  Generated log-ast: dispose old state
6  Generated log-plus: disposition disconnect
7  --------------Candidate log end ---------------
8  connected.putSubscriptions(this.subscriptions);
9  ...
```

(b)

```
1  private void printHelp() {
2      ...
3      logger.info("Default values:");
4
5  ---------------Candidate log start----------------
6  Original log: logger.info(DOUBLE_INDENT + "HOST_OR_IP
7                                       =" + DFLT_HOST)
8  Generated log-ast: default values
9  Generated log-plus: default values
10 ---------------Candidate log end----------------
11 ...
```

(c)

```
1  ...
2  if (jobScheduler != null) {
3      jobScheduler.removeJob(jobId);
4      LOG.info("Removed scheduled Job " + jobId);
5  } else {
6  ---------------Candidate log start---------------
7  Original log: LOG.warn("Scheduler not configured")
8  Generated log-ast: removed scheduled job vid
9  Generated log-plus: removed scheduled job vid
10 ---------------Candidate log end----------------
11 ...
```

(d)

Fig. 7. An illustration of error cases generated by *LoGenText* and *LoGenText-Plus*.

Figure 7b, line 8 is the corresponding code that the logging statement (i.e., line 4) is describing, but our approaches ignore such information.

Similar source input in the training set. There are some test cases that share a very similar input with the training cases, where both approaches may simply copy the logging texts from the training cases. Specifically, 1) There are two consecutive logging statements in the original source code and one of them is used as the training case. For example, line 3 of Figure 7c appears in the training set and has the same source input as line 4 (i.e., the logging text to generate). 2) There are two logging statements that are close to each other in the original code. For example, in Figure 7d, the logging statement in the if block (i.e., line 4) is used as the training set, and when generating the logging statement in the else block (i.e, line 7), our approaches may simply copy the logging text from line 4, due to the minimal difference in source input.

Based on these findings, future work may consider 1) incorporating the data flow information to discriminate similar source input, 2) utilizing the method call graph to identify more context for the logging text at the beginning of a method, and 3) identifying more relevant source code, while avoiding the introducing of noise.

Summary

LoGenText-Plus generally outperforms the baseline approach that leverages the existing logging texts in similar code snippets as well as the best performing version of *LoGenText* that incorporates the AST context. Our results illustrate the effectiveness of using templates for guiding the generation of the logging text.

Table 8. Evaluation results of incorporating templates constructed with different depth in *LoGenText-Plus* for logging text generation (RQ5).

BLEU(%)											
Depth	**ActiveMQ**	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**
1	26.6	**25.5**	31.2	40.1	**35.0**	**23.8**	23.7	**30.3**	28.8	37.9	**30.3**
2	**28.9**	24.4	**31.5**	**40.8**	34.2	22.9	24.2	29.0	28.9	36.4	30.1
3	28.2	24.6	29.3	40.6	34.5	22.1	24.1	29.1	**30.5**	37.4	30.0
4	24.8	23.4	28.2	40.2	**35.0**	21.9	24.3	29.0	29.2	35.0	29.1
5	25.9	23.2	30.5	39.6	33.2	22.3	22.7	28.8	28.1	**38.0**	29.2
6	26.4	23.2	29.0	**40.8**	34.0	21.9	23.7	28.4	27.7	33.4	28.9
7	25.6	23.9	30.4	39.3	33.8	21.8	24.2	28.9	27.6	34.6	29.0
8	26.1	22.2	27.6	39.6	33.7	21.2	23.3	28.8	28.5	34.1	28.5
9	27.8	23.6	27.3	38.9	33.2	21.5	**24.6**	29.0	26.9	35.7	28.8
Best	**28.9**	**25.5**	**31.5**	**40.8**	**35.0**	**23.8**	**24.6**	**30.3**	**30.5**	**38.0**	**30.9**

ROUGE(%)											
Depth	**ActiveMQ**	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**
1	44.4	**44.0**	46.6	**54.0**	50.1	**42.5**	46.2	47.6	47.3	54.3	**47.7**
2	44.6	42.2	**46.7**	52.3	**50.8**	41.0	46.6	47.3	48.3	52.4	47.2
3	**44.8**	43.2	45.9	53.3	50.2	41.2	47.0	47.4	**48.6**	**55.2**	**47.7**
4	43.5	42.7	45.5	52.1	50.4	41.5	46.9	47.3	47.7	53.6	47.1
5	42.4	43.5	44.4	53.0	48.9	40.7	45.2	**48.0**	47.2	54.0	46.7
6	43.5	42.4	45.1	53.2	49.9	40.1	46.8	47.2	47.0	53.6	46.9
7	42.7	43.8	45.8	52.4	49.9	41.3	**47.7**	47.7	46.5	52.3	47.0
8	42.7	42.4	44.7	53.0	50.0	40.2	46.2	47.2	48.5	51.7	46.7
9	44.5	43.3	43.5	52.5	49.8	40.7	**47.7**	47.5	46.3	53.5	46.9
Best	**44.8**	**44.0**	**46.7**	**54.0**	**50.8**	**42.5**	**47.7**	**48.0**	**48.6**	**55.2**	**48.2**

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

RQ5: How does the granularity of logging templates impact the performance of *LoGenText-Plus* in generating logging texts?

Motivation.

In RQ4, we have introduced the approach of how to build the templates from the consistency-based parse tree and have shown the effectiveness of using these templates. On one hand, we start building the templates with the depth of one, which produces simple yet effective templates. On the other hand, choosing different depths can result in diverse templates, which may convey different levels of information for guiding the generation of logging texts. For example, as shown in Figure 5b, if we set the depth to one, the template only has one symbol "NP", which means that the logging text is a noun phrase. The template is very short and cannot capture too much information. Meanwhile, if we set the depth to six (or a larger number), the template is exactly the same as the logging text, which violates our assumption. Therefore, in this research question, we aim to explore the impact of incorporating different templates derived from the tree with different depths on automated logging text generation.

Approach.

In this RQ, we adopt the same approach as in RQ4 to construct and incorporate the templates. In particular, we extract the templates with depths from one to nine and concatenate them with the pre-log code, separately. Based on the newly constructed source input, we train different models and evaluate the performance of *LoGenText-Plus* that incorporates different templates using the BLEU and ROUGE-L metrics.

Results.

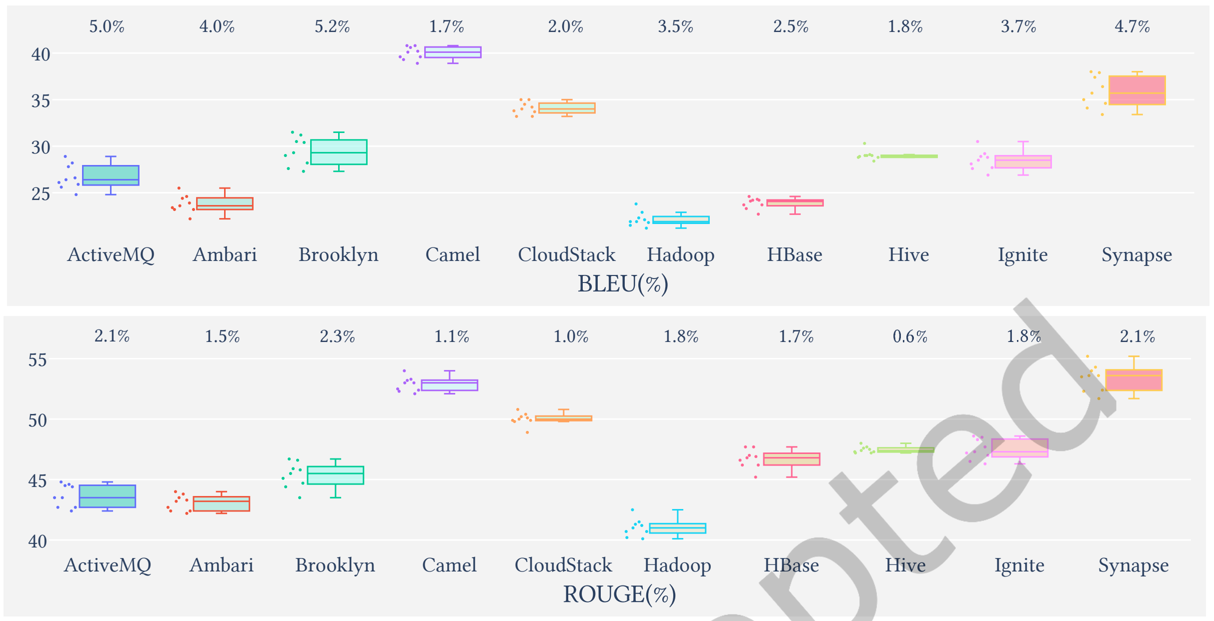


Fig. 8. The distribution of the results produced by *LoGenText-Plus* with different templates. The horizontal axis represents all the studied projects; the vertical axis is the performance (i.e., BLEU or ROUGE) in different projects. The numbers on top of each box are the corresponding coefficient of variance.

The performance of *LoGenText-Plus* on the subject systems can be further improved by incorporating templates with different depths compared to that of *LoGenText-Plus* using templates with a depth of one. As shown in Table 8, we find that by incorporating the new templates constructed with different depths, the performance of *LoGenText-Plus* can be increased in half of the 10 studied projects (e.g., the average BLEU score increases from 29.5 (i.e., *LoGenText* with AST, cf. RQ2) to 30.9 (i.e., the best performing depths)). The biggest improvement is observed in the project ActiveMQ. When setting the depth to two, *LoGenText-Plus* achieves a BLEU score of 28.9, which is 19.9% and 8.7% higher than *LoGenText* (i.e., 24.1, cf. RQ2) and *LoGenText-Plus* with the depth of one (i.e., 26.2, cf. RQ4). The results indicate that the templates constructed with different depths may capture various types of knowledge to help with automatic logging text generation. With proper depth, the template can further improve *LoGenText-Plus* for the studied projects.

However, similar to our findings in RQ2, incorporating templates with different depths can result in diverse performance for each project. Figure 8 shows the distribution of performance results produced by *LoGenText-Plus* with different templates. To quantify the variance of the differences, we also calculate the coefficient of variation (CV) for each project. The results show that the projects with a relatively small size (SLOC), are more sensitive to the templates. As Figure 8 shows, the four smallest projects, ActiveMQ, Ambari, Brooklyn, and Synapse projects have the largest variances, 5.0%, 4.0%, 5.2%, 4.7%, respectively. While for large projects, the impact of utilizing different templates is relatively weak. For example, for the project Synapse, using the template with a depth of six can only have a BLEU score of 33.4, compared to a BLEU score of 38.0 when using the template with a depth of five, but for the project Hive, the biggest performance different is 1.9 (i.e., 30.3 when depth is one vs. 28.4 when depth is six). One explanation for this phenomenon may be that larger projects may contain more source data, and thus, produce more powerful models and mitigate the differences between different templates.

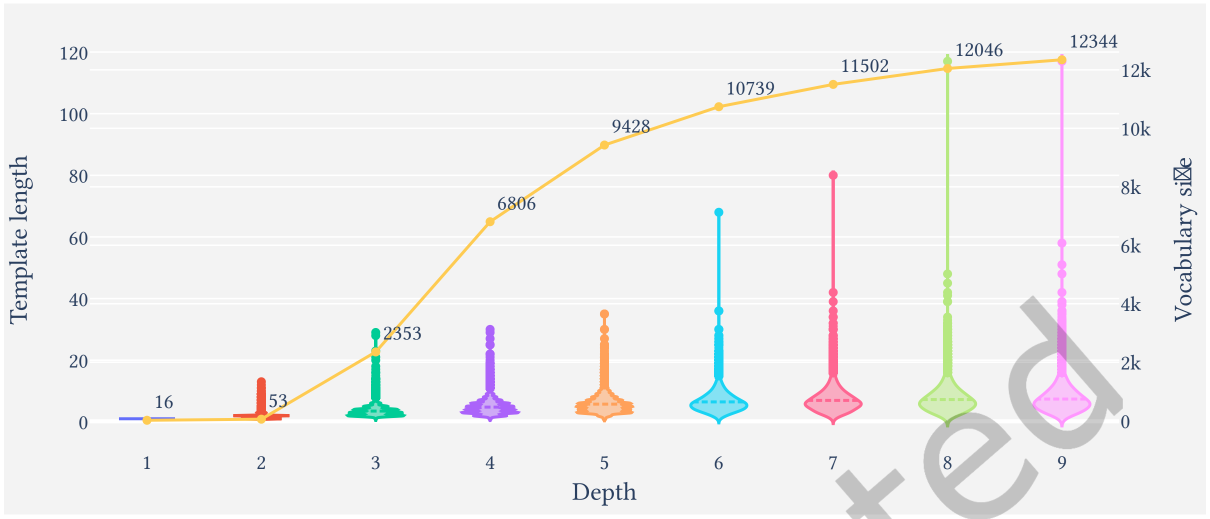


Fig. 9. The distribution of the length of templates (i.e., violin plot) constructed with different depths and the vocabulary size (i.e., line plot). The horizontal axis represents the different depths; the left vertical axis is the template length (i.e., the number of tokens in each template); the right vertical axis is the vocabulary size (i.e., the number of distinct tokens in all the templates).

In addition, as shown in Table 8, with the increases in depth, the average BLEU score tends to decrease. In particular, when the depth is one, *LoGenText-Plus* has the best BLEU and ROUGE scores on average. Besides, for most of the projects, *LoGenText-Plus* achieves the best performance when the depth is less than five. For example, the BLEU scores on projects ActiveMQ, Brooklyn and Camel reach the highest with a depth of three. This is reasonable given that with the increases in depth, more terminal symbols (tokens in logging text) are captured, resulting in more complex templates. Figure 9 shows the distribution of the length of templates under different depths as well as the vocabulary size (i.e., the number of unique tokens in templates). It is obvious that both the length of templates and the vocabulary size increase significantly with the increases in depth. As a result, the task of template generation becomes more difficult, which means that *LoGenText-Plus* may not generate accurate templates, and thus negatively impacting the performance of generating the logging texts.

Discussion

In the above sections, we have quantitatively demonstrated the superiority of *LoGenText-Plus* on the 10 subjects. Based on the extensive experimental results, in this part, we would like to elaborate more on the design choice as well as the potential limitations of *LoGenText-Plus*.

Strengths and limitations

In *LoGenText-Plus*, we divide the logging text generation task into two stages: template generation and template-based logging text generation. Therefore, the advantages of *LoGenText-Plus* can be discussed from two aspects:

- Design paradigm: Instead of directly predicting the logging texts, we are trying to solve the problem with a coarse-to-fine strategy. Intuitively, predicting the templates is a little easier compared to predicting the logging texts directly, as 1) the size of the symbols (about 60 tags) in the templates is much smaller than the number of tokens (i.e., natural language words plus source code tokens) in the logging texts, and 2) the templates are relatively shorter than the corresponding logging texts. In particular, if the depth is set to one, the template only contains one element. As a result, we are actually converting the template generation task to a multi-class classification problem, which should be easier to tackle.

- Enriched information: By incorporating the templates of the logging texts, we are explicitly introducing more knowledge into the model. Previous works [14, 29] mainly focus on the information extracted from the source code, such as the abstract syntax trees, and code sequences, while few works consider utilizing the information from the target sequence. In this work, the template is constructed from the constituency-based parse tree of the logging text which provides a syntactical representation of the logging texts. The low-level syntactic information (i.e., syntactic tags) with the high-level semantic information (i.e., some tokens in the logging text) in the template complements the source code and thus may be useful for the generation of the logging texts. This finding also explains the experimental results in RQ4 that *LoGenText-Plus* can provide relatively good performance for relatively small projects, as *LoGenText-Plus* can bring more external knowledge to the trained model.

However, *LoGenText-Plus* also has limitations. Although using the template may provide useful information, there is still a chance that the generated templates contain noise, and thus may harm the performance of *LoGenText-Plus*. As the experimental results in RQ5 show, with the increases in depths, predicting the templates tends to be a more challenging task. As a result, the generated templates may be of poor quality with a risk of misleading the generation process of the logging texts. Moreover, we also observe that for some projects, *LoGenText-Plus* does not bring significant benefits, especially when the size of the projects is large enough. For example, *LoGenText-Plus* has the same BLEU score on the Ignite project with *LoGenText*. This may be due to the fact that larger datasets may produce more powerful models and mitigate the differences caused by the external information (i.e., syntactic information from the logging texts.). In future work, to further improve the performance of *LoGenText-Plus*, we can focus on generating more accurate templates.

Another limitation may come from the fact that we use the pre-log code as the source input, while there may exist some logging texts describing the succeeding source code. Due to the lack of information on the corresponding source code, our approach may fall short of generating satisfactory logging texts. In this work, the design choice of only considering the pre-log code as the main source input is based on the findings from previous work by He et al. [29], where they find that using the code surrounding (i.e., code preceding and succeeding) the logging text would result in worse results, compared to only using the code preceding the logging text.

We also conduct another two experiments on *LoGenText* and *LoGenText-Plus*, where we concatenate the pre-log code and post-log code as the source input and utilize the AST as the context. We find that both *LoGenText* and *LoGenText-Plus* have a performance degradation, with an average BLEU score of 27.6 and 26.7 respectively, compared to the initial BLEU scores of 30.1 and 30.3 without the use of post-log code. The results show that incorporating more contexts can not always guarantee a better model, which conforms to the results in the work of He et al. [29]. This phenomenon can be explained by the fact that developers prefer to insert logging statements to describe the actions in the preceding source code [17, 29], and thus, the use of post-log code may bring in some noise. Especially for *LoGenText-Plus*, as it involves the use of post-log code in two stages: template generation and logging text generation. The errors in the generated template may propagate to a later stage, and mislead the generation of the logging texts. Meanwhile, there is a possibility that the use of all the pre-log code might not be optimal, as the logging statements may only describe a few lines of code (e.g., the example in our introduction section). Future work may consider improving the performance by identifying the most relevant source code as input.

Summary

The performance of *LoGenText-Plus* on the subject systems can be further improved by incorporating different templates with different depths. However, the selection of the templates is a trade-off, as incorporating templates with different depths can result in diverse performance for each project.

RQ6: Can we harmonize the wording in the generated logging text with the logging text written by developers using *n*-gram dictionaries?

Motivation.

Prior work [14] proposes an NMT-based model to automatically generate the logging text based on the source code (i.e., *LoGenText*) and achieves promising results. However, while reviewing the generated logging texts, they find that the generated text sequence and the text sequence in the developer-written logging text may not always be consistent. For example, the logging text in Figure 5a uses the term “localfile”, while the generated logging text uses the noun phrase “local file”. Although these two terms have a very similar meaning for developers, the use of different words may cause inconsistency in the wording and affect downstream log-related tasks (e.g., log parsing, log compression). Meanwhile, it is obvious that the term “localfile” should be an identifier from source code (e.g., method names, or variables) and is constructed by two natural language tokens (“local” and “file”). Therefore, in this research question, we aim to explore whether we can refine the wording and make it consistent with the original logging text written by developers based on the extracted identifiers in the source code.

Approach.

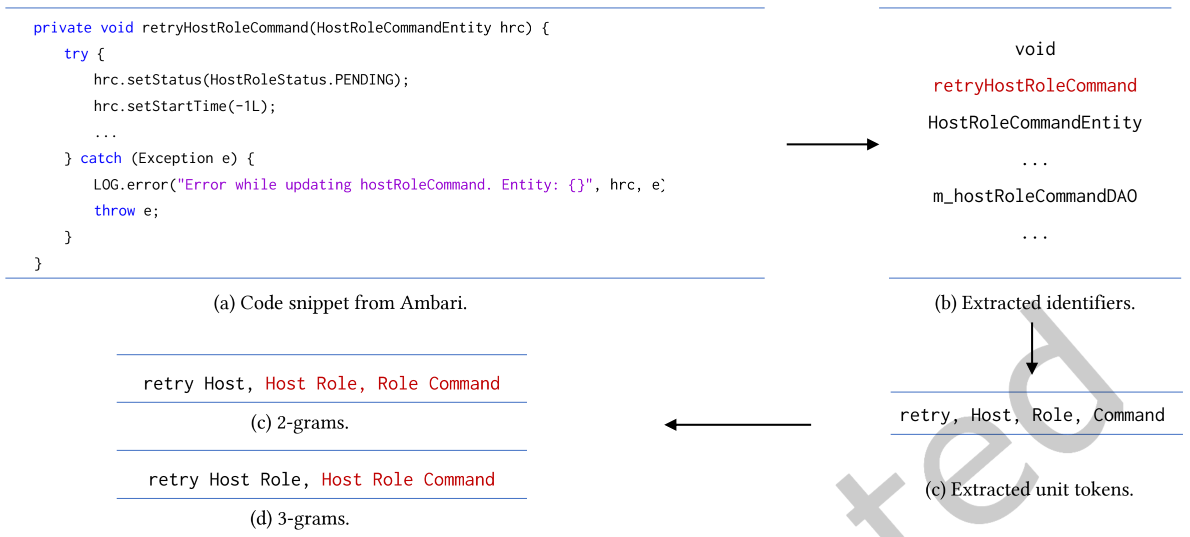
In this RQ, we first build *n*-gram dictionaries from the source code and then extract the token sequences of a certain length (i.e., *n*-grams) from the generated logging text. Then, we check whether we can compose the extracted sequence into bigger units based on the dictionary. We here choose the *n*-gram model, as it has been proven to be useful for processing log data [13].

Building *n*-gram dictionaries from the source code. *Pre-processing source code.* In this step, we try to extract identifiers from the source code. We first apply srcML[9] to the method that contains the logging text. srcML converts the source code into an XML tree, where the leaf nodes are the tokens in the source code. We then use Beautiful Soup to select the nodes with a “name” tag, which is used to mark the identifiers of the source code. For example, Figure 10a shows a code snippet from the project Ambari and the extracted identifiers are shown in Figure 10b. Then, we tokenize the identifiers based on the camel case convention. Figure 10c is the tokenized token units of “retryHostRoleCommand”, which are later used for building the dictionaries.

Building n-gram dictionaries from tokens. In this step, we build *n*-gram dictionaries based on the extracted token units. In our approach, an *n*-gram is a contiguous subsequence of *n* token units from a tokenized identifier. For example, given the sequence of token units “retry, Host, Role, Command”, we can build a dictionary with three 2-grams or a dictionary with two 3-grams, as shown in Figure 10. We build such *n*-gram dictionaries for each method that contains the logging text.

Composing *n*-grams in the generated logging text together into bigger units. *Identifying n-grams in generated logging text.* Similar to the last step, we extract *n*-grams for each generated logging text. For each *n*-gram from the logging text, we check whether it appears in the pre-constructed dictionary in the last step. If the *n*-gram is found in the dictionary, we consider it as an *n*-gram that may be combined into one new token. We filter out the *n*-grams that never appear in the dictionaries. Figure 11a is the generated logging text based on the given source code in Figure 10a. Based on the dictionary built from the source code (i.e., Figure 11c and Figure 10d), we filter out all the 3-grams except the “host role command”.

[9]https://www.srcml.org/



```
private void retryHostRoleCommand(HostRoleCommandEntity hrc) {
    try {
        hrc.setStatus(HostRoleStatus.PENDING);
        hrc.setStartTime(-1L);
        ...
    } catch (Exception e) {
        LOG.error("Error while updating hostRoleCommand. Entity: {}", hrc, e)
        throw e;
    }
}
```

(a) Code snippet from Ambari.

```
void
retryHostRoleCommand
HostRoleCommandEntity
...
m_hostRoleCommandDAO
...
```

(b) Extracted identifiers.

```
retry, Host, Role, Command
```

(c) Extracted unit tokens.

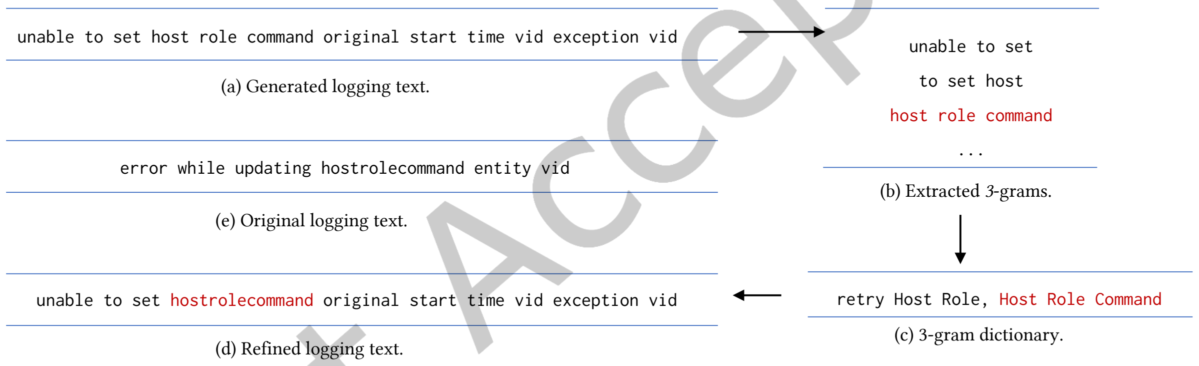
```
retry Host, Host Role, Role Command
```

(c) 2-grams.

```
retry Host Role, Host Role Command
```

(d) 3-grams.

Fig. 10. An overview of the process of building *n*-gram dictionaries from the source code.



```
unable to set host role command original start time vid exception vid
```

(a) Generated logging text.

```
unable to set
to set host
host role command
...
```

(b) Extracted 3-grams.

```
retry Host Role, Host Role Command
```

(c) 3-gram dictionary.

```
unable to set hostrolecommand original start time vid exception vid
```

(d) Refined logging text.

```
error while updating hostrolecommand entity vid
```

(e) Original logging text.

Fig. 11. An overview of the process of composing *n*-grams in the generated logging text together into bigger units.

Composing identified possible n-grams. From the last step, we have obtained a list of *n*-grams that may be combined into new tokens. However, not all the *n*-grams are meaningful or frequently used in the logging text. If we simply combine all these identified *n*-grams, we may have a lot of False Positives, especially when *n* is small (e.g., 2). For example, in Figure 11, if we use a 2-gram dictionary (i.e., Figure 10c), we may combine "host role" or "role command", which results in worse wording in the generated logging text. To avoid such improper combinations, we define two rules: 1) syntactic analysis, and 2) logging practice. The details are shown below:

- Syntactic analysis-based rule (rule 1): The *n*-gram to be combined should be the only siblings of the same parent (e.g., "NP", "VP") in the consistency-based parse tree. Figure 12 illustrates the constraint that "host, role, command" are siblings (i.e., three nouns) with the same parent node (i.e, "NP"), and their parent only has these three children. Thus, they can be used together to compose a bigger token unit.
- Logging practice-based rule (rule 2): Meanwhile, some developers prefer to use separate tokens of identifiers in the logging text. Therefore, to make the combined new tokens conform to the existing logging text conventions, we propose our second rule: for a 2-gram, the newly composed bigger token unit in the

existing logging texts (i.e. the training corpus) should be more frequently used than that of the 2-gram (i.e., the number of occurrences of the newly composed bigger token unit should be larger than that of a 2-gram); for 3-grams, the newly combined token should appear at least once in the existing logging text.

Note that in our experiment, to balance the recall and precision, we further adjust the scope of the rules, that is we limit the identifiers for building dictionaries to the method name.

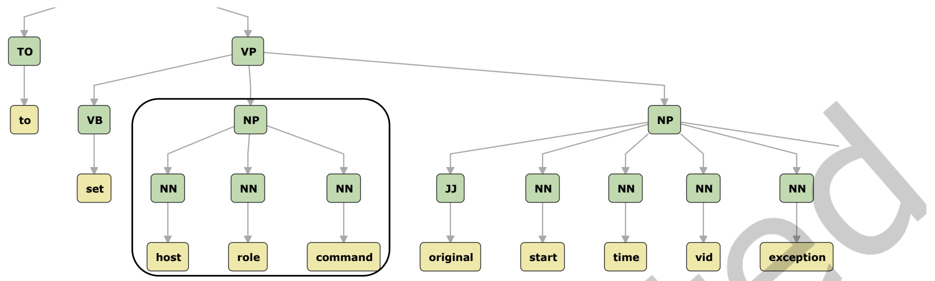


Fig. 12. The constituency-based parse tree of the logging text in Figure 11a. The tokens delimited by the black lines are three siblings and the only children of the node "NP".

To examine the effect of our dictionary-based combination strategy, we first compare the generated logging texts and the logging texts written by developers, and then we manually study the results.

Table 9. Evaluation results of applying different constraints for composing *n*-grams in the generated logging text together into bigger units (RQ6). **Ground truth** represents the number of the generated logging texts containing *n*-grams that should be combined. **Detected** is the number of the detected logging texts by our strategy. **Relevant** represents the number of the detected ground truth. **Both, None, Rule 1, and Rule 2** represent when applying both constraints, no constraint, the first constraint and the second constraint respectively.

Rules		**ActiveMQ**	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**
	Ground truth	1	4	0	4	6	6	4	2	2	2
Both	Detected	1	1	0	0	0	3	2	1	0	1
	Relevant	1	1	0	0	0	0	0	0	0	0
None	Detected	12	24	21	48	165	82	30	30	25	8
	Relevant	1	2	0	1	4	4	2	1	1	0
Rule 1	Detected	5	13	9	16	96	42	20	16	7	4
	Relevant	1	1	0	0	0	2	1	1	1	0
Rule 2	Detected	1	1	0	0	3	5	2	1	0	1
	Relevant	1	1	0	0	0	1	0	0	0	0

Results.

Although we can refine the generated logging text with the post-processing strategy, the improvement is limited. We first manually check the generated logging texts and compare them with the original logging texts extracted from the source code. For a generated logging text, if it contains a combined *n*-gram that can be found in the original logging texts, then it is considered as a ground truth. We find there do exist *n*-grams in the generated logging text that can be possibly combined, but the number is relatively small (i.e., Ground truth in Table 9). As shown in Table 9, the number of ground truth is small, which means that there are a few generated logging texts containing a *n*-gram that should be combined to harmonize the wording (e.g., the generated logging text in Figure 11a).

To examine the impact of our proposed two rules, we have conducted another three ablation experiments. First, we remove the two rules and combine the *n*-grams only based on the dictionary (i.e., None in Table 9). As Table 9 shows, a much larger size of the generated logging texts is detected, while a small number of the detected logging texts are relevant to the ground truth (i.e., True Positives). To filter out the irrelevant logging texts (i.e., False Positives), we apply the two rules to the detected logging texts. As a result, we successfully detect two logging texts that can be further refined. The results show that by using our proposed post-processing strategy we can further improve the generated logging texts, but the improvement is incremental. Future research may consider modifying the architecture of the Transformer-based model and incorporating the *n*-gram dictionary into the model (e.g., *N*-Grammer [68]) during the training or inference stage to improve the quality of the generated logging texts.

Finally, we manually study the detection results and identify two possible reasons for incorrectly detected logging texts: 1) Inconsistent writing convention. For example, although this research question is motivated by the logging text in Listing 5.1, we still fail to refine this generated logging text by combining the 2-gram "local file". We check the method and find both "localfile" and "local file" are used in the logging statements of this method, but "localfile" only appears once in this extracted logging text, thus the combination operation is ignored. 2) Missed *n*-grams in the dictionary. This is reasonable, as we limit the identifier to the method name for building the dictionaries, we have the chance to miss some *n*-grams. However, selecting the identifiers is a trade-off. If we chose more identifiers, as a result, we would produce more false positives. On the contrary, fewer identifiers would cause the miss of the possible combinations.

Summary

It is possible to harmonize the wording in the generated logging text with the post-processing strategy that leverages the token sequence (i.e., *n*-grams) in the source code, however, the improvement is limited. Future research may consider incorporating the obtained *n*-grams from the source code into the logging text generation model during the training or inference stage, to improve the quality of the generated logging texts.

5 HUMAN EVALUATION

Our approach *LoGenText* and its extension *LoGenText-Plus* are 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 two separate human evaluations to further evaluate *LoGenText* and *LoGenText-Plus*.

5.1 Evaluation of *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 for *LoGenText* 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 [87], 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 4 show that the context-aware form of *LoGenText* incorporating the AST context has the best overall performance, we used it to generate logging texts for 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 13 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 test [84] to

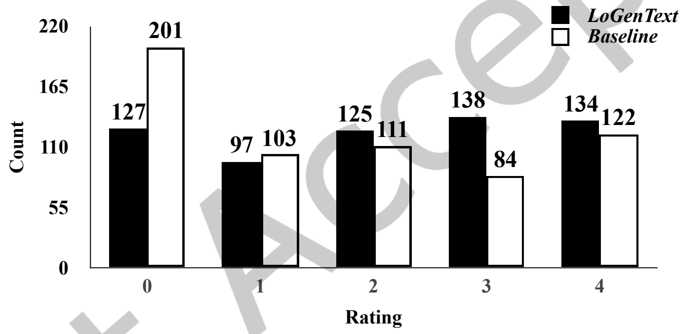


Fig. 13. Distribution of the rating results (in task 1) in terms of the similarity between the generated logging texts and the reference logging texts.

statistically 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 10. 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 4), 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 scores of the logging texts in each group. As shown in Table 10, 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 the 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 high (i.e., 3 or 4) while the BLEU scores are lower than the 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 Ambari,

Listing 1. An example generated logging text from Ambari.

```
// Original logging statement:
LOG.info("copying localfile := " + sourceFilepath + " to hdfsPath := " + destFilePath)
```

$\downarrow$

```
// Extracted logging text after preprocessing:
"copying localfile <vid> to hdfspath <vid>"
```

$\xrightarrow{\textit{compares}}$

```
// 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 a 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.

Listing 2. An example generated logging text from Hadoop.

```
// Original logging statement:
log.debug("No beanstalks defined for initialization.")
```

$\downarrow$

```
// Extracted logging text after preprocessing:
"no beanstalks defined for initialization"
```

$\xrightarrow{\textit{compares}}$

```
// Generated logging text:
"no beanstalk definitions found for
     initialization"
```

LoGenText **outperforms the baseline approach in all three aspects.** Table 11 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, 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

Table 11. 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.

developers can still be further improved and call for high-quality logging texts to record the software execution information.

Summary

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.

5.2 Evaluation of *LoGenText-Plus*

In the last section, we have conducted a human evaluation to compare *LoGenText* and the baseline and the results show that *LoGenText* outperforms the baseline approach in all aspects. Therefore, in this section, we conduct another human evaluation to compare baseline and *LoGenText* with *LoGenText-Plus*. Considering the number of cases to evaluate, we invited 10 out of the 42 participants and nine of them are from academia and one from industry.

Our human evaluation for *LoGenText-Plus* contains two tasks: **task 1)** comparing the quality of the logging texts generated by *LoGenText* with that of *LoGenText-Plus* based on the given source code, and **task 2)** comparing the quality of the logging texts generated by the baseline approach with that of *LoGenText-Plus*. To be consistent with the evaluation for *LoGenText*, we use the same dataset as that of task 2 in Section 5.1. In order to avoid redundancy, we filter the dataset to remove cases where *LoGenText-Plus* and *LoGenText* as well as the baseline approach, generate the identical logging texts. As a result, approximately half of the samples are filtered out, leaving us with 69 out of 126 samples for task 1 and 96 out of 126 samples for task 2. Similar to the methodology used in task 2 in Section 5.1, each participant was presented with the two generated logging texts as *log-1* and *log-2* and the surrounding method of the logging statement that highlights the location of the logging statement. Note that the names *log-1* and *log-2* were randomly assigned to avoid bias. Then each participant was asked to examine whether *log-1* is better than *log-2* based on the given code snippet. We listed three options for each comparison, *TRUE*, *FALSE*, and *NA*. *TRUE* means that *log-1* is better than *log-2*, *FALSE* means that *log-2* is better, and *NA* means the two logging texts are hard to compare (e.g., both are similar or useless). Besides, each participant was asked to provide reasons why they made the decision.

Overall, *LoGenText-Plus* **generates better logging texts compared to that generated by** *LoGenText* **and the baseline approach.** Figure 14 presents the distribution of the user ratings in our evaluation, where "Neutral" means that the two generated logging texts are hard to compare. We find that *LoGenText-Plus* generates more logging texts that are better than *LoGenText* (i.e., 53.6% vs. 34.8%) and the baseline approach (i.e., 57.3% vs. 28.1%), which shows the improvement of *LoGenText-Plus* over *LoGenText* and the baseline. However, we still observe that around 30% of the automatically generated logging texts by *LoGenText* and the baseline approach receive a better rating. The results suggest opportunities for future research to further improve *LoGenText-Plus*.

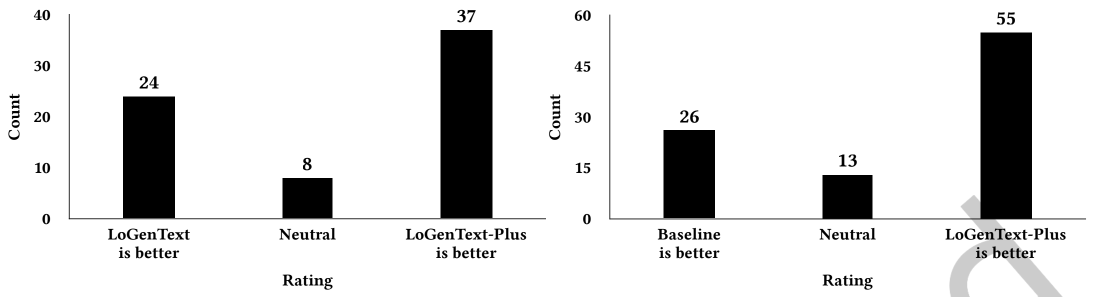


Fig. 14. Comparing the human ratings of the logging texts generated by *LoGenText-Plus*, *LoGenText* and the baseline approach. "Neutral" means that the two generated logging texts are hard to compare.

Besides, to uncover the reasons why *LoGenText-Plus* or *LoGenText* receives a higher rating, we further manually examine the comments provided by participants as well as the generated logging texts. In other words, the goal of this step is to find out what kind of logging texts (i.e., the characteristics) are considered with a higher quality by developers, aiming to provide guidance for writing or generating good logging texts.

We summarize five main reasons that a developer may favor a logging text. The reasons together with examples are presented in Table 12. We discuss each reason in detail in the rest of this section.

More relevant to the source code. The logging text should be more relevant to the source code, which means that the actions that happened in the source code should be similar or the same as the described actions in the logging text. For example, as shown in Table 12a, the method is about the action "shut down", while the generated log-2 is describing the action of "connection", which is irrelevant to the source code, thus, log-1 is selected as a better logging text.

More descriptive and useful information to the source code. We find that some logging texts are very short and simple, and thus cannot detailly describe what is happening in the software system. For example, as shown in Table 12b, the generated log-1 provides more information, as it not only tells the action taken (i.e., "load") before the logging statement but also shows the result of this action (i.e., an exception occurs).

More succinct and less confusing/redundant words. On one hand, the logging text should provide enough information for failure diagnosis. On the other hand, the logging text should also be more succinct and avoid confusing or redundant descriptions. For example, in the generated logging text log-2 in Table 12c, the last few words are a little meaningless and may confuse developers while analyzing the logs. Besides, our approach can insert "<vid>"s to the generated logging text as variable placeholders. However, one participant commented that some "<vid>"s may be unnecessary.

More accurate. Another important factor for a better logging text is to use more accurate descriptions. The logging text should avoid providing the wrong information, which may mislead the developers. For example, as shown in Table 12d, the generated log-1 uses "an old" to describes the object "session". However, based on the source code in the "try" block, the "session" should be a "new" session, instead of the "old".

More specific and focusing on critical actions in source code. We notice that there may exist several statements inserted before the target logging statement in the source code. For such cases, the logging text should focus on more critical statements and describe the specific statements with less general words. For example, as shown in Table 12e, there are a list of database-related actions, including "doDropTables" and "doCreateTables", which is exactly what the generated log-2 describes, "executing sql <vid>". However, based on the feedback from

Table 12. The summary of the five main reasons that a developer may favor a logging text. The "Generated log-1/2" represents two logging texts by either *LoGenText-Plus* or *LoGenText*. "Original log" denotes the logging statement written by developers. The better logging text is highlighted in bold green.

Reason	Example
More relevant to the source code	`public void shutdown() {` `---------------Candidate log start----------------` **`Generated log-1: shutting down connection to zk <vid>`** `Generated log-2: connection to zookeeper <vid>` `Original log: LOG.debug("CamelDestination shutdown()")` `---------------Candidate log end----------------` `...` **(a) CamelDestination.java from Camel**
More descriptive and useful information to the source code	`...` `load(key.file(), props);` `} catch (IOException e) {` `---------------Candidate log start----------------` **`Generated log-1: load of <vid> failure exception <vid>`** `Generated log-2: load of <vid>` `Original log: LOG.error("Failed to load: " + key + ", reason:" + e.getLocalizedMessage())` `---------------Candidate log end----------------` `...` **(b) ReloadableProperties.java from ActiveMQ**
More succinct and less confusing/redundant words	`...` `if(filterPart.size() != 2) {` `---------------Candidate log start----------------` **`Generated log-1: invalid filter specification <vid> skipping`** `Generated log-2: invalid filter specification filters count <vid> skipping split s` `Original log: LOG.warn("Invalid filter specification " + filterClass + " - skipping")` `---------------Candidate log end----------------` `} else {` `...` **(c) ThriftServer.java from HBase**
More accurate	`...` `try {` `    newSession.close(false);` `} catch (Exception ex) {` `---------------Candidate log start----------------` `Generated log-1: failed to close an old session ignoring` **`Generated log-2: failed to close session <vid>`** `Original log: LOG.error("Failed to close an unneeded session", ex)` `---------------Candidate log end----------------` `...` **(d) TezSessionPool.java from Hive**
More specific and focusing on critical actions in source code	`public void deleteAllMessages() throws IOException {` `...` `    getAdapter().doDropTables(c);` `    getAdapter().setUseExternalMessageReferences(isUseExternalMessageReferences());` `    getAdapter().doCreateTables(c);` `---------------Candidate log start----------------` **`Generated log-1: deleted apache activemq <vid>`** `Generated log-2: executing sql <vid>` `Original log: LOG.info("Persistence store purged.")` `---------------Candidate log end----------------` `...` **(e) JDBCPersistenceAdapter.java from ActiveMQ**

the participants as well as the original logging statement, the logging text "executing sql" is too general, while "deleted" is more specific and describes the more critical action "doDropTables".

Besides, as shown in Table 14, there are also some cases where log-1 and log-2 are hard to compare. This may be caused by two reasons: 1) **The two generated logging texts are inaccurate or even wrong.** For example, in Listing 3, the actual action in the source code is "scanning" file, instead of the generated "restoring" or "deleting". 2) **The two generated logging texts have a very similar meaning.** For example, in Listing 4, the two generated logging texts are almost the same, except for the missing preposition "to" in generated log-2, of which the influence can be ignored. Therefore, they are considered to convey the same information.

Listing 3. An example generated logging text from Synapse.

```
private void scanFileOrDirectory(final ...) {
    FileObject fileObject = null;
    if (log.isDebugEnabled()) {
---------------Candidate log start----------------
Generated log-1: restoring the file <vid> at <vid>
Generated log-2: deleting temporary file <vid>
Original log: log.debug("Scanning directory or file :
     " + VFSUtils.maskURLPassword(fileURI))
---------------Candidate log end----------------
...
```

Listing 4. An example generated logging text from Brooklyn.

```
...
---------------Candidate log start----------------
Generated log-1: failed to transfer <vid> to <vid>
     retryable error attempt <vid> vid <vid>
Generated log-2: failed transfer <vid> to <vid>
     retryable error attempt <vid> vid <vid>
Original log: log.warn("Failed to transfer " +
     urlToInstall + " to " + machine + ", not a
     retryable error so failing: " + e)
---------------Candidate log end----------------
```

Summary

Overall, the logging texts generated by *LoGenText-Plus* have a higher quality than that generated by *LoGenText*. Besides, we identify five possible reasons that a developer may favor a logging text. The reasons can be used as a guideline for practitioners to improve the process of automated logging generation.

6 THREATS TO VALIDITY

Internal Validity. In this work, we compare our approach with prior work by He et al. [29]. Meanwhile, pre-trained models of code have achieved new state-of-the-art results for several code-related tasks, such as clone detection, code search, and code completion [25]. Therefore, we also try to select UniXcoder [25] as a comparison, which is most recently released and has shown to have better performance than CodeBERT [19], CodeT5 [82], and GraphCodeBERT [26]. We conduct the experiments under two settings: 1) zero-shot logging statement completion, and 2) fine-tuning the model on our training dataset. Similar to our experiments, we use the pre-log code as input. Under the zero-shot setting, there is only an average of 26.6% logging statements generated among all the test inputs, with an average BLEU score of 12.9. Besides, we also fine-tune UniXcoder on our training dataset of each project, and there is an average of 34.7% logging statements generated among all the test inputs, with an average BLEU score of 22.2, which is slightly worse than that of the baseline approach and our proposed approaches. The reasons may come from 1) the lack of training data on logging statements and 2) the lack of optimized training objectives for logging statement-related tasks. Future work may consider designing logging statement-specific pre-training objectives and pre-training the model using the dataset curated for logging. Meanwhile, Mastropaolo et al. [59] propose to train a T5 model to support the automatic generation of complete log statements, including the generation of logging texts, where to log, and which level to log. However, as mentioned in the work of Mastropaolo et al. [59], the generated logging texts have a BLEU score of 15, which is also lower than the average result (i.e., 30.3) reported in our paper. However, we believe that the performance of the T5 model can be further improved by, for example, 1) training on a larger corpus (currently, it is only trained with 6M Java methods) and 2) including the AST or other types of information extracted from source code, which has shown to be useful for code-related tasks. In RQ2, we attempt to include two types of context information to further improve *LoGenText*. Similarly, we design two strategies in RQ3 to incorporate logging texts from similar code snippets. 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* and *LoGenText-Plus* 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. However, to mitigate this effect, we try our best to invite more than 40 participants. The number of participants is much larger than that of previous research [80]. Future work should consider further evaluating *LoGenText* and *LoGenText-Plus* by using them in a real-life industrial setting.

External Validity. In this paper, we evaluate *LoGenText* and *LoGenText-Plus* based on 10 subject systems. All of the subject systems are open-source systems that are mainly written in Java. In addition, all of the subject systems are server or desktop applications, while logging practices on mobile devices are found to be different [97]. Evaluating *LoGenText* and *LoGenText-Plus* 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.

Construct Validity. Our data (e.g., logging texts, pre-log code, post-log and ASTs) are extracted based on the *srcML* tool [12]. *srcML* is a mature tool and has been widely used in various software engineering research. Nevertheless, the quality of the data generated by *srcML* may impact the results of our study. *LoGenText* and *LoGenText-Plus* require 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 [38, 77] to configure the hyper-parameters. Performing further fine-tuning on these hyper-parameters may even further improve the results from *LoGenText* and *LoGenText-Plus*. In our evaluation, the data from each project is randomly split into 80%/10%/10% training, validation and testing datasets. The evaluation results may show some differences with other splits of the training, validation, and testing datasets. Besides, we find that there exists a duplication of the data samples between the training and testing datasets, which may also impact the evaluation results. Specifically, there are 1 to 29 or 0.5% to 6.6% duplicate logging statements in the studied subjects. However, we did not remove these duplicates as the number of duplicate instances is relatively small, and we want to evaluate our approach in a real-life situation where duplicate logging statements do exist [45, 46]. Future work may consider exploring how duplicate logging statements would impact the tool.

7 RELATED WORK

Automated logging suggestions. Although logs are of much value to software practitioners, the usefulness of logs highly depends on their quality. Both logging too much and logging too little are undesired in practice [40, 96]. There exists a significant challenge for developers to make proper logging decisions. In general, prior research has proposed two main types of approaches to address the challenge including 1) proactively providing suggestions to developers, and 2) retroactively detecting issues in logging statements.

Prior research has proposed automated approaches that provide different logging suggestions including the locations of logging statements [22, 39, 91, 100, 104], the verbosity levels [41, 47], the variables to include in a logging statement [51], and the need to update an existing logging statement [42]. The most related work to our paper is from He et al. [29], 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 3). Recently, pre-trained models of code have achieved new state-of-the-art results for several code-related tasks, such as clone detection, code search, and code completion [25]. Inspired by these advances, Mastropaolo et al. [59] propose to train a Text-To-Text-Transfer-Transformer (T5) model to support the automatic generation of the complete logging statement, including the log positions, log levels, and the logging text (the focus of our work). However, although the model is trained on more than 1,000 projects, the generated logging texts only have a

BLEU score of 15, which is lower than the average result (i.e., 30.3) reported in our paper. Other research aims to detect issues in logging statements. Chen et al. [8] and Hassani et al. [28] 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 [40, 95]. 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. [96]. Afterwards, follow-up studies by Chen et al. [9] and Zeng et al. [97] extend Yuan et al's study from C/C++ projects to Java projects and Android app projects, respectively. Similarly, Shang et al. [70] conduct a study focusing on the evolution of logging statements. Recently, Li et al. [40] 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 [35], logging utilities [10] and libraries[34], logging configurations [103], and the relationship between logging practices and software quality [72] and performance [11, 97].

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

8 CONCLUSION

In this paper, we present our approach, *LoGenText*, and its improved version, *LoGenText-Plus*, 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 both *LoGenText* and *LoGenText-Plus* show promising results in the automated generation of logging texts. Our approach *LoGenText-Plus*, which leverages the logging template information, outperforms the state-of-the-art *LoGenText* and the 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 developers' efforts in logging development and maintenance and potentially improve the overall quality of software logging.

REFERENCES

[1] Ruchit Agrawal, Marco Turchi, and Matteo Negri. [n.d.]. *Contextual Handling in Neural Machine Translation: Look Behind, Ahead and on Both Sides.* European Association for Machine Translation. http://rua.ua.es/dspace/handle/10045/76016 EAMT.

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. *arXiv preprint arXiv:2005.00653* (2020).

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[4] T. Barik, R. DeLine, S. Drucker, and D. Fisher. 2016. 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 (ICSE Companion '16).* 92–101.

[5] Rachel Bawden, Rico Sennrich, Alexandra Birch, and Barry Haddow. 2018. 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, 1304–1313. https://doi.org/10.18653/v1/n18-1118

[6] Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. 2013. 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.* 335–340. http://www.ppgia.pucpr.br/ismir2013/wp-content/uploads/2013/09/243_Paper.pdf

[7] Lutz Büch and Artur Andrzejak. 2019. 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, 95–104. https://doi.org/10.1109/SANER.2019.8668039

[8] Boyuan Chen and Zhen Ming (Jack) Jiang. 2017. Characterizing and Detecting Anti-Patterns in the Logging Code. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17).* IEEE Press, 71–81. https://doi.org/10.1109/ICSE.2017.15

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

[10] Boyuan Chen and Zhen Ming (Jack) Jiang. 2020. Studying the Use of Java Logging Utilities in the Wild. In *Proceedings of the 42th International Conference on Software Engineering (ICSE '20).*

[11] S. Chowdhury, S. D. Nardo, A. Hindle, and Z. M. Jiang. 2017. An exploratory study on assessing the energy impact of logging on Android applications. *Empirical Software Engineering* 23 (2017), 1422–1456.

[12] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. 2011. 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, 173–184. https://doi.org/10.1109/SCAM.2011.19

[13] Hetong Dai, Heng Li, Che-Shao Chen, Weiyi Shang, and Tse-Hsun Chen. 2022. Logram: Efficient Log Parsing Using nn-Gram Dictionaries. *IEEE Trans. Software Eng.* 48, 3 (2022), 879–892. https://doi.org/10.1109/TSE.2020.3007554

[14] Zishuo Ding, Heng Li, and Weiyi Shang. 2022. LoGenText: Automatically Generating Logging Texts Using Neural Machine Translation. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022.* IEEE, 349–360. https://doi.org/10.1109/SANER53432.2022.00051

[15] Zishuo Ding, Heng Li, Weiyi Shang, and Tse-Hsun Peter Chen. 2022. Can pre-trained code embeddings improve model performance? Revisiting the use of code embeddings in software engineering tasks. *Empirical Software Engineering* 27, 3 (2022), 63.

[16] Zishuo Ding, Heng Li, Weiyi Shang, and Tse-Hsun (Peter) Chen. 2022. Towards Learning Generalizable Code Embeddings Using Task-Agnostic Graph Convolutional Networks. *ACM Trans. Softw. Eng. Methodol.* (jun 2022). https://doi.org/10.1145/3542944 Just Accepted.

[17] Zishuo Ding, Yiming Tang, Yang Li, Heng Li, and Weiyi Shang. 2023. On the Temporal Relations between Logging and Code. In *Proceedings of the 45rd International Conference on Software Engineering (ICSE) (ICSE '23).* IEEE.

[18] Li Dong and Mirella Lapata. 2018. Coarse-to-Fine Decoding for Neural Semantic Parsing. 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,* Iryna Gurevych and Yusuke Miyao (Eds.). Association for Computational Linguistics, 731–742. https://doi.org/10.18653/v1/P18-1068

[19] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL),* Trevor Cohn, Yulan He, and Yang Liu (Eds.), Vol. EMNLP 2020. Association for Computational Linguistics, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[20] Qiang Fu, Jian-Guang Lou, Qingwei Lin, Rui Ding, Dongmei Zhang, and Tao Xie. 2013. Contextual Analysis of Program Logs for Understanding System Behaviors. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13).* 397–400.

[21] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution Anomaly Detection in Distributed Systems Through Unstructured Log Analysis. In *Proceedings of the 9th IEEE International Conference on Data Mining (ICDM '09).* 149–158.

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

[23] Philip Gage. 1994. A new algorithm for data compression. *C Users Journal* 12, 2 (1994), 23–38.

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[25] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499

[26] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=jLoC4ez43PZ

[27] Kelvin Guu, Tatsunori B. Hashimoto, Yonatan Oren, and Percy Liang. 2018. Generating Sentences by Editing Prototypes. *Trans. Assoc. Comput. Linguistics* 6 (2018), 437–450. https://doi.org/10.1162/tacl_a_00030

[28] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. 2018. Studying and detecting log-related issues. *Empirical Software Engineering* 23, 6 (2018), 3248–3280.

[29] Pinjia He, Zhuangbin Chen, Shilin He, and Michael R. Lyu. 2018. 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, 178–189. https://doi.org/10.1145/3238147.3238193

[30] Vincent J. Hellendoorn and Premkumar T. Devanbu. 2017. 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, 763–773. https://doi.org/10.1145/3106237.3106290

[31] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *ICSE*. IEEE Computer Society, 837–847.

[32] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*. ACM, 200–210. https://doi.org/10.1145/3196321.3196334

[33] Zhen Ming Jiang, Ahmed E Hassan, Gilbert Hamann, and Parminder Flora. 2008. Automatic identification of load testing problems. In *Proceedings of the 2008 IEEE International Conference on Software Maintenance (ICSM '08)*. 307–316.

[34] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. 2016. 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*, Miryung Kim, Romain Robbes, and Christian Bird (Eds.). ACM, 154–164. https://doi.org/10.1145/2901739.2901769

[35] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, Mark D. Syer, and Ahmed E. Hassan. 2018. Examining the stability of logging statements. *Empirical Software Engineering* 23, 1 (01 Feb. 2018), 290–333. https://doi.org/10.1007/s10664-017-9518-0

[36] Yunsu Kim, Duc Thanh Tran, and Hermann Ney. 2019. 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, 24–34. https://doi.org/10.18653/v1/D19-6503

[37] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.

[38] Bei Li, Hui Liu, Ziyang Wang, Yufan Jiang, Tong Xiao, Jingbo Zhu, Tongran Liu, and Changliang Li. 2020. 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, 3512–3518. https://www.aclweb.org/anthology/2020.acl-main.322/

[39] Heng Li, Tse-Hsun (Peter) Chen, Weiyi Shang, and Ahmed E. Hassan. 2018. Studying software logging using topic models. *Empirical Software Engineering* 23, 5 (01 Oct. 2018), 2655–2694. https://doi.org/10.1007/s10664-018-9595-8

[40] H. Li, W. Shang, B. Adams, M. Sayagh, and A. E. Hassan. 2020. A Qualitative Study of the Benefits and Costs of Logging from Developers' Perspectives. *IEEE Transactions on Software Engineering* (2020), 1–1.

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

[42] Heng Li, Weiyi Shang, Ying Zou, and Ahmed E Hassan. 2017. Towards just-in-time suggestions for log changes. *Empirical Software Engineering* 22, 4 (2017), 1831–1865.

[43] Yangguang Li, Zhen Ming (Jack) Jiang, Heng Li, Ahmed E. Hassan, Cheng He, Ruirui Huang, Zhengda Zeng, Mian Wang, and Pinan Chen. 2020. Predicting Node Failures in an Ultra-Large-Scale Cloud Computing Platform: An AIOps Solution. *ACM Transactions on Software Engineering and Methodology* 29, 2 (2020), 13:1–13:24. https://doi.org/10.1145/3385187

[44] Zhenhao Li, Tse-Hsun Chen, and Weiyi Shang. 2020. Where Shall We Log? Studying and Suggesting Logging Locations in Code Blocks. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*. 361–372.

[45] Zhenhao Li, Tse-Hsun (Peter) Chen, Jinqiu Yang, and Weiyi Shang. 2019. Dlfinder: characterizing and detecting duplicate logging code smells. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 152–163. https://doi.org/10.1109/ICSE.2019.00032

[46] Zhenhao Li, Tse-Hsun (Peter) Chen, Jinqiu Yang, and Weiyi Shang. 2022. Studying Duplicate Logging Statements and Their Relationships with Code Clones. *IEEE Transactions on Software Engineering* (2022), 1–19.

[47] Zhenhao Li, Heng Li, Tse-Hsun Chen, and Weiyi Shang. 2021. DeepLV: Suggesting Log Levels Using Ordinal Based Neural Networks. In *Proceedings of the 43rd International Conference on Software Engineering, ICSE 2021*. 1–12.

[48] Zhenhao Li, Chuan Luo, Tse-Hsun Chen, Weiyi Shang, Shilin He, Qingwei Lin, and Dongmei Zhang. 2023. Did We Miss Something Important? Studying and Exploring Variable-Aware Log Abstraction. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 830–842. https://doi.org/10.1109/ICSE48619.2023.00078

[49] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[50] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Ming Li, Zhiyi Fu, and Zhi Jin. 2020. Characterizing logging practices in open-source software. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC '20)*.

[51] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li. 2019. Which Variables Should I Log? *IEEE Transactions on Software Engineering* (2019), 1–1.

[52] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. 2018. CloudRaid: hunting concurrency bugs in the cloud via log-mining. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 3–14. https://doi.org/10.1145/3236024.3236071

[53] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. 2019. CrashTuner: detecting crash-recovery bugs in cloud systems via meta-info analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 114–130. https://doi.org/10.1145/3341301.3359645

[54] Thang Luong, Ilya Sutskever, Quoc Le, Oriol Vinyals, and Wojciech Zaremba. 2015. 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)*. Association for Computational Linguistics, Beijing, China, 11–19. https://doi.org/10.3115/v1/P15-1002

[55] L. Mariani and F. Pastore. 2008. Automated Identification of Failure Causes in System Logs. In *Proceedings of the 19th International Symposium on Software Reliability Engineering (ISSRE '08)*. 117–126.

[56] Sameen Maruf and Gholamreza Haffari. 2018. 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, 1275–1284. https://doi.org/10.18653/v1/P18-1118

[57] Sameen Maruf, André F. T. Martins, and Gholamreza Haffari. 2019. 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, 3092–3102. https://doi.org/10.18653/v1/n19-1313

[58] Antonio Mastropaolo, Nathan Cooper, David Nader-Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2022. Using Transfer Learning for Code-Related Tasks. *CoRR* abs/2206.08574 (2022). https://doi.org/10.48550/arXiv.2206.08574 arXiv:2206.08574

[59] Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. 2022. Using Deep Learning to Generate Complete Log Statements. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2279–2290. https://doi.org/10.1145/3510003.3511561

[60] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader-Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 336–347. https://doi.org/10.1109/ICSE43902.2021.00041

[61] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. 26–26.

[62] Masato Neishi, Jin Sakuma, Satoshi Tohda, Shonosuke Ishiwatari, Naoki Yoshinaga, and Masashi Toyoda. 2017. 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, 99–109. https://www.aclweb.org/anthology/W17-5708/

[63] Emilio Soria Olivas, Jose David Martin Guerrero, Marcelino Martinez Sober, Jose Rafael Magdalena Benedito, and Antonio Jose Serrano Lopez. 2009. *Handbook Of Research On Machine Learning Applications and Trends: Algorithms, Methods and Techniques - 2 Volumes.*

Information Science Reference - Imprint of: IGI Publishing, Hershey, PA.

[64] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. 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, 48–53. https://doi.org/10.18653/v1/n19-4009

[65] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. 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, 311–318. https://doi.org/10.3115/1073083.1073135

[66] Matt Post. 2018. 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, 186–191. https://doi.org/10.18653/v1/w18-6319

[67] Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D. Manning. 2020. Stanza: A Python Natural Language Processing Toolkit for Many Human Languages. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations, ACL 2020, Online, July 5-10, 2020*, Asli Celikyilmaz and Tsung-Hsien Wen (Eds.). Association for Computational Linguistics, 101–108. https://doi.org/10.18653/v1/2020.acl-demos.14

[68] Aurko Roy, Rohan Anil, Guangda Lai, Benjamin Lee, Jeffrey Zhao, Shuyuan Zhang, Shibo Wang, Ye Zhang, Shen Wu, Rigel Swavely, Tao Yu, Phuong Dao, Christopher Fifty, Zhifeng Chen, and Yonghui Wu. 2022. N-Grammer: Augmenting Transformers with latent n-grams. *CoRR* abs/2207.06366 (2022). https://doi.org/10.48550/arXiv.2207.06366 arXiv:2207.06366

[69] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. 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. https://doi.org/10.18653/v1/p16-1162

[70] Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. [n.d.]. An exploratory study of the evolution of communicated information about the execution of large software systems. *Journal of Software: Evolution and Process* 26, 1 ([n. d.]), 3–26. https://doi.org/10.1002/smr.1579

[71] Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Bram Adams, Ahmed E. Hassan, and Patrick Martin. 2013. Assisting Developers of Big Data Analytics Applications when Deploying on Hadoop Clouds. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13).* 402–411.

[72] Weiyi Shang, Meiyappan Nagappan, and Ahmed E. Hassan. 2015. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering* 20, 1 (01 Feb. 2015), 1–27. https://doi.org/10.1007/s10664-013-9274-8

[73] Mark D Syer, Zhen Ming Jiang, Meiyappan Nagappan, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. 2013. Leveraging performance counters and execution logs to diagnose memory-related performance issues. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM '13).* 110–119.

[74] Yiming Tang, Allan Spektor, Raffi Khatchadourian, and Mehdi Bagherzadeh. 2022. Automated evolution of feature logging statement levels using Git histories and degree of interest. *Science of Computer Programming* 214 (2022), 102724. https://doi.org/10.1016/j.scico.2021.102724

[75] Jörg Tiedemann and Yves Scherrer. 2017. 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, 82–92. https://doi.org/10.18653/v1/w17-4811

[76] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep learning similarities from different representations of source code. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018.* ACM, 542–553. https://doi.org/10.1145/3196398.3196431

[77] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. 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.* 5998–6008. http://papers.nips.cc/paper/7181-attention-is-all-you-need

[78] Elena Voita, Rico Sennrich, and Ivan Titov. 2019. 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*, Anna Korhonen, David R. Traum, and Lluís Màrquez (Eds.). Association for Computational Linguistics, 1198–1212. https://doi.org/10.18653/v1/p19-1116

[79] Elena Voita, Pavel Serdyukov, Rico Sennrich, and Ivan Titov. 2018. 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, 1264–1274. https://doi.org/10.18653/v1/P18-1117

[80] Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. 2021. Context-aware Retrieval-based Deep Commit Message Generation. *ACM Trans. Softw. Eng. Methodol.* 30, 4 (2021), 56:1–56:30. https://doi.org/10.1145/3464689

[81] Kai Wang, Xiaojun Quan, and Rui Wang. 2019. BiSET: Bi-directional Selective Encoding with Template for Abstractive Summarization. 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, Anna Korhonen, David R. Traum, and Lluís Màrquez (Eds.). Association for Computational Linguistics, 2153–2162. https://doi.org/10.18653/v1/p19-1207

[82] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[83] Lesly Miculicich Werlen, Dhananjay Ram, Nikolaos Pappas, and James Henderson. 2018. 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, 2947–2954. https://doi.org/10.18653/v1/d18-1325

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

[85] Sam Wiseman, Stuart M. Shieber, and Alexander M. Rush. 2018. Learning Neural Templates for Text Generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.). Association for Computational Linguistics, 3174–3187. https://doi.org/10.18653/v1/d18-1356

[86] Shuangzhi Wu, Dongdong Zhang, Nan Yang, Mu Li, and Ming Zhou. 2017. 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*, Regina Barzilay and Min-Yen Kan (Eds.). Association for Computational Linguistics, 698–707. https://doi.org/10.18653/v1/P17-1065

[87] Bowen Xu, Zhenchang Xing, Xin Xia, and David Lo. 2017. AnswerBot: Automated Generation of Answer Summary to Developersun-defined Technical Questions. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017).* IEEE Press, 706–716.

[88] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. 2009. Online System Problem Detection by Mining Patterns of Console Logs. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining (ICDM '09).* 588–597.

[89] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting Large-scale System Problems by Mining Console Logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09).* ACM, New York, NY, USA, 117–132. https://doi.org/10.1145/1629575.1629587

[90] Jian Yang, Shuming Ma, Dongdong Zhang, Zhoujun Li, and Ming Zhou. 2020. Improving Neural Machine Translation with Soft Template Prediction. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). Association for Computational Linguistics, 5979–5989. https://doi.org/10.18653/v1/2020.acl-main.531

[91] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. 2020. Log4Perf: suggesting and updating logging locations for web-based systems' performance monitoring. *Empir. Softw. Eng.* 25, 1 (2020), 488–531. https://doi.org/10.1007/s10664-019-09748-z

[92] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. 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.* 3320–3328. http://papers.nips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks

[93] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 249–265. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

[94] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10).* 143–154.

[95] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael Mihn-Jong Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging.. In *OSDI*, Vol. 12. 293–306.

[96] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12).* IEEE, 102–112.

[97] Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun (Peter) Chen. 2019. Studying the characteristics of logging practices in mobile apps: a case study on F-Droid. *Empir. Softw. Eng.* 24, 6 (2019), 3394–3434. https://doi.org/10.1007/s10664-019-09687-9

[98] Jiacheng Zhang, Huanbo Luan, Maosong Sun, Feifei Zhai, Jingfang Xu, Min Zhang, and Yang Liu. 2018. 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, 533–542. https://doi.org/10.18653/v1/d18-1049

[99] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 783–794. https://doi.org/10.1109/ICSE.2019.00086

[100] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 565–581.

[101] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 629–644. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zhao

[102] Zaixiang Zheng, Xiang Yue, Shujian Huang, Jiajun Chen, and Alexandra Birch. 2020. Towards Making the Most of Context in Neural Machine Translation. In *IJCAI*. ijcai.org, 3983–3989.

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

[104] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. 2015. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 415–425.