The Journal of Systems and Software 216 (2024) 112146

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

# End-to-end log statement generation at block-level☆

Ying Fu[a], Meng Yan[a,*], Pinjia He[b], Chao Liu[a], Xiaohong Zhang[a], Dan Yang[c]

[a] School of Big Data and Software Engineering, Chongqing University, Chongqing, China
[b] School of Data Science, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China
[c] Southwest Jiaotong University, Chengdu, China

ARTICLE INFO

Dataset link: https://github.com/cqu-isse/ELogger

*Keywords:*
Log statement
End-to-end
Block-level
Deep learning

ABSTRACT

Logging is crucial in software development for addressing runtime issues but can pose challenges. Logging encompasses four essential sub-tasks: whether to log (Whether), where to log (Position), which log level (Level), and what information to log (Message). While existing approaches have performed well, they suffer from two limitations. Firstly, they address only a subset of the logging sub-tasks. Secondly, most of them focus on generating single log statements at class or method level, potentially overlooking multiple log statements within those scopes.

To address these issues, we propose ELogger, which enables end-to-end log statement generation at block-level. Furthermore, ELogger implements block-level log generation, enabling it to handle multiple log statements within different code blocks of a method. Evaluation results indicate that ELogger correctly predicts all four sub-tasks in 19.55% of cases. Compared to the baselines that combined existing approaches for end-to-end log statement generation, ELogger demonstrates a significant improvement with a 50.85% to 78.21% average increase. Additionally, ELogger correctly predicts whether to log in 71.68% of cases, two sub-tasks (Whether and Position) in 58.29% of cases, and three sub-tasks (Whether, Position, and Level) in 41.97% of cases, all of which outperform the baselines.

## 1. Introduction

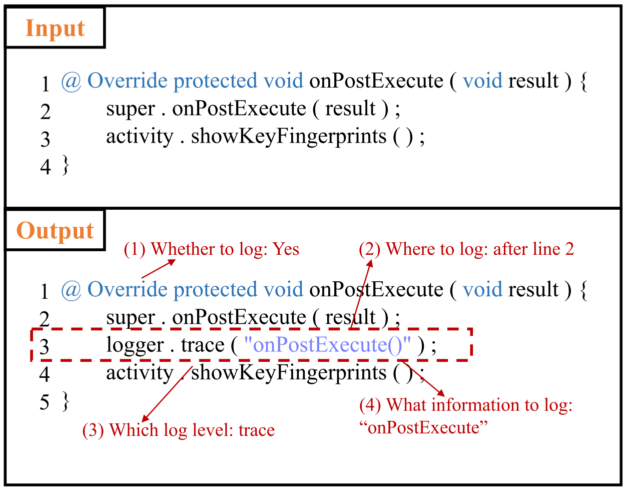
Logging statements are inserted in the source code to record important information during system runtime, such as system status, resource information, and business operation information. The rich runtime information contained in logs is essential for the proper functioning and upkeep of software systems, including the ever-growing popularity of service-oriented systems. It enables developers to diagnose issues that arise during runtime and perform various maintenance tasks, including detecting system anomalies (e.g., Breier and Branišová (2015), Le and Zhang (2021), Liu et al. (2019a), Nandi et al. (2016), Xia et al. (2020), Fu et al. (2023) and Guan et al. (2023)), diagnosing failures (e.g., Babenko et al. (2009), Chen (2019), Jia et al. (2017) and Lu et al. (2018)), predicting failures (e.g., Berrocal et al. (2014), Das et al. (2020, 2018) and Lin et al. (2018)), and system comprehension (e.g., Nagappan et al. (2009) and Nagaraj et al. (2012)). Recent studies have further integrated logs with traces to perform maintenance tasks in service-oriented systems because logs can provide fault information that is not captured by traces (Zhang et al., 2022; Lee et al., 2023; Yu et al., 2023).

Fig. 1 illustrates the four essential sub-tasks of logging practices, which include: (1) Whether to log (Whether); (2) Where to log (Position); (3) Which log level (Level); and (4) What information to log (Message). During development, the decision-making process of developers begins by determining whether a log statement is needed in the code (*Yes*). Next, they identify the location within the code where the log statement should be inserted (*after line 2*). Then, determining which log level (e.g., info, trace, error, and debug) is appropriate for the severity of the logged event (*trace*). Finally, defining what relevant contextual information should be logged in the log message (*"onPostExecute()"*).

Many studies have demonstrated that crafting appropriate logging statements is a crucial yet challenging task (Yuan et al., 2012; Shang et al., 2014; Chen et al., 2017; Zeng et al., 2019). Each of the four sub-tasks within logging practice presents its unique challenges. Firstly, deciding whether to log presents a delicate balance; coarse-grained logs may obscure critical runtime information, while fine-grained ones can strain system resources. Secondly, identifying where to log requires

☆ Editor: Gabriele Bavota.
* Corresponding author.
*E-mail addresses:* fuying@cqu.edu.cn (Y. Fu), mengy@cqu.edu.cn (M. Yan), hepinjia@cuhk.edu.cn (P. He), liu.chao@cqu.edu.cn (C. Liu), xhongz@cqu.edu.cn (X. Zhang), dyang@cqu.edu.cn (D. Yang).

https://doi.org/10.1016/j.jss.2024.112146
Received 5 October 2023; Received in revised form 31 May 2024; Accepted 18 June 2024
Available online 21 June 2024


Input
Output
(1) Whether to log: Yes
(2) Where to log: after line 2
(3) Which log level: trace
(4) What information to log: "onPostExecute"

```
Input
1 @ Override protected void onPostExecute ( void result ) {
2       super . onPostExecute ( result ) ;
3       activity . showKeyFingerprints ( ) ;
4 }

Output
1 @ Override protected void onPostExecute ( void result ) {
2       super . onPostExecute ( result ) ;
3       logger . trace ( "onPostExecute()" ) ;
4       activity . showKeyFingerprints ( ) ;
5 }
```

**Fig. 1.** The four essential sub-tasks of logging practices.

strategic placement to capture relevant runtime data without unintended consequences. Thirdly, determining which log level to use is crucial; opting for lower levels, like 'debug' for critical events may hinder diagnosis, while choosing higher levels for trivial events can confuse users and add to log management overhead. Finally, defining what information to log is essential; developers often rely on log statement text, but misleading text can render logs counterproductive.

Recently, researchers have proposed various approaches to assist developers in logging practices, with the goal of reducing developer effort and enhancing the quality of log statements. For example, H.Li et al. and Z.Li et al. focused on the 'whether to log' sub-task (Li et al., 2018, 2020a). Liu et al. focused on the 'where to log' sub-task (Liu et al., 2019b). Li et al. focused on the 'which log level' sub-task (Li et al., 2021). Ding et al. focused on the 'what information to log' sub-task (Ding et al., 2022). Additionally, Mastropaolo et al. proposed LANCE, an approach that addresses three sub-tasks (where to log, which log level, and what information to log) for Java methods (Mastropaolo et al., 2022). It is important to highlight that LANCE operates under the assumption that a method should log and subsequently addresses these three. However, in practice, developers need to first determine if a method requires logging or not (the first sub-task). Thus, LANCE does not provide an end-to-end solution, and trivially applying LANCE to every method will inevitably lead to excessive log usage, which can bring heavy runtime overhead to the systems.

Although existing approaches have achieved notable performance, they still suffer from two major limitations. Firstly, current research can only provide partial support to developers by addressing a subset of the sub-tasks involved in logging practices, such as whether or what information to log (Li et al., 2020a; Ding et al., 2022). However, the demand for comprehensive support in logging practices persists among developers. Integrating existing methods to offer end-to-end log generation support presents intricate challenges. This involves separate data acquisition and feature extraction processes for each model during the model-building phase due to the distinct data types and granularity employed by each model. Furthermore, the need to switch between multiple models during the model usage phase arises, potentially leading to increased model complexity. From our perspective, adopting an end-to-end log generation approach holds greater suitability for developers. Secondly, a substantial number of existing approaches focus on generating single log statements at the class or method level, yet classes or methods often contain multiple log statements. This can result in the omission of several log statements when using a class-level or method-level approach. For example, referring to the statistical data from the training dataset presented by Mastropaolo et al. (2022), approximately 37.43% of the logged methods contain more than one log statement. This observation prompts us to advocate for a more granular approach to log generation, specifically at the block level within a method. Implementing log generation at this finer granularity can effectively alleviate the issue of multiple log statements within a class or a method.

In this paper, we propose ELogger (**E**nd-to-End **Log** Statement **Gener**ation), an approach for end-to-end log statement generation at block-level. The end-to-end log statement generation refers to complete all four essential sub-tasks of logging practices in a single unified process. ELogger comprises two components, whether-to-log and www-log, to facilitate the process. To reduce the complexity of integrating diverse tasks and streamline the workflow of method, ELogger employs a consistent feature representation method across both components. Meanwhile, ELogger implements end-to-end log generation at the block-level, effectively alleviating the issue of multiple log statements across different code blocks of a class or a method.

We evaluated ELogger using the dataset provided by Mastropaolo et al. (2022) and found that it can correctly predict all four sub-tasks (Whether, Position, Level, and Message) in 19.55% of cases. ELogger demonstrates a significant improvement over the baselines, which incorporated existing approaches to support end-to-end log statement generation, achieving an average increase of 50.85% to 78.21% in generating correct end-to-end log statements (all four sub-tasks). Furthermore, ELogger can predict whether to log in 71.68% of cases, two of the four sub-tasks (Whether and Position) in 58.29% of cases, three of the four sub-tasks (Whether, Position, and Level) in 41.97% of cases, all of which outperform the baselines. These results suggest that ELogger has the potential to automate the process of end-to-end log statement generation. In summary, this paper makes the following contributions:

(1) We propose ELogger, an end-to-end log statement generation approach that completes the four essential sub-tasks of logging practices in a single unified process. Additionally, we open source our replication package[1] for further studies.

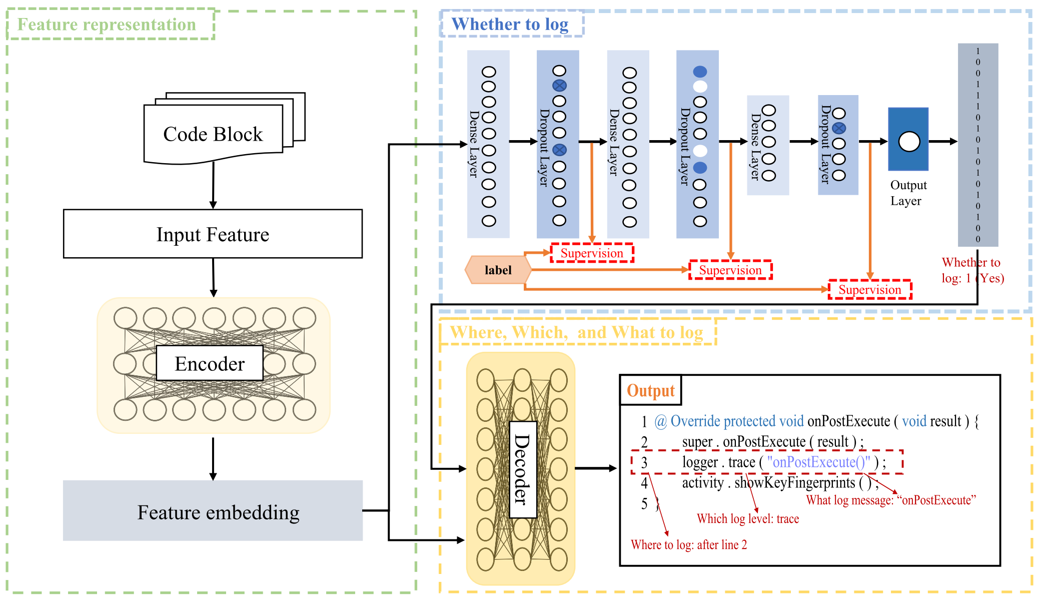
(2) We conducted thorough experiments to evaluate the performance of ELogger and found that it significantly outperforms the baselines that combined existing approaches to provide end-to-end log statement generation support. Additionally, we found that both components of ELogger contribute significantly to its effectiveness.

**Paper organization.** Section 2 presents the details of our approach. Section 3 presents the experimental settings. Section 4 details the experimental results. Section 5 reviews the related studies. Section 6 discusses the impact of different pre-trained model on our approach and the limitations of our approach. Section 7 draws a conclusion for this paper.

## 2. Approach

This paper proposes ELogger, consisting of two components: whether-to-log and www-log, designed for end-to-end log statement generation at block-level. Both components of ELogger utilize a consistent feature representation method, thereby reducing the complexity of integrating the four sub-tasks (whether to log, where to log, which log level, and what information to log) in logging practices and streamlining the workflow of method. The whether-to-log component focuses on predicting whether logging is needed, treating it as a binary classification problem. The www-log component addresses the remaining three sub-tasks, which include generating a logging statement and seamlessly injecting it into the appropriate position within the code block that requires logging. It treats the generation and injection of a logging statement as a text generation task.

[1] https://github.com/cqu-isse/ELogger.



```
1 @ Override protected void onPostExecute ( void result ) {
2     super . onPostExecute ( result ) ;
3     logger . trace ( "onPostExecute()" ) ;
4     activity . showKeyFingerprints ( ) ;
5 }
```

**Fig. 2.** The framework of ELogger.

### 2.1. Overview of our approach

The overall framework of ELogger is illustrated in Fig. 2. ELogger consists of a whether-to-log component and a www-log component.

(1) **Feature representation**. The encoder from the pre-trained model is used to convert the input features into feature embeddings. These embeddings serve as a uniform input to both the "whether-to-log" and "www-log" components.

(2) **Whether-to-log component**. The whether-to-log component employs a deep supervised network to process feature embeddings generated during the feature representation stage. It determines whether logging is necessary for each code block based on these embeddings.

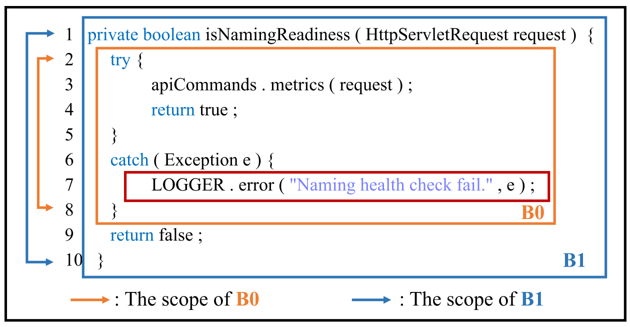
(3) **Www-log component**. After identifying code blocks that require logging through the whether-to-log component, the www-log component receives and processes the feature embeddings of these identified code blocks. It is specifically designed to generate complete log statements and seamlessly inject them into the appropriate positions within these identified code blocks.

The use of the consistent feature representation by both the whether-to-log and www-log components allows ELogger to integrate the four sub-tasks of logging practices for implementing end-to-end log statement generation. This strategy streamlines the workflow and reduces the complexity of integrating diverse tasks. During the training phase, we use only blocks that require log statement insertion to train the www-log component by removing non-logged blocks from the training and validation datasets. During the testing phase, we first use the whether-to-log component to identify the blocks that require log statement insertion and then use the www-log component to produce complete log statements for these blocks in the test dataset.

We select GraphCodeBERT as the pre-trained model for our approach based on its effectiveness in code related tasks, relatively smaller model size, and the moderate computational resources it requires for fine-tuning. Additionally, we implement the variants of ELogger based on other pre-trained models, including UniXcoder (Guo et al., 2022) and CodeT5+ (Wang et al., 2023). The impact of different pre-trained models on our method is further discussed in Section 6.1.

### 2.2. Feature representation

Firstly, we preprocess the methods into code blocks. For example, consider the method depicted in Fig. 3, which is preprocessed into two code blocks. Block B0 spans from line 2 to line 8, while block



```
1  private boolean isNamingReadiness ( HttpServletRequest request )  {
2      try {
3          apiCommands . metrics ( request ) ;
4          return true ;
5      }
6      catch ( Exception e ) {
7          LOGGER . error ( "Naming health check fail." , e ) ;
8      }
9      return false ;
10 }
```

**Fig. 3.** An example of how to extract code blocks and label them.

B1 encompasses lines 1 through 10. Secondly, the encoder, derived from the pre-trained model, is employed to encode the input features, thereby creating a denser representation. It is noteworthy that the encoder's output assumes the role of input for both the whether-to-log component and the www-log component enabling a consistent approach to providing relevant feature information for each sub-task.

### 2.3. Whether-to-log component

The whether-to-log component serves the purpose of suggesting whether to log for the given code block. Comprising three hidden layers, three dropout layers and an output layer, this component's architecture is depicted in the upper right section of Fig. 2. To address the risk of overfitting and reduce the potential impact of excessive dependence on the training data, we included a dropout layer with a dropout rate of 0.2 in all the hidden layers. Furthermore, we implemented an integrated direct supervision approach to the hidden layers, rather than the conventional method of providing supervision solely at the output layer and then propagating it back to earlier layers. The integrated direct hidden layer supervision is achieved by incorporating auxiliary classifiers for each hidden layer, which serve as soft constraints during the learning process. To address the class imbalance in the dataset and balance precision and recall, focal loss (Lin et al., 2017) was utilized as a modification of the cross-entropy loss function. Focal loss assigns a higher weight to hard-to-classify examples, which helps the model to focus more on the minority class and improve its overall performance on the imbalanced dataset. Focal loss can be computed using Eq. (1).

$$FL(p_t) = -\alpha_t(1-p_t)^{\gamma}\log(p_t) \tag{1}$$

where $p_t$ is the predicted probability of the true class, $\alpha_t$ is the weighting factor for balancing the contribution of the positive and negative class samples, $\gamma$ is the focusing parameter that reduces the loss contribution of well-classified examples and focuses more on hard examples. Typically, the value of $\gamma$ is set to 2.

In the training, we use fuse loss as training loss. The training loss can be computed as Eq. (2). Let $\ell_m$ denote the loss function for the $m$th auxiliary classifier, where $W$ and $w_b^{(m)}$ are the parameters of the mainstream network and the weights of the auxiliary classifier in the $m$th branch, respectively.

$$\ell_{fuse}(W, w_b) = \sum_{m=1}^{M} \ell_m(W, w_b^{(m)}) \tag{2}$$

To make a binary suggestion of whether a code block should be logged or not, we utilize a one-dimensional dense layer with a sigmoid activation function as the output layer. The output of this layer represents the final suggestion for the given code block, indicating whether it should be logged or not.

### 2.4. Www-log component

The www-log component considers the generation and injection of a logging statement as a text generation task. It carries out this task by concurrently predicting the log position, log level, and log message within a single pass. For generating the target code, ELogger employs a transformer-based decoder comprised of 6 stacked transformer blocks. This decoder receives the feature embeddings from the encoder as input. During training, we utilized teacher forcing, where the ground truth tokens from the output sequence were used as inputs to the decoder at each time step. The objective of the decoder during training is to minimize cross-entropy by minimizing the following objective function:

$$H(y) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{l} logp(y_j^{(i)}) \tag{3}$$

where N is the total number of training instances, and $l$ is the length of each target sequence. $y_j^{(i)}$ denotes the $j$th word in the $i$th instance. The cross-entropy function measures the difference between the predicted probability and the ground truth. We optimized the objective function using the AdamW (You et al., 2019) optimizer, which includes weight decay regularization to prevent overfitting. During inference, we utilized beam search to generate the most probable output sequence.

## 3. Experimental setup

In this section, we introduce our research questions, selected dataset, evaluation metrics, and baseline approaches.

### 3.1. Research questions

We want to investigate the following research questions:

- RQ1: How effective is ELogger?
- RQ2: How effective is ELogger for different categories of code blocks?
- RQ3: How effective are the main components of ELogger?

### 3.2. Dataset

We utilized the dataset provided by Mastropaolo et al. (2022), which comprises 1465 Java projects obtained from GitHub. The dataset consists of two pre-training datasets and one fine-tuning dataset. In our study, we use only the fine-tuning dataset because we use the mainstream pre-trained model GraphCodeBert to implement our method, eliminating the need for retraining with a separate pre-training dataset. Table 1 presents the statistics of the fine-tuning dataset, which include the number of methods, log statements, and code blocks. The training set has 61,597 methods, the validation set has 7699 methods, and the test set has 7125 methods. Because each instance corresponds to a method with a specific log statement removed, methods with multiple log statements may generate more than one instance. For example, the test set has a total of 12,020 log statements, resulting in 12,020 instances in the test set. To enable ELogger to generate end-to-end log statements at the block-level, we transformed all methods in the dataset into code blocks during the data processing stage. Fig. 3 depicts an example, where block B0 spans from line 2 to line 8, and block B1 spans from line 1 to line 10. Consequently, the training set has 453,644 blocks, the validation set has 56,080 blocks, and the test set has 52,796 blocks.

**Table 1**
The statistics of the dataset.

| Type | # method | # log statement | # block | # logged block |
|---|---|---|---|---|
| Train | 61,597 | 106,382 | 453,644 | 106,382 |
| Valid | 7699 | 13,260 | 56,080 | 13,260 |
| Test | 7125 | 12,020 | 52,796 | 12,020 |

To train the whether-to-log component, each block in the dataset was labeled as either a logged block or a non-logged block based on whether the block directly contained a logging statement. Consistent with Li et al. (2020a), only blocks that directly contained a logging statement were labeled as logged blocks. For instance, in Fig. 3, block B0 was labeled as a logged block because it contains a logging statement in line 7. However, block B1 was labeled as a non-logged block because the logging statement in line 7 is not directly contained within block B1. An assignment statement was used instead of a nested code block to reduce the interference of redundant information. For example, block B0 was replaced by "code_block = TryStatement". Consistent with Mastropaolo et al. a block that has multiple log statements may produce more than one instance.It is worth noting that due to the way Mastropaolo et al. built their dataset, a number of repetitive blocks are generated when methods are processed into blocks. When building our training dataset, we removed the duplicate blocks.

### 3.3. Experimental settings

The hidden layers of the whether-to-log component consist of two dense layers with a size of 512, followed by a dense layer with a size of 256. For the www-log component, we use 512 embedding dimensions for both the encoder and the decoder, as referenced in Mastropaolo et al. (2022), and a batch size of 32 for training. It should be noted that we only utilized the blocks that require log statement insertion in the training dataset to train the www-log. The training of our models is conducted on a cluster of machines, each with a Tesla V100 GPU.

### 3.4. Evaluation metrics

The performance of our approach in end-to-end log statement generation includes the ability to suggest whether to log, place the log statement in the correct position, recommend the correct log level, and generate a reasonable log description. Consistent with prior studies (Li et al., 2020a; Mastropaolo et al., 2022), we evaluated our approach using the following metrics: Balanced Accuracy, Precision, Recall, F1 Score, Position Accuracy, Level Distance, BLEU, METEOR, and ROUGE-LSC.

Given a code block as input, the whether-to-log component predicts whether to insert a logging statement, treating it as a binary classification problem. We measure the effectiveness of this component using **Balanced Accuracy, Precision, Recall, and F1 Score**. Balanced accuracy is a widely used metric to evaluate model performance on imbalanced data (Li et al., 2017; Zhu et al., 2015), and is computed using Eq. (4) where TP, TN, FP, and FN refer to True Positive, True Negative, False Positive, and False Negative. Precision measures the ability of the whether-to-log component to suggest logged code blocks correctly, and is computed using Eq. (5). Recall measures the ability of the whether-to-log component to find logged code blocks from the dataset, and is computed using Eq. (6). F1 score considers both precision and recall, and is computed using Eq. (7).

$$Balanced\ Accuracy = (\frac{TP}{TP+FN} + \frac{TN}{TN+FP})/2 \tag{4}$$

$$Precision = \frac{TP}{TP+FP} \tag{5}$$

$$Recall = \frac{TP}{TP+FN} \tag{6}$$

$$F1\ Score = 2 * (\frac{Precision * Recall}{Precision + Recall}) \tag{7}$$

**Position Accuracy** is used to measure the correctness of position prediction. A higher Position Accuracy indicates that more log statements are inserted into the correct position of the method.

**Level Distance** measures the deviation between the actual log level and the suggested log level for each logging statement, with a value between 0 (correct prediction) and 5 (worst-case scenario where a Trace is suggested instead of a Fatal, or vice versa). A smaller level distance signifies that the recommended log levels are closer to their actual log levels.

**BLEU** (Bilingual Evaluation Understudy) is a widely-used evaluation metric in text summarization and machine translation tasks (Papineni et al., 2002; Bahdanau et al., 2014). It measures the match between the generated text and the reference text. We use BLEU to evaluate the match between the generated logging description and the reference logging description. BLEU scores can be calculated for different sequence lengths by considering up to four values of $n$ (namely, BLEU-1, BLEU-2, BLEU-3, and BLEU-4), allowing for a comprehensive analysis of both the lexical similarity and the structural coherence of the text. BLEU is calculated as follows:

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{8}$$

$BP$, which penalizes excessively short candidates; $N$, which represents the maximum number of n-grams used in the experiments; $p_n$, the modified n-gram precision; and $w_n$, the positive weight of each $p_n$. $BP$ is calculated using the following equation:

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \le r \end{cases} \tag{9}$$

where $c$ is the length of the generated logging description and $r$ is the length of the reference logging description.

**METEOR** (Metric for Evaluation of Translation with Explicit Ordering) is a metric used to evaluate machine translation by comparing the alignment of words between reference text and machine-translated outputs (Banerjee and Lavie, 2005). It improves upon BLEU by considering synonyms and stemming, and emphasizes recall in its calculation of precision and recall; METEOR also penalizes translations with incorrect word order, addressing BLEU's limitations like its focus on exact word matches. This makes METEOR more aligned with human perceptions of sentences that have similar meanings.

**ROUGE** (Recall-Oriented Understudy for Gisting Evaluation) is a collection of metrics used primarily for evaluating automatic summarization of texts and machine translation (Lin, 2004). We specifically employ the ROUGE-LCS (Longest Common Subsequence) variant, which measures three aspects: ROUGE-LCS precision, ROUGE-LCS recall, and ROUGE-LCS F-measure. ROUGE-LCS precision is calculated as the ratio of the longest common subsequence (LCS) between two token sequences, X and Y, to the length of Y, and ROUGE-LCS recall is defined as the ratio of LCS to the length of X. The ROUGE-LCS F-measure is the harmonic mean of both ROUGE-LCS precision and ROUGE-LCS recall.

### 3.5. Baseline approaches

ELogger is an end-to-end log statement generation approach that addresses all four sub-tasks of logging practices. In contrast, LANCE, the state-of-the-art complete log statement generation method proposed by Mastropaolo et al. (2022), focuses on three sub-tasks (where to log, which log level, and what information to log). Since LANCE does not address the 'whether to log' sub-task, making a direct comparison with ELogger becomes challenging. We combined the approach proposed by Li et al. (2020a) with LANCE to create our first baseline approach named Li-LANCE. Li et al.'s approach is currently the state-of-the-art for automatically suggesting logging locations at the block level. The Li-Model utilizes an RNN to model the relationship between a block's logging decision and its syntactic, semantic, and fused features. We implemented the Li-Model using the replication package provided by Li et al. ensuring consistency by adhering to their parameter settings. Mastropaolo et al. initially pre-trained the T5 model using a set of Java methods and subsequently fine-tuned the pre-trained model to develop LANCE. To ensure a fair comparison, we adopted the parameter settings from their replication package and fine-tuned LANCE for 200,000 steps with code block data, enhancing its ability to process code blocks effectively. Our second baseline approach, named W-LANCE, combines our whether-to-log component with LANCE.

## 4. Evaluation results

### 4.1. RQ1: The effectiveness of ELogger

**Motivation.** In this RQ, we aim to evaluate the effectiveness of ELogger for end-to-end log statement generation at code blocks. We will evaluate ELogger's performance in four sub-tasks in logging practices: (1) determining whether to log (Whether), (2) identifying where to log (Position), (3) determining which log level to use (Level), and (4) defining what information to log (Message). By evaluating ELogger's performance in these sub-tasks, we can assess the overall effectiveness of ELogger for end-to-end log statement generation in Java blocks. Simultaneously, we will perform the multi-log injection analysis to assess the capability of ELogger in inserting multiple log statements into a method.

**Method.** To evaluate the effectiveness of ELogger in end-to-end log statement generation at block-level, we compared it to two baseline approaches, Li-LANCE and W-LANCE, as described in Section 3.5. As the sub-tasks involved in logging practices are interdependent, whether to log is a prerequisite for where to log. Where to log, in turn, is necessary for which log level and what information to log. We evaluated the combined performance of ELogger and the two baseline approaches in completing all four sub-tasks. To examine the statistical significance of the difference between the baseline approaches and ELogger, we utilize the Wilcoxon test (Wilcoxon, 1992) at a 95% confidence level. This non-parametric hypothesis test compares two matched samples to determine whether their population mean ranks differ. Additionally, we employ Cliff's delta to quantify the effect size. Cliff's delta, a non-parametric effect size measure, quantifies the extent of difference between two approaches. It categorizes delta values as follows: less than 0.147 as "Negligible (N)", between 0.147 and 0.33 as "Small (S)", between 0.33 and 0.474 as "Medium (M)", and above 0.474 as "Large (L)" effect size (Cliff, 2014).

To more thoroughly evaluate the quality of the log messages generated by ELogger and W-LANCE, we employ a comprehensive set of metrics. These include BLEU-1, BLEU-2, BLEU-3, BLEU-4, METEOR, and ROUGE-LCS, which enable us to quantitatively assess the effectiveness of the generated log messages themselves. Additionally, for the qualitative analysis of inaccurately generated log messages, we manually inspected 200 partially correct predictions made by ELogger and W-LANCE. The objective of this inspection is to investigate whether the inaccurately generated log messages are semantically equivalent to

**Table 2**
Effectiveness of ELogger.

| | Log generation sub-task | | | | Li-LANCE | W-LANCE | ELogger |
|---|---|---|---|---|---|---|---|
| | Whether | Position | Level | Message | Preds(%) | Preds(%) | Preds(%) |
| 1 out of 4 | ✓ | – | – | – | 65.30 (↑9.77%) | 71.68 (↑0.00%) | 71.68 |
| 2 out of 4 | ✓ | ✓ | – | – | 45.11 (↑29.22%) | 48.74 (↑19.59%) | 58.29 |
| 3 out of 4 | ✓ | ✓ | ✓ | – | 29.92 (↑40.27%) | 33.27 (↑26.15%) | 41.97 |
| | ✓ | ✓ | – | ✓ | 11.83 (↑79.04%) | 13.93 (↑52.05%) | 21.18 |
| | ✓ | ✓ | ✓ | ✗ | 18.95 (↑18.31%) | 20.31 (↑10.39%) | 22.42 |
| | ✓ | ✓ | ✗ | ✓ | 0.86 (↑88.37%) | 0.98 (↑65.31%) | 1.62 |
| All | ✓ | ✓ | ✓ | ✓ | 10.97 (↑78.21%) | 12.96 (↑50.85%) | 19.55 |
| p-value | | | | | <0.05 | <0.05 | |
| Cliff's Delta | | | | | 0.265 (S) | 0.204 (S) | |

↑ denotes performance improvement against the baseline. In the first row, ELogger and W-LANCE have the same performance since they use the same component for this sub-task.

the ground truth. To ensure the reliability of our findings, two of the authors independently reviewed all 400 log messages, and any conflicts were resolved by a third reviewer.

To assess the capability of ELogger in inserting multiple log statements into a method, we first reassemble the code blocks it generates back into complete methods. We then measure the accuracy by the number of instances where ELogger correctly inserts all intended log statements. Partially correct predictions are not included in this calculation because if any part of the code block within a method is inaccurately predicted, it implies that ELogger did not successfully inject the full set of required log statements.

**Results.** As the results are shown in Table 2, Figs. 4, 5, 6, 7, and 8, we make the following observations:

Table 2 shows the effectiveness of the four approaches (Li-LANCE, W-LANCE, and ELogger) across the four logging practices sub-tasks (Whether, Position, Level, and Message). The table is organized in such a way that each row corresponds to one of the four tasks. A check mark (✓) under a task indicates that the corresponding approach correctly predicted the task for the log statements in that row. A dash mark (–) indicates that the approach could be either correct or incorrect for the predictions in that row. Finally, a cross mark (✗) indicates that the corresponding approach predicted the task incorrectly for the log statements in that row. For example, the second row (Whether= ✓∧ Position = ✓∧ Level = –∧ Message = –) shows the percentage of correct predictions for whether to log and the position of the log statement, regardless of the correctness of the predicted log level and message. The final row, labeled as "All" represents the most challenging scenario in which the generated log statement must be identical to the reference log statement for all tasks.

The first row of Table 2 indicates that ELogger's ability to correctly predict whether to log is 71.68%, which surpasses Li-LANCE's performance of 65.30%. ELogger and W-LANCE have the same performance in predicting whether to log since they use the same component for this sub-task. The second row indicates that ELogger correctly predicts two out of the four sub-tasks (Whether and Position) in 58.29% of cases, surpassing both Li-LANCE (45.11%) and W-LANCE (48.74%). Moving on to the third and fourth rows, ELogger correctly predicts three out of the four sub-tasks (Whether, Position, and Level) in 41.97% of cases, and another three out of the four sub-tasks (Whether, Position, and Message) in 21.18% of cases. This two-fold achievement significantly surpasses the performance of Li-LANCE (29.92% and 11.83%, respectively) and W-LANCE (33.27% and 13.93%, respectively). Finally, in the last row, ELogger correctly predicts all four sub-tasks (Whether, Position, Level, and Message) in 19.55% of cases, surpassing both Li-LANCE (10.97%) and W-LANCE (12.96%). Compared to Li-LANCE and W-LANCE, ELogger outperforms the two baselines in terms of all the measures with a statistical significance (i.e., p-value $< 0.05$) and a non-negligible effect size according to Cliff's delta from Table 2.

Fig. 4 presents the number of unique correct predictions generated by ELogger compared with Li-LANCE and W-LANCE. We observe that

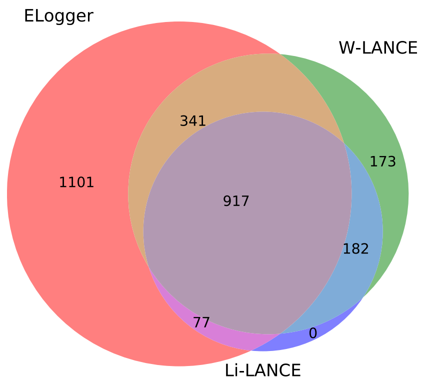


**Fig. 4.** Correct predict venn diagram.

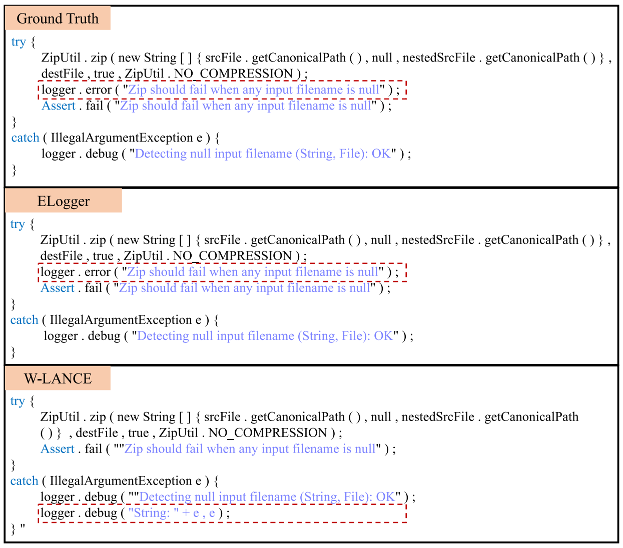
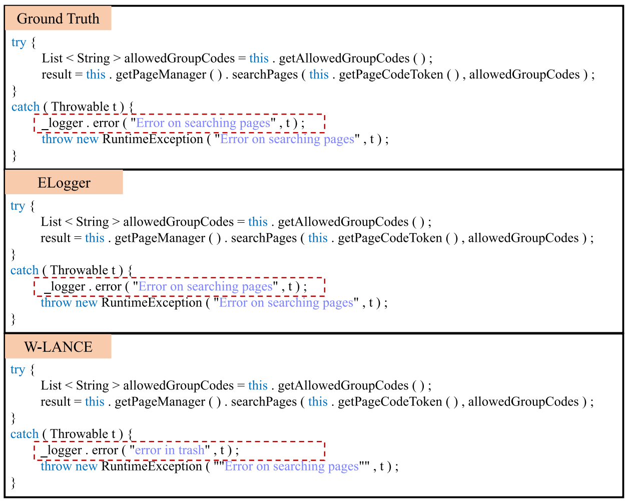

Ground Truth
```
try {
      ZipUtil . zip ( new String [ ] { srcFile . getCanonicalPath ( ) , null , nestedSrcFile . getCanonicalPath ( ) } ,
      destFile , true , ZipUtil . NO_COMPRESSION ) ;
      logger . error ( "Zip should fail when any input filename is null" ) ;
      Assert . fail ( "Zip should fail when any input filename is null" ) ;
}
catch ( IllegalArgumentException e ) {
      logger . debug ( "Detecting null input filename (String, File): OK" ) ;
}
```
ELogger
```
try {
      ZipUtil . zip ( new String [ ] { srcFile . getCanonicalPath ( ) , null , nestedSrcFile . getCanonicalPath ( ) } ,
      destFile , true , ZipUtil . NO_COMPRESSION ) ;
      logger . error ( "Zip should fail when any input filename is null" ) ;
      Assert . fail ( "Zip should fail when any input filename is null" ) ;
}
catch ( IllegalArgumentException e ) {
       logger . debug ( "Detecting null input filename (String, File): OK" ) ;
}
```
W-LANCE
```
try {
      ZipUtil . zip ( new String [ ] { srcFile . getCanonicalPath ( ) , null , nestedSrcFile . getCanonicalPath
      ( ) }  , destFile , true , ZipUtil . NO_COMPRESSION ) ;
      Assert . fail ( ""Zip should fail when any input filename is null" ) ;
}
catch ( IllegalArgumentException e ) {
      logger . debug ( ""Detecting null input filename (String, File): OK" ) ;
      logger . debug ( "String: " + e , e ) ;
} "
```

**Fig. 5.** The unique correct prediction by ELogger - Case 1.

ELogger generates 1101 unique correct predictions, significantly outperforming W-LANCE, which produces 173, and Li-LANCE, which fails to generate any. This indicates that ELogger is more effective in generating unique correct predictions. Further analysis revealed that, out of the 1101 unique correct predictions by ELogger, W-LANCE incorrectly predicted the position in 738 instances and either the position or level in 863 instances. Figs. 5 and 6 illustrate two unique correct predictions generated by ELogger. As shown in Fig. 5, ELogger correctly predicted the position, level, and log message whereas W-LANCE failed to correctly predict these elements. Fig. 6 shows that although W-LANCE


Ground Truth

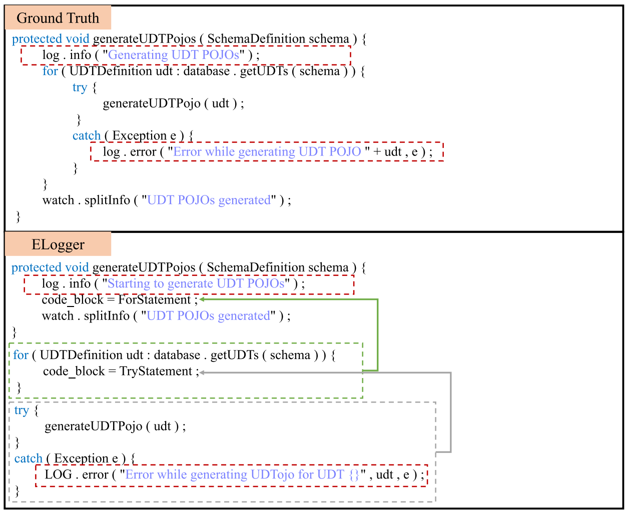
```
try {
    List < String > allowedGroupCodes = this . getAllowedGroupCodes ( ) ;
    result = this . getPageManager ( ) . searchPages ( this . getPageCodeToken ( ) , allowedGroupCodes ) ;
}
catch ( Throwable t ) {
    _logger . error ( "Error on searching pages" , t ) ;
    throw new RuntimeException ( "Error on searching pages" , t ) ;
}
```

ELogger

```
try {
    List < String > allowedGroupCodes = this . getAllowedGroupCodes ( ) ;
    result = this . getPageManager ( ) . searchPages ( this . getPageCodeToken ( ) , allowedGroupCodes ) ;
}
catch ( Throwable t ) {
    _logger . error ( "Error on searching pages" , t ) ;
    throw new RuntimeException ( "Error on searching pages" , t ) ;
}
```

```
try {
    List < String > allowedGroupCodes = this . getAllowedGroupCodes ( ) ;
    result = this . getPageManager ( ) . searchPages ( this . getPageCodeToken ( ) , allowedGroupCodes ) ;
}
catch ( Throwable t ) {
    _logger . error ( "error in trash" , t ) ;
    throw new RuntimeException ( ""Error on searching pages"" , t ) ;
}
```

**Fig. 6.** The unique correct prediction by ELogger - Case2.


Ground Truth

```
protected void generateUDTPojos ( SchemaDefinition schema ) {
    log . info ( "Generating UDT POJOs" ) ;
    for ( UDTDefinition udt : database . getUDTs ( schema ) ) {
        try {
            generateUDTPojo ( udt ) ;
        }
        catch ( Exception e ) {
            log . error ( "Error while generating UDT POJO " + udt , e ) ;
        }
    }
    watch . splitInfo ( "UDT POJOs generated" ) ;
}
```

ELogger

```
protected void generateUDTPojos ( SchemaDefinition schema ) {
    log . info ( "Starting to generate UDT POJOs" ) ;
    code_block = ForStatement ;
    watch . splitInfo ( "UDT POJOs generated" ) ;
}
for ( UDTDefinition udt : database . getUDTs ( schema ) ) {
    code_block = TryStatement ;
}
try {
    generateUDTPojo ( udt ) ;
}
catch ( Exception e ) {
    LOG . error ( "Error while generating UDTojo for UDT {}" , udt , e ) ;
}
```

**Fig. 7.** The correct prediction made by ELogger when more than one log statement needs to be inserted into a method.

**Table 3**
The quality of log messages generated by ELogger and W-LANCE.

| Metrics | W-LANCE (%) | ELogger (%) |
|---|---|---|
| BLEU-1 | 46.21 | **51.63** |
| BLEU-2 | 37.30 | **43.97** |
| BLEU-3 | 31.85 | **39.22** |
| BLEU-4 | 26.83 | **34.46** |
| METEOR | 47.34 | **52.58** |
| ROUGE-LCS precision | 53.88 | **64.80** |
| ROUGE-LCS recall | 53.42 | **57.53** |
| ROUGE-LCS F-measure | 52.07 | **59.40** |

correctly predicted the position and level, it incorrectly predicted the log message.

Further examination of the quality of the generated log messages themselves, Table 3 presents the results calculated using the metrics (BLEU, METEOR, and ROUGE) outlined in Section 3.4 for both ELogger and W-LANCE. The results across all metrics indicate that the log messages generated by ELogger are better than W-LANCE.

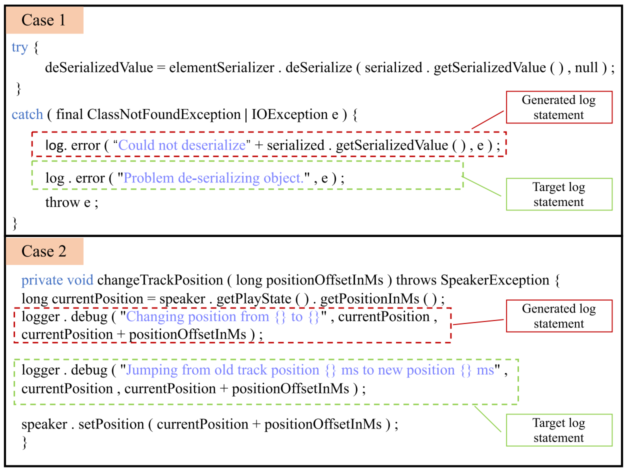
For the qualitative analysis, we manually inspected 200 partially correct log messages generated by ELogger and W-LANCE. Conflicts that arose in 42 cases (10.50%) were resolved with the assistance of a third reviewer. The results show that 127 (63.50%) of the log messages generated by ELogger conveyed the same information as the target log message, despite not being identical. In contrast, 117 (58.50%) of the log messages generated by W-LANCE were found to be semantically equivalent to their targets. These results further indicate that the log messages generated by ELogger are better than W-LANCE.

To evaluate the capability of ELogger in inserting multiple log statements into a method, we reassemble method from code blocks and then measure accuracy. The original test set comprises 7125 methods, of which 63.82% require a single log statement insertion, and 36.18% necessitate multiple log statements. As previously mentioned, partially correct predictions are not calculable in the scenarios involving multiple log statements. Within the subset requiring the insertion of multiple log statements, ELogger accurately predicts the number of log statements needed in 42.36% of cases and accurately predicts all four subtasks in 8.87%. Fig. 7 displays a correct prediction made by ELogger when more than one log statement needs to be inserted into a method. In the Ground Truth, the method can be processed into three code blocks: a method declaration block, a looping block, and a try-catch block. Log statements are required in both the method declaration and the try-catch blocks. ELogger correctly predicted that both the method declaration block and the try-catch block required log statements, and it generated and inserted semantically equivalent log messages for these blocks.

ELogger demonstrates quite good performance in predicting log level and position, but it shows more room for improvement in the what information to log subtask compared to the others. While ELogger correctly predicted whether to log, position, and level in 41.97% of cases, the overall accuracy in predicting all four sub-tasks is 19.55%. At the same time, the cumulative error caused by the four tasks in log statement generation can lead to a significant decrease in the performance of the final output. Upon manually examining data in which ELogger correctly predicted whether to log, position, and level, we found that some of the log messages generated by ELogger were semantically equivalent but not entirely consistent with messages written by developers. As shown in Fig. 8, Case 1 and Case 2 are two instances where the log statements generated by ELogger are semantically equivalent but not entirely consistent with the messages written by developers. In case 1, the log message generated by ELogger (*"Could not deserialize" + serialized.getSerializedValue(), e*) was even more detailed than the one written by developers (*"Problem de-serializing object"., e*). In case 2, the log message generated by ELogger (*"Changing position from {} to {}", currentPosition, currentPosition + positionOffsetInMs*) conveys the same meaning as the one written by developers (*"Jumping from old track position {} ms to new position {} ms", currentPosition, currentPosition + positionOffsetInMs*), although it uses completely different words.

The differences in performance observed among the four sub-tasks are not surprising. This is because the three sub-tasks of determining whether to log, where to log, and which log level to use involve making decisions within a relatively limited search space. For example, whether to log can only have two possible values, the log level can only take on a few possible values, and the position of log statements is limited to the number of tokens in the method. In contrast, defining what information to log requires understanding the code context and selecting relevant information for debugging and troubleshooting. Moreover, natural language log messages can take on diverse forms, making it less likely to obtain identical sentences.

**Summary for RQ1** ☛ *The performance of ELogger in end-to-end log statement generation is promising, and it outperforms two baseline approaches.*


Case 1
Generated log statement
Target log statement
Case 2
Generated log statement
Target log statement

```
try {
    deSerializedValue = elementSerializer . deSerialize ( serialized . getSerializedValue ( ) , null ) ;
}
catch ( final ClassNotFoundException | IOException e ) {
    log. error ( "Could not deserialize" + serialized . getSerializedValue ( ) , e ) ;
    log . error ( "Problem de-serializing object." , e ) ;
    throw e ;
}
```

```
private void changeTrackPosition ( long positionOffsetInMs ) throws SpeakerException {
long currentPosition = speaker . getPlayState ( ) . getPositionInMs ( ) ;
logger . debug ( "Changing position from {} to {}" , currentPosition ,
currentPosition + positionOffsetInMs ) ;

logger . debug ( "Jumping from old track position {} ms to new position {} ms" ,
currentPosition , currentPosition + positionOffsetInMs ) ;

speaker . setPosition ( currentPosition + positionOffsetInMs ) ;
}
```

**Fig. 8.** The cases where ELogger generates log messages that are semantically equivalent to those written by developers but not consistent with them.

### 4.2. RQ2: The effectiveness of ELogger across different categories of code blocks

**Motivation.** The objective of this research question is to evaluate the performance of ELogger across different categories of code blocks. The ability of ELogger to capture and use different types of information may affect its performance in predicting these block categories. This analysis can provide developers with deeper insights into ELogger's performance across different code block types and offer guidance for better utilization of ELogger in practical applications.

**Method.** We adopted the classification method of Li et al. (2020a) to categorize the preprocessed code blocks into four distinct categories: try-catch, branching, looping, and method declaration blocks. Then, we conducted a quantitative analysis to evaluate the performance of ELogger on each category. This analysis included results on whether to log, the position to inject the log statement, the log level distance, and the BLEU-4 scores of the generated logging message. The assessment of whether to log was based on the test data. Meanwhile, the evaluation of the log position, log level, and BLEU-4 scores focused on the specific code blocks where log statement insertion was required within the test data. This focus was due to the inapplicability of these sub-tasks to code blocks that did not require log statement insertion. In the analysis of log position, we divided the position distance into intervals of 50, with a range from 0 (indicating correct prediction) to 300 (where the predicted location is 250 to 300 tokens away from the target location). For the log level analysis, we prioritized the log levels defined in Log4j, specifically: Trace (1), Debug (2), Info (3), Warn (4), Error (5), and Fatal (6). To assess the quality of the generated logging descriptions, we employ the BLEU-4 metric, which evaluates the alignment between the generated logging message and the reference logging message.

**Results.** As the results are shown in Figs. 9, 10, 11, 12, and Table 4, we make the following observations:

Fig. 9 shows the Balanced Accuracy, Precision, Recall, and F1 of the models when suggesting on different types of blocks. We can observe that ELogger achieves the best performance for the try-catch block across all evaluation metrics (82.84% balanced accuracy, 75.86% precision, 86.77% recall and 80.94% F1), while it exhibits the worst performance for the looping block across all evaluation metrics (64.97% balanced accuracy, 50.00% precision, 32.80% recall and 39.61% F1). The performance for the method-declare block (80.83% balanced accuracy, 75.86% precision, 67.80% recall and 71.61% F1) and the branching block (79.28% balanced accuracy, 69.58% precision, 66.13% recall and 67.81% F1) are close to the overall performance of ELogger (81.72% balanced accuracy, 72.87% precision, 71.68% recall and

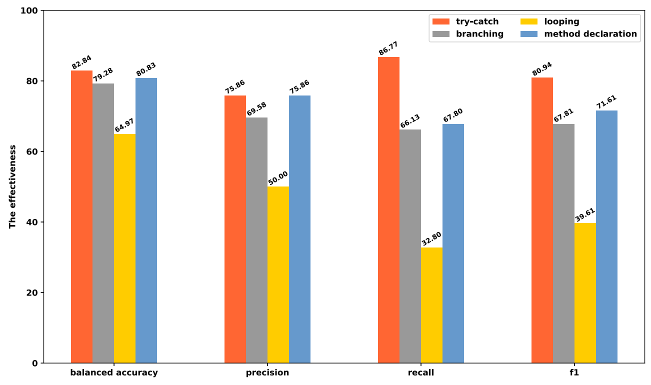


**Fig. 9.** Effectiveness of suggesting whether to log when applied on different types of code block.

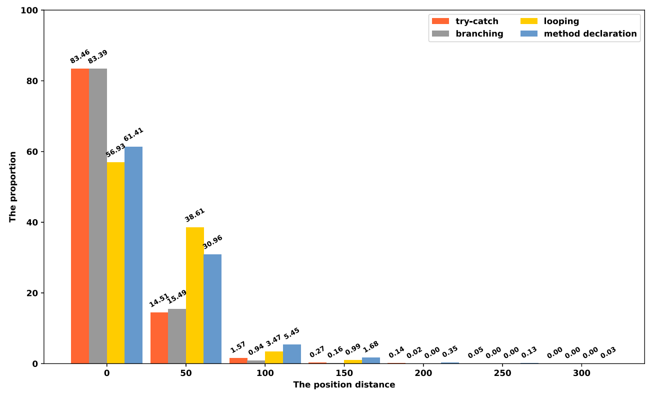


**Fig. 10.** Log distance in the predictions generated by ELogger. Zero indicates predictions having a correct position.

72.27% F1). Overall, the performance of ELogger varies across different types of blocks, with the best performance seen on try-catch blocks and the worst on looping blocks. This variation in performance can be attributed to differences in block complexity and the specific type of information that needs to be logged in each block.

Fig. 10 presents the position distance (in tokens) between the predicted and target locations, allowing us to investigate the performance of ELogger's position prediction. The results reveal that try-catch blocks have the highest position prediction accuracy (83.46%), followed by branching blocks (83.39%). For both types of blocks, most errors occur when the predicted location is less than 50 tokens away from the target (14.51% and 15.49%, respectively). Conversely, the position prediction accuracy for looping blocks (56.93%) is the lowest, followed by method declaration blocks (61.41%). For both types of blocks, the majority of prediction errors also occur when the predicted location is less than 50 tokens away from the target (38.61% and 30.96%, respectively). Therefore, we can conclude that ELogger's position prediction performs better for try-catch and branching blocks than for looping and method declaration blocks. Furthermore, for all block types, most prediction errors occur when the predicted location is within 50 tokens of the target location.

Fig. 11 shows the level distance between the predicted log level and the target log level, ranging from 0 (indicating correct prediction) to 5 (representing the worst-case scenario where a Trace is recommended instead of a Fatal or vice versa). The results show that the level prediction accuracy is highest for method declaration blocks (75.42%), followed by branching blocks (67.37%) and try-catch blocks (67.17%). The majority of errors for method declaration blocks, branching, and

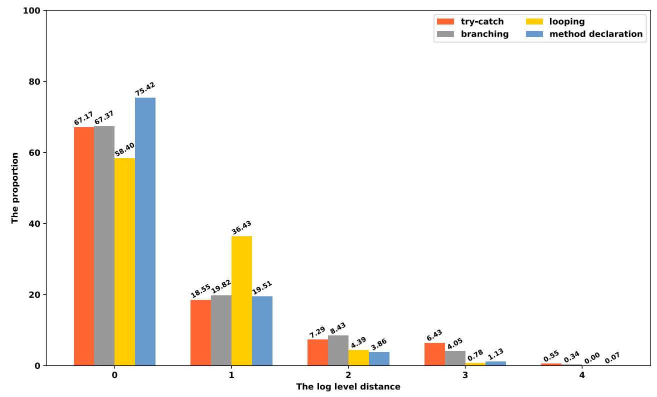
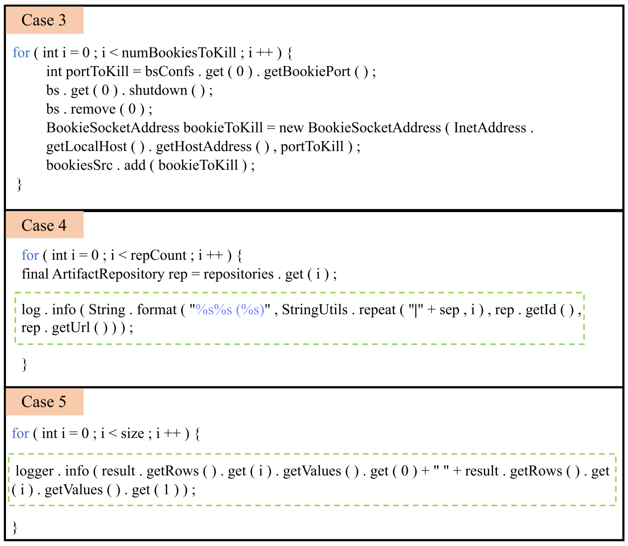


**Fig. 11.** Log level distance in the predictions generated by ELogger. Zero indicates predictions having a correct level.

**Table 4**
BLEU-4 of log message generated by ELogger.

| Type | Try-catch (%) | Branching (%) | Looping (%) | Method declaration (%) |
|---|---|---|---|---|
| BLEU-4 | 33.76 | 22.33 | 20.10 | 30.08 |


Case 3

```
for ( int i = 0 ; i < numBookiesToKill ; i ++ ) {
        int portToKill = bsConfs . get ( 0 ) . getBookiePort ( ) ;
        bs . get ( 0 ) . shutdown ( ) ;
        bs . remove ( 0 ) ;
        BookieSocketAddress bookieToKill = new BookieSocketAddress ( InetAddress .
        getLocalHost ( ) . getHostAddress ( ) , portToKill ) ;
        bookiesSrc . add ( bookieToKill ) ;
}
```

Case 4

```
for ( int i = 0 ; i < repCount ; i ++ ) {
final ArtifactRepository rep = repositories . get ( i ) ;

log . info ( String . format ( "%s%s (%s)" , StringUtils . repeat ( "|" + sep , i ) , rep . getId ( ) ,
rep . getUrl ( ) ) ) ;

}
```

Case 5

```
for ( int i = 0 ; i < size ; i ++ ) {

logger . info ( result . getRows ( ) . get ( i ) . getValues ( ) . get ( 0 ) + " " + result . getRows ( ) . get
( i ) . getValues ( ) . get ( 1 ) ) ;

}
```

**Fig. 12.** The cases of looping block.

try-catch blocks are only off by one level from the target (19.51%, 19.82%, and 18.55%, respectively), with differences of greater than three being rare (1.20%, 4.39%, and 6.98%, respectively). Conversely, the level prediction accuracy for looping blocks (58.40%) is the lowest, and the majority of errors (36.43%) are only off by one level. The majority of prediction errors occur when the predicted level is only one level away from the target level, and errors with differences of more than three levels are infrequent.

Table 4 shows the BLEU-4 of the generated log message. It is evident that the BLEU-4 is best for try-catch blocks (33.76%), followed by method declaration blocks (30.08%). The BLEU-4 for looping blocks (20.10%) is the lowest. Therefore, we can conclude that ELogger's message generation performs better for try-catch than for other block types.

Based on the evaluation results presented above, it is evident that ELogger performed the best on try-catch blocks and the worst on looping blocks. Developers may prioritize reviewing the logging statement in try-catch blocks. To investigate the reason for the poor performance of ELogger in looping blocks, we manually examined the looping blocks. Our analysis revealed that many of the looping blocks used logging statements to record variable states. The use of logging statements to record variable state is a matter of personal preference or coding style, and some developers may prefer it as a way of keeping the code organized and easy to read. Conversely, others may prefer to use other methods such as debugging tools to track variable values. Fig. 12 displays three looping block cases, where Case 4 and Case 5 log variable states, while Case 3 does not. However, we did not find a strong basis for why some developers choose to log variable states in looping blocks while others do not. This indicates that there is no one-size-fits-all guideline for logging practices. Developers should exercise caution when reviewing logging statements in looping blocks to avoid excessive entries that could impact model performance. Additionally, future research may consider adopting more stringent data selection criteria and exploring additional sources of information to enhance end-to-end log statement generation for looping blocks.

**Summary for RQ2** 👉 *The varied performance of ELogger across different block types could be attributed to the complexity and type of information required to be logged in each block. Specifically, ELogger performed the best on try-catch blocks and the worst on looping blocks.*

### *4.3. RQ3: The effectiveness of the main components in ELogger*

**Motivation**. The two main components of ELogger are the whether-to-log component and the www-log. In order to gain a better understanding of how each component contributes to the effectiveness of ELogger in generating end-to-end complete log statements, we perform an ablation experiment to evaluate the performance of each component individually.

**Method**. We compared ELogger's two main components with existing methods. Specifically, we compared the whether-to-log component with the Li-Model, and the www-log component with LANCE to evaluate the effectiveness of each individual component of ELogger. The Li-Model predicts whether a logging statement should be added to a given code block based on syntactic features, and Li et al. demonstrated that Li-Model, trained using these features, achieved the best results across all evaluation metrics (Li et al., 2020a). For evaluating the effectiveness of the www-log component, we only compared ELogger and LANCE on code blocks requiring logging in the test dataset since LANCE is designed specifically to generate complete log statements for Java methods identified as requiring logging. This evaluation approach allowed us to focus on the quality of the generated log statements, as neither method can suggest whether to log.

**Results.** As the results are shown in Tables 5 and 6, we make the following observations:

Table 5 presents the evaluation results of the whether-to-log component and Li-Model. BA, P, R, and F1 represent Balanced Accuracy, Precision, Recall, and F1 score, respectively. The results indicate that the whether-to-log component outperforms Li-Model, as evidenced by its higher scores across all evaluation metrics. Specifically, the whether-to-log component achieves a 5.04% improvement in balanced accuracy, a 19.11% improvement in precision, a 2.51% improvement in recall, and a 10.74% improvement in F1 score. These results suggest that the whether-to-log component is more effective in predicting whether a logging statement should be added to a code block than Li-Model.

In Table 6, we present the effectiveness of the www-log component and LANCE in complete log statement generation. The first row shows that the www-log component correctly predicted the position in 73.55% of cases. The second and third rows show that the www-log component correctly predicted two out of three sub-tasks (Position, and Level) in 49.61% of cases, and another two out of three sub-tasks (Position, and Message) in 24.34% of cases. Moreover, in 22.30% of cases, the www-log component accurately predicted all three sub-tasks

**Table 5**
The effectiveness of whether-to-log component.

| Method | BA (%) | P (%) | R (%) | F1 (%) |
|---|---|---|---|---|
| Li-Model | 77.80 (↑5.04%) | 61.18 (↑19.11%) | 69.92 (↑2.51%) | 65.26 (↑10.74%) |
| Whether-to-log | 81.72 | 72.87 | 71.68 | 72.27 |

**Table 6**
The effectiveness of www-log component.

| | Logging practices sub-tasks | | | LANCE | Www-log |
|---|---|---|---|---|---|
| | Position | Level | Message | Pred (%) | Pred (%) |
| 1 out of 3 | ✓ | – | – | 61.63 (↑19.34%) | 73.55 |
| 2 out of 3 | ✓ | ✓ | – | 39.34 (↑26.11%) | 49.61 |
| | ✓ | – | ✓ | 16.20 (↑50.25%) | 24.34 |
| | ✓ | ✓ | ✗ | 24.35 (↑12.16%) | 27.31 |
| | ✓ | ✗ | ✓ | 1.21 (↑68.60%) | 2.04 |
| All | ✓ | ✓ | ✓ | 14.99 (↑48.77%) | 22.30 |

(Position, Level, and Message). In contrast, LANCE achieved 61.63%, 39.34%, 16.20%, and 14.99%, respectively, which is lower than that of the www-log component. These findings suggest that the www-log component is more effective than LANCE in complete log statement generation.

**Summary for RQ3** 👉 *Both the whether-to-log and www-log components make significant contributions to the overall performance of ELogger.*

## 5. Related work

### 5.1. Improving logging practices

**Whether to log**. The term "Whether to log" refers to whether logging should occur in a code snippet. In order to improve the accuracy of automated log placement, Zhu et al. proposed LogAdvisor, a logging tool that provides recommendations on whether to insert a log statement into a code snippet (Zhu et al., 2015). They demonstrated the feasibility and effectiveness of LogAdvisor by evaluating it on two Microsoft industrial systems and two open-source systems. Jia et al. proposed SmartLog, an intention-aware log automation tool that mines log placement rules from existing logs (Jia et al., 2018). The experimental results of SmartLog on six real-world open-source projects demonstrate its effectiveness. Li et al. used topic models to automatically extract the topics of a code snippet that can describe its functionality (Li et al., 2018). They then employed these topics to determine the likelihood of a code snippet being logged. Li et al. proposed a deep learning framework that suggests whether to log at the code block level (Li et al., 2020a). The framework achieved promising results in the evaluation, with an average balanced accuracy of 80.1% for within-project and 67.3% for cross-project scenarios.

**Where to log**. The term "where to log" refers to the specific location where logging statements should be placed. Liu et al. proposed a deep learning-based approach to recommend variables that should be logged in software applications (Liu et al., 2019b). The model was trained and evaluated on 9 high-quality and large Java projects, demonstrating better performance compared to several baseline methods.

**Which log level**. The term "which log level" refers to which level should be used by log statements. Li et al. developed DeepLV, a deep-learning-based method that automatically recommends log levels for logging statements by extracting syntactic and semantic features from source code (Li et al., 2021). The evaluation results of DeepLV on nine open-source projects showed that it is effective in recommending log levels both within-project and cross-project.

**What information to log**. The term "what information to log" refers to the description that should be used to record information in log statements. He et al. conducted an empirical study to investigate the usage of natural language descriptions in logging statements (He et al., 2018). Based on their findings, they proposed an automatic log description generation method that reuses descriptions from logging statements in similar code snippets. Ding et al. proposed LoGenText, an automatic log description generation method that employs neural machine translation models and contextual information from the preceding source code (Ding et al., 2022).

Additionally, Mastropaolo et al. proposed LANCE to automate the process of generating complete log statements for Java methods, which is based on the T5 model (Mastropaolo et al., 2022). Although LANCE can generate complete logging statements and insert them at the right code location, it needs to be given an input method that necessitates logging statements. This indicates it does not address the problem of whether to log.

ELogger distinguishes itself from the aforementioned techniques by generating end-to-end log statements, offering comprehensive support to developers. This encompasses the capacity to complete all four essential sub-tasks (whether to log, where to log, which log level, and what information to log) of logging practices in a single unified process. Developed concurrently with ELogger, LEONID enhances the capabilities established by LANCE, by completing the same four sub-tasks and enabling the insertion of multiple log statements as needed (Mastropaolo et al., 2024). Unlike LEONID, which targets entire Java methods and employs distinct feature representations for different stages of its logging process, ELogger provides granular logging by focusing on individual Java code blocks and using a consistent feature representation. Additionally, ELogger offers adaptable model options, enhancing its flexibility.

### 5.2. Empirical studies on logging practices

Yuan et al. conducted the first empirical study to investigate the practice of logging on four large C/C++ open-source projects (Yuan et al., 2012). Their findings demonstrated the pervasiveness and benefit of logging in software development. Chen et al. extended the study of Yuan et al. to 21 different Java-based open-source projects and confirmed the prevalence of logging in most of the studied projects, as well as the active maintenance of logging code (Chen et al., 2017). Similarly, Zeng et al. extended the study by Yuan et al. to 1444 open-source Android apps and found that while logging is also widespread in Android apps, it was less pervasive and less actively maintained than logging in server and desktop applications (Zeng et al., 2019).

Kabinna et al. conducted a study on the stability of logging statements in four open-source projects and reported that 20% to 45% of logging statements change throughout the software's lifetime (Kabinna et al., 2018). Similarly, Zhang et al. investigated log evolution in a real-world Microsoft system and confirmed the instability of log statements during the software's lifetime (Zhang et al., 2019).

Fu et al. conducted a study on logging practices in industry by analyzing two large systems from Microsoft, with a focus on identifying where developers log (Fu et al., 2014). Their study yielded six findings, and they developed a tool that utilizes these findings to automatically predict where to log for two types of code blocks. Zhi et al. conducted

**Table 7**
The effectiveness of ELogger based on different pre-trained model.

| | Log generation sub-task | | | | $\text{ELogger}_{uxb}$ | $\text{ELogger}_{c5p}$ | ELogger |
|---|---|---|---|---|---|---|---|
| | Whether | Position | Level | Message | Preds(%) | Preds(%) | Preds(%) |
| 1 out of 4 | ✓ | – | – | – | 68.75 | 69.96 | 71.68 |
| 2 out of 4 | ✓ | ✓ | – | – | 55.64 | 57.58 | 58.29 |
| 3 out of 4 | ✓ | ✓ | ✓ | – | 39.63 | 42.84 | 41.97 |
| | ✓ | ✓ | – | ✓ | 19.63 | 22.39 | 21.18 |
| | ✓ | ✓ | ✓ | ✗ | 21.53 | 21.92 | 22.42 |
| | ✓ | ✓ | ✗ | ✓ | 1.53 | 1.48 | 1.62 |
| All | ✓ | ✓ | ✓ | ✓ | 18.10 | 20.92 | 19.55 |
| p-value | | | | | <0.05 | >0.05 | |
| Cliff's Delta | | | | | 0.143 (N) | 0.020 (N) | |

The variant of ELogger based on UniXcoder is denoted as $\text{ELogger}_{uxb}$, while the variant based on CodeT5+ based is denoted as $\text{ELogger}_{c5p}$.

an empirical study on the usage and evolution of logging configurations in 10 open-source and 10 industrial Java projects, which resulted in 10 findings about logging configuration practices (Zhi et al., 2019). Additionally, Li et al. conducted a qualitative study, revealing that developers consider a wide range of logging benefits and costs during development (Li et al., 2020b).

In this paper, our goal is to propose an innovative approach for end-to-end log statement generation, distinguishing ourselves from the aforementioned empirical studies on logging practices. To the best of our knowledge, we are the first to address the end-to-end log statement generation problem.

## 6. Discussion

### 6.1. The impact of different pre-trained model

We evaluate three mainstream pre-trained models, including: GraphCodeBERT (model size: 125M), UniXcoder (model size: 110M), and CodeT5+ (model size: 220M). We select these pre-trained models primarily for two reasons: first, they have widespread recognition as large language models for code-related tasks, and second, their model sizes are comparable. We denote the UniXcoder-based variant as $\text{ELogger}_{uxb}$ and the CodeT5+ based variant as $\text{ELogger}_{c5p}$. Both $\text{ELogger}_{uxb}$ and $\text{ELogger}_{c5p}$ share identical experimental settings with the ELogger. Table 7 presents the effectiveness of ELogger when using different pre-trained models. The last two rows show the results of the Wilcoxon test and Cliff's delta. The results indicate that there is a significant difference in performance between the ELogger and $\text{ELogger}_{uxb}$ (p-value $< 0.05$). The Cliff's delta between the ELogger and $\text{ELogger}_{uxb}$ is 0.143(N) close to 0.147(S). This indicates that pre-trained models, even those of similar size and identical architecture, still have some impact on our method. There is no significant difference between the performance of $\text{ELogger}_{c5p}$ and the ELogger (p-value $>$ 0.05, and Cliff's delta is 0.020 (N)). Upon further analyzing the experimental results, we found that GraphCodeBERT outperforms CodeT5+ in classification tasks, whereas CodeT5+ excels in text generation tasks compared to GraphCodeBERT. This difference in performance could be attributed to the inherent model architectures. The Encoder-Only model of GraphCodeBERT, which provides a thorough understanding of context, is better suited for classification tasks. On the other hand, the Encoder-Decoder model of CodeT5+, with its dual capabilities of encoding and decoding, is better equipped for text generation tasks.

We would like to emphasize that while GraphCodeBERT is our pre-trained model for ELogger, our method is not inherently restricted to this pre-trained model. Developers have the flexibility to select a pre-trained model that aligns with their application's specific requirements. For example, if the priority is on classification tasks and model size, GraphCodeBERT would be an optimal choice. On the other hand, for a focus on improving text generation tasks, CodeT5+ would be more suitable. Additionally, as more advanced pre-trained models become available, developers can modify ELogger to incorporate these new models, potentially boosting performance.

### 6.2. Validity

**Construct Validity.** The studies of "learning to log" are based on the assumption that the training data have high logging quality. This means that the model constructed from the data can represent common and good logging practices and can generalize well to predict new instances. However, there is no standard definition of what constitutes high-quality or optimal logging practices. In our study, we make the assumption that the dataset we used contains good logging implementations because it was carefully selected based on specific criteria. Mastropaolo et al. selected Java projects that met certain requirements (Mastropaolo et al., 2022), such as having at least 500 commits, 10 contributors, 10 stars, and not being forks, in order to construct their dataset, which was then used in our study.

**Internal Validity.** To ensure the reliability of our results, we followed advanced practices outlined in prior studies (Mastropaolo et al., 2022; Guo et al., 2020) to set the hyper-parameters for our approach, as various hyper-parameters used in neural networks can affect the effectiveness of the trained models. We also utilized the base version of GraphCodeBERT as the pre-trained model to obtain the feature embedding. However, to explore the performance of generating end-to-end complete log statements further, future studies may investigate other feature representation approaches.

**External Validity.** We utilized the latest dataset in logging research (Mastropaolo et al., 2022), which is consists of projects written in Java. While this limits the diversity of the dataset and may affect the generalizability of our results to other programming languages, the dataset used in our experiments is sufficiently large and robust to demonstrate the effectiveness of ELogger. However, evaluating ELogger on projects written in other languages or industrial projects can further showcase its effectiveness and limitations.

## 7. Conclusion

Our paper proposes ELogger, an innovative approach for end-to-end log statement generation at block-level. To reduce the complexity of integrating distinct tasks and the workflow of the method, both components of ELogger, namely whether-to-log and www-log, utilize a consistent feature representation method. Our experimental results demonstrate that ELogger achieves promising performance in end-to-end log statement generation at block-level. Specifically, ELogger can correctly predict all four sub-tasks (Whether, Position, Level, and Message) in 19.55% of cases. When compared to the baselines that combined existing approaches to provide end-to-end log statement generation support, ELogger demonstrates a significant improvement, achieving an average increase of 50.85% to 78.21% in end-to-end log statement generation. Moreover, ELogger can predict whether to log in 71.68% of cases, two of the four sub-tasks (Whether and Position) in 58.29% of cases, three of the four sub-tasks (Whether, Position, and Level) in 41.97% of cases, all of which outperform the baselines.

## CRediT authorship contribution statement

**Ying Fu:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology. **Meng Yan:** Writing – review & editing, Supervision, Project administration. **Pinjia He:** Writing – review & editing. **Chao Liu:** Writing – review & editing. **Xiaohong Zhang:** Writing – review & editing, Supervision. **Dan Yang:** Supervision, Project administration.

## Data availability

https://github.com/cqu-isse/ELogger.

## Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (No. 62372071 and 62202074), the Chongqing Technology Innovation and Application Development Project (No. CSTB2022TIAD-STX0007 and No. CSTB2023TIAD-STX0025), and the Fundamental Research Funds for the Central Universities (No. 2023CDJKYJH013).

## Declaration of competing interest

The authors have no conflict of interest.

## References

Babenko, A., Mariani, L., Pastore, F., 2009. Ava: Automated interpretation of dynamically detected anomalies. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. pp. 237–248.
Bahdanau, D., Cho, K., Bengio, Y., 2014. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473.
Banerjee, S., Lavie, A., 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In: Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization. pp. 65–72.
Berrocal, E., Yu, L., Wallace, S., Papka, M.E., Lan, Z., 2014. Exploring void search for fault detection on extreme scale systems. In: 2014 IEEE International Conference on Cluster Computing. CLUSTER, IEEE, pp. 1–9.
Breier, J., Branišová, J., 2015. Anomaly detection from log files using data mining techniques. In: Information Science and Applications. Springer, pp. 449–457.
Chen, A.R., 2019. An empirical study on leveraging logs for debugging production failures. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings. ICSE-C, IEEE, pp. 126–128.
Chen, B., et al., 2017. Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation. Empir. Softw. Eng. 22 (1), 330–374.
Cliff, N., 2014. Ordinal Methods for Behavioral Data Analysis. Psychology Press.
Das, A., Mueller, F., Rountree, B., 2020. Aarohi: Making real-time node failure prediction feasible. In: 2020 IEEE International Parallel and Distributed Processing Symposium. IPDPS, IEEE, pp. 1092–1101.
Das, A., Mueller, F., Siegel, C., Vishnu, A., 2018. Desh: deep learning for system health prediction of lead times to failure in hpc. In: Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing. pp. 40–51.
Ding, Z., Li, H., Shang, W., 2022. Logentext: automatically generating logging texts using neural machine translation. In: SANER. IEEE.
Fu, Y., Liang, K., Xu, J., 2023. MLog: Mogrifier LSTM-based log anomaly detection approach using semantic representation. IEEE Trans. Serv. Comput..
Fu, Q., Zhu, J., Hu, W., Lou, J.-G., Ding, R., Lin, Q., Zhang, D., Xie, T., 2014. Where do developers log? an empirical study on logging practices in industry. In: Companion Proceedings of the 36th International Conference on Software Engineering. ICSE, pp. 24–33.
Guan, W., Cao, J., Gu, Y., Qian, S., 2023. GRASPED: A GRU-AE network based multi-perspective business process anomaly detection model. IEEE Trans. Serv. Comput..
Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., Yin, J., 2022. Unixcoder: Unified cross-modal pre-training for code representation. arXiv preprint arXiv:2203.03850.
Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al., 2020. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366.
He, P., Chen, Z., He, S., Lyu, M.R., 2018. Characterizing the natural language descriptions in software logging statements. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 178–189.
Jia, T., Chen, P., Yang, L., Li, Y., Meng, F., Xu, J., 2017. An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services. In: 2017 IEEE International Conference on Web Services. ICWS, IEEE, pp. 25–32.
Jia, Z., Li, S., Liu, X., Liao, X., Liu, Y., 2018. SMARTLOG: Place error log statement by deep understanding of log intention. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 61–71.
Kabinna, S., Bezemer, C.-P., Shang, W., Syer, M.D., Hassan, A.E., 2018. Examining the stability of logging statements. Empir. Softw. Eng. 23 (1), 290–333.
Le, V.-H., Zhang, H., 2021. Log-based anomaly detection without log parsing. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 492–504.
Lee, C., Yang, T., Chen, Z., Su, Y., Lyu, M.R., 2023. Eadro: An end-to-end troubleshooting framework for microservices on multi-source data. In: Proceedings of the 45th International Conference on Software Engineering. ICSE.
Li, Z., Chen, T.-H., Shang, W., 2020a. Where shall we log? Studying and suggesting logging locations in code blocks. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. ASE, pp. 361–372.
Li, H., Chen, T.-H.P., Shang, W., Hassan, A.E., 2018. Studying software logging using topic models. Empir. Softw. Eng. 23 (5), 2655–2694.
Li, Z., Li, H., Chen, T.-H., Shang, W., 2021. Deeplv: Suggesting log levels using ordinal based neural networks. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. ICSE, IEEE, pp. 1461–1472.
Li, H., Shang, W., Adams, B., Sayagh, M., Hassan, A.E., 2020b. A qualitative study of the benefits and costs of logging from developers' perspectives. IEEE Trans. Softw. Eng..
Li, H., Shang, W., Zou, Y., E. Hassan, A., 2017. Towards just-in-time suggestions for log changes. Empir. Softw. Eng. 22, 1831–1865.
Lin, C.Y., 2004. ROUGE: A package for automatic evaluation of summaries. In: Proceedings of the Workshop on Text Summarization Branches Out. was 2004.
Lin, T.-Y., Goyal, P., Girshick, R., He, K., Dollár, P., 2017. Focal loss for dense object detection. In: Proceedings of the IEEE International Conference on Computer Vision. pp. 2980–2988.
Lin, Q., Hsieh, K., Dang, Y., Zhang, H., Sui, K., Xu, Y., Lou, J.-G., Li, C., Wu, Y., Yao, R., et al., 2018. Predicting node failure in cloud service systems. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE, pp. 480–490.
Liu, F., Wen, Y., Zhang, D., Jiang, X., Xing, X., Meng, D., 2019a. Log2vec: a heterogeneous graph embedding based approach for detecting cyber threats within enterprise. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1777–1794.
Liu, Z., Xia, X., Lo, D., Xing, Z., Hassan, A.E., Li, S., 2019b. Which variables should i log? IEEE Trans. Softw. Eng. 47 (9), 2012–2031.
Lu, J., Li, F., Li, L., Feng, X., 2018. Cloudraid: hunting concurrency bugs in the cloud via log-mining. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 3–14.
Mastropaolo, A., Ferrari, V., Pascarella, L., Bavota, G., 2024. Log statements generation via deep learning: Widening the support provided to developers. J. Syst. Softw. 210, 111947.
Mastropaolo, A., Pascarella, L., Bavota, G., 2022. Using deep learning to generate complete log statements. In: Proceedings of the 44th International Conference on Software Engineering. ICSE, pp. 2279–2290.
Nagappan, M., Wu, K., Vouk, M.A., 2009. Efficiently extracting operational profiles from execution logs using suffix arrays. In: 2009 20th International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 41–50.
Nagaraj, K., Killian, C.E., Neville, J., 2012. Structured comparative analysis of systems logs to diagnose performance problems.. In: NSDI. p. 353, no. 1.
Nandi, A., Mandal, A., Atreja, S., Dasgupta, G.B., Bhattacharya, S., 2016. Anomaly detection using program control flow graph mining from execution logs. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 215–224.
Papineni, K., Roukos, S., Ward, T., Zhu, W.-J., 2002. BLEU: a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics. Association for Computational Linguistics, pp. 311–318.
Shang, W., Jiang, Z.M., Adams, B., Hassan, A.E., Godfrey, M.W., Nasser, M., Flora, P., 2014. An exploratory study of the evolution of communicated information about the execution of large software systems. J. Softw.: Evol. Process 26 (1), 3–26.

Wang, Y., Le, H., Gotmare, A.D., Bui, N.D., Li, J., Hoi, S.C., 2023. Codet5+: Open code large language models for code understanding and generation. arXiv preprint arXiv:2305.07922.

Wilcoxon, F., 1992. Individual comparisons by ranking methods. In: Breakthroughs in Statistics: Methodology and Distribution. Springer, pp. 196–202.

Xia, B., Bai, Y., Yin, J., Li, Y., Xu, J., 2020. LogGAN: a log-level generative adversarial network for anomaly detection using permutation event modeling. Inf. Syst. Front. 1–14.

You, Y., Li, J., Reddi, S., Hseu, J., Kumar, S., Bhojanapalli, S., Song, X., Demmel, J., Keutzer, K., Hsieh, C.-J., 2019. Large batch optimization for deep learning: Training bert in 76 minutes. arXiv preprint arXiv:1904.00962.

Yu, G., Chen, P., Li, Y., Chen, H., Li, X., Zheng, Z., 2023. Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE.

Yuan, D., Park, S., Zhou, Y., 2012. Characterizing logging practices in open-source software. In: 2012 34th International Conference on Software Engineering. ICSE, IEEE, pp. 102–112.

Zeng, Y., Chen, J., Shang, W., Chen, T.-H.P., 2019. Studying the characteristics of logging practices in mobile apps: a case study on f-droid. Empir. Softw. Eng. 24 (6), 3394–3434.

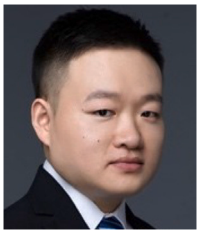
Zhang, C., Peng, X., Sha, C., Zhang, K., Fu, Z., Wu, X., Lin, Q., Zhang, D., 2022. DeepTraLog: Trace-log combined microservice anomaly detection through graph-based deep learning. In: Proceedings of the 44th International Conference on Software Engineering. ICSE, pp. 623–634.

Zhang, X., Xu, Y., Lin, Q., Qiao, B., Zhang, H., Dang, Y., Xie, C., Yang, X., Cheng, Q., Li, Z., et al., 2019. Robust log-based anomaly detection on unstable log data. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE, pp. 807–817.

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

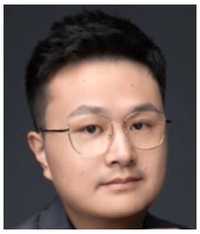
Zhu, J., He, P., Fu, Q., Zhang, H., Lyu, M.R., Zhang, D., 2015. Learning to log: Helping developers make informed logging decisions. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Vol. 1, IEEE, pp. 415–425.

**Ying Fu** received the B.S. and M.S. degrees from Chongqing University, China, where she is currently pursuing the Ph.D. degree at the School of Big Data & Software Engineering. Her current research focuses on data mining and analyzing.



**Meng Yan** is now a Research Professor at the School of Big Data & Software Engineering, Chongqing University, China. He received Ph.D. degree in June 2017 under the supervision of Prof. Xiaohong Zhang from Chongqing University, China. His current research focuses on how to improve developers' productivity, how to improve software quality, and how to reduce the effort during software development by analyzing rich software repository data.

**Pinjia He** received the BEng degree in computer science from the South China University of Technology, Guangzhou, China in 2013 and the Ph.D. degree in computer science from The Chinese University of Hong Kong, Hong Kong, China in 2018. He worked as a postdoctoral scholar at the Department of Computer Science at ETH Zurich from 2018 to 2021. He is currently an assistant professor in the School of Data Science, The Chinese University of Hong Kong, Shenzhen. His research interests include software reliability engineering, AIOps, software testing, and natural language processing. He received the most influential paper award from ISSRE. He received the first IEEE Open Software Services Award.



**Chao Liu** received a Ph.D. in software engineering from Chongqing University, China, in 2018. He was a Post-Doctoral Research Fellow at Zhejiang University, China, from 2019 to 2021. He is currently an Associate Professor at Chongqing University, China. His research interests include intelligent software engineering, program analysis, and software reuse.

**Xiaohong Zhang** is currently a Professor and the Vice Dean of the School of Big Data and Software Engineering, Chongqing University. He received the M.S. degree in applied mathematics and the Ph.D. degree in computer software and theory from Chongqing University, China, in 2006. His current research interests include data mining of software engineering, topic modeling, image semantic analysis, and video analysis.

**Dan Yang** is currently the President of Southwest Jiaotong University. He is also a Professor with the School of Big Data and Software Engineering, Chongqing University. He received the B.S. degree in automation, the M.S. degree in applied mathematics, and the Ph.D. degree in machinery manufacturing and automation from Chongqing University, Chongqing. From 1997 to 1999, he held a postdoctoral position at the University of ElectroCommunications, Tokyo, Japan. His research interests include computer vision, image processing, pattern recognition, software engineering, and scientific computing.