

Received 2 December 2025, accepted 14 December 2025, date of publication 17 December 2025,
date of current version 23 December 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3645237

RESEARCH ARTICLE

Automated Logging With Large Language Models: Eliminating Task-Specific Pretraining

DAE-SUNG WANG¹, TAE-IL KIM¹, CHAN-GUN LEE¹, AND HYUN-TAEK HONG^{1,2}

¹Department of Computer Science and Engineering, Chung-Ang University, Seoul 06974, South Korea

²Vehicle Solution Company, LG Electronics, Seoul 07796, South Korea

Corresponding author: Hyun-Taek Hong (elec130@cau.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) Grant funded by the Korea Government Ministry of Science and ICT (MSIT) under Grant RS-2024-00342059, and in part by the LG Electronics.

ABSTRACT Logging is a fundamental practice in software engineering, supporting monitoring, debugging, and system comprehension. However, determining where to insert log statements, what information to record, and how to phrase messages remains a labor-intensive and error-prone task. Recent research has explored automated logging; however, most approaches rely on domain- or language-specific pretraining, limiting applicability across diverse projects and environments. This paper investigates whether large language models (LLMs) trained on general code and natural-language corpora can eliminate the need for task-specific pretraining in automated logging. In our setting, models do not perform any Java- or task-specific pretraining but are trained only through lightweight supervised fine-tuning on the logging dataset. We compare LEONID, a state-of-the-art pretraining-centric logging system, and off-the-shelf models, including T5, CodeT5, and Qwen, without additional Java-specific pretraining. This evaluation encompasses single-log injection (levels, variables, positions, and messages), log-message quality assessed using reference-based metrics, and multilog injection, requiring multiple statements in a method. The results indicate that general-purpose LLMs rival or surpass LEONID in several aspects. Notably, Qwen-7B improves insertion position accuracy by 14 percentage points, addressing one of the most critical factors for the practical deployment of automated logging. Moreover, CodeT5 and Qwen achieve competitive or superior performance in message generation, and Qwen-7B demonstrates stronger structural reasoning for insertion positions and multilog scenarios. Although LEONID remains more stable in level selection and variable binding, the findings indicate that task-specific pretraining is not universally required. This work provides the first head-to-head evaluation of a pretraining-based approach versus off-the-shelf LLM approaches for automated logging, offering practical guidance for when to apply custom pretraining and when general-purpose models are sufficient.

INDEX TERMS Automated logging, code intelligence, large language model, software engineering automation.

I. INTRODUCTION

Logging is a crucial practice in software engineering that enables the monitoring, debugging, and comprehension of complex systems [23]. Well-placed and carefully written log statements allow developers to identify performance bottlenecks, investigate failures, and ensure system reliability [4].

The associate editor coordinating the review of this manuscript and approving it for publication was Xin Sun¹.

In contrast, misplaced or low-quality logs hinder root-cause analyses, increase operational costs, and may introduce cybersecurity and privacy risks [7]. Despite well-established best practices, determining where to insert logs, what information to record, and which log level to use remains challenging and laborious.

An increasing body of research [5], [14], [15], [19], [26] has explored automated logging techniques to address this challenge. Early efforts have focused on specific

subproblems, such as recommending log levels or predicting insertion points. More recent approaches have extended this scope to end-to-end log statement generation, addressing multiple elements (e.g., position, level, variable, and message) [8], [17], [18]. Collectively, these studies have demonstrated the feasibility of deep or machine learning-based automation, providing developers with valuable support beyond manual practices.

However, previous state-of-the-art (SOTA) methods [17], [18] share a critical limitation: they rely on pretraining tailored to specific software repositories or programming languages. For example, LEONID [17], which represents a SOTA method in automated log injection, requires domain-specific pretraining on Java projects to achieve competitive accuracy. This dependency restricts generalizability, as models trained on one repository or language often perform poorly when applied to other projects [11]. Hence, the industrial applicability of these methods remains limited, and deploying them in practice would require costly and repeated pretraining for every new codebase.

This paper addresses this limitation by investigating whether large language models (LLMs) can eliminate the need for task-specific pretraining. Unlike prior systems, LLMs, including T5 [22], CodeT5 [25], and Qwen [2], are pretrained on massive multilingual corpora that include diverse programming languages and natural-language text. These models are readily available without additional pretraining for a repository or programming language, which prompts the central research question of this paper: To what extent can general-purpose code LLMs, without domain-specific pretraining, achieve competitive or superior performance in automated logging tasks?

To answer this question, we conduct a systematic empirical study comparing SOTA pretraining-centric systems with general-purpose code LLMs on core logging tasks: single-log injection (position, level, variable, and message), log-message quality, and multilog injection. The results reveal that general-purpose models perform comparably to or better than specialized pretrained models while eliminating the need for project-specific pretraining. This finding significantly broadens the applicability of automated logging techniques to diverse repositories and industrial contexts.

This paper makes the following contributions:

- Systematic evaluation of logging is performed without domain-specific pretraining. This work compares pretrained systems (i.e., LEONID) with general-purpose code LLMs under identical conditions.
- This work demonstrates the effectiveness of code LLMs, revealing that they can match or outperform pretraining-based systems across multiple logging tasks, challenging the assumption that task-specific pretraining is indispensable.
- This paper provides empirical evidence of the general applicability of code LLMs, enabling deployment across

diverse repositories and environments without repeated pretraining.

The remainder of this paper is organized as follows. Section II reviews prior work on automated logging and machine learning for code. Next, Section III describes the experimental design and models. Then, Section IV presents the results and analysis. Finally, Section V discusses the implications for practice and research, and Section VI concludes the paper.

II. RELATED WORK

This section reviews prior studies on automatic log generation and the baseline methods upon which we built our proposed approach.

A. AUTOMATED LOG GENERATION

Automated log generation has been studied from multiple perspectives, reflecting the importance of logging for program comprehension, debugging, and maintainability [4], [23]. Early research [5], [14], [15], [19], [26] has relied on heuristics or lightweight statistical models to identify potential log statements. Although these methods provide initial insight, they are limited in capturing the semantic and structural context of the source code.

Later efforts introduced machine learning techniques to enhance automation. Several studies have addressed specific subproblems, including predicting whether a given location requires a log statement, recommending appropriate log levels, or generating natural-language log messages. These approaches demonstrated that data-driven methods could outperform static rules and offer practical support for developers.

More recently, end-to-end approaches have emerged, aiming to generate complete log statements, including position, level, variables, and messages [8], [17], [18]. This trend represents a significant step toward realistic automation because it mirrors the multifaceted decisions developers face when writing logs. However, as summarized in Table 1, most of these studies remain constrained in two respects. First, they are often limited to single-log injections and address only a subset of logging elements. Second, many systems rely on pretraining that is tightly coupled with specific repositories or programming languages, hindering their applicability to diverse projects.

This dependency on task- and domain-specific pretraining raises concerns regarding their generalizability and industrial adoption. A system that must be retrained for every new repository or language incurs significant overhead, making widespread deployment challenging. This work revisits automated log generation under this changing landscape by examining whether large, general-purpose code models can deliver comparable or superior performance without tailored pretraining.

In an early attempt, Yao et al. [26] introduced Log4Perf, a statistical approach that identifies performance-critical code regions and recommends them for logging locations. This

TABLE 1. Summary of methods and targets related to this study.

Author	Method	Pretrain	Level	Position	Message	Single	Multiple
Yao <i>et al.</i> (2018)	Statistical	No need	✗	✓	✗	✓	✗
Mizouchi <i>et al.</i> (2019)	Dynamic Adapt	No need	✓	✗	✗	✓	✓
Li <i>et al.</i> (2020)	RNN (Seq2Seq)	Need	✗	✓	✗	✓	✗
Li <i>et al.</i> (2021)	Bi-LSTM	Need	✓	✗	✗	✓	✗
Ding <i>et al.</i> (2022)	Transformer	Need	✗	✗	✓	✓	✗
Mastropaolo <i>et al.</i> (2022)	T5	Need	✓	✓	✓	✓	✗
Mastropaolo <i>et al.</i> (2024)	T5	Need	✓	✓	✓	✓	✓
Ours	T5/CodeT5/Qwen	No need	✓	✓	✓	✓	✓

study was among the first to connect log placement with performance monitoring, demonstrating improved support for web-based systems. However, the method was tailored to this context and lacks general applicability to broader logging tasks.

Building on runtime adaptation, Mizouchi *et al.* [19] developed PADLA, a dynamic log-level adapter that uses online phase detection to adjust logging verbosity at runtime. Their approach introduced the novel idea of adapting log levels dynamically, reducing the cost of continuous detailed logging while preserving useful diagnostic information. However, its effectiveness depends on anomalies that manifest as measurable performance deviations.

Li *et al.* [14] applied a sequence-to-sequence model based on the recurrent neural network to generate automatic log messages from source code. This study was the first to employ neural networks for log text synthesis, improving message fluency and consistency compared to rule-based baselines. Nevertheless, the model captured surface-level patterns without incorporating deeper semantic information.

Extending their work, Li *et al.* [15] proposed DeepLV, which treats log-level recommendation as an ordinal classification task. By recognizing the ordinal nature of log levels, the approach presented a more principled method to predict severity than prior heuristic methods. This method achieved higher accuracy than the baselines; however, it relies on training data from specific Java projects and struggles with cross-project generalization.

Advancing automated message generation, Ding *et al.* [5] presented LoGenText, a transformer-based neural machine translation system that generates logging texts from code and its context. Their method achieved strong empirical results, significantly outperforming the prior baselines in BLEU and human evaluations. However, this approach requires task-specific pretraining and remains limited to Java projects.

Mastropaolo *et al.* [18] contributed LANCE, the first framework capable of generating complete log statements, including position, level, and message. This work marked substantial progress toward end-to-end log automation, demonstrating the feasibility of integrating multiple logging elements in a single framework. Although promising, LANCE requires large-scale pretraining to achieve limited accuracy for message generation.

Mastropaolo *et al.* [17] extended this line of research with LEONID, which combines retrieval with pretraining to improve log statement injection. The system displayed stronger consistency and structural reliability than LANCE but continued to depend on Java-specific pretraining, limiting its portability across languages and domains.

In summary, prior studies have made significant progress toward automating various aspects of logging, from log placement and level recommendations to natural-language message generation and end-to-end statement injection. However, as discussed, all recent deep learning-based methods share a critical dependency on task- or domain-specific pretraining, limiting their generalizability and practical adoption in industrial contexts. This gap motivates this study, which evaluates whether large code models can achieve competitive or superior performance in automated logging tasks *without* custom pretraining.

B. LARGE LANGUAGE MODELS

Automated log generation can be considered a sequence-to-sequence problem. The source code is given as input, and the corresponding log statement is produced as output. This formulation aligns well with encoder-decoder architectures, where the encoder extracts structural information from the code and the decoder generates the target log message. Earlier neural approaches [17], [18] rely on encoder-decoder models, with T5 [22] and its extensions as representatives.

The T5 method [22] introduced the text-to-text paradigm, performing all natural-language processing tasks into a unified encoder-decoder framework. Although pretrained primarily on natural language, the T5 architecture is suitable for tasks that map structured inputs to textual outputs, making it a natural baseline for log generation.

Extending this design with code-aware pretraining objectives and a vocabulary enriched with source-code tokens, CodeT5 [25] enables stronger program understanding and semantic alignment. Thus, CodeT5 has become a common choice for code-related sequence generation tasks, including logging.

In contrast, recent LLMs, such as Qwen [2], have adopted a decoder-only architecture. Despite the absence of a separate encoder, their massive pretraining on mixed code and natural-language corpora endows them with strong structural

reasoning and fluent text generation. The Qwen2.5-Coder¹ is instruction-tuned for programming tasks across multiple languages. Its large parameter scales (1.5 B to 7 B) enable the examination of whether general-purpose decoder-only models can rival or surpass encoder-decoder systems in automated logging, potentially reducing the need for custom pretraining.

This architectural contrast reflects a broader shift in the field. Previous log automation systems have consistently relied on encoder-decoder models combined with task- and language-specific pretraining, often tied to a single programming language or repository. Although effective in controlled settings, such reliance hinders scalability and industrial adoption. Decoder-only models, including Qwen, demonstrate that large-scale pretraining across diverse code and natural-language corpora can yield competitive or superior performance without task-specific customization.

In summary, prior studies have heavily relied on encoder-decoder models with task-specific pretraining, whereas recent decoder-only LLMs demonstrate that broad pretraining can achieve comparable results without tailored adaptation. This transition motivates this work, which conducts the first comparison of pretraining-centric encoder-decoder approaches and general-purpose decoder-only code models for automated logging, clarifying when specialized pretraining is necessary and when modern LLMs already suffice.

III. RESEARCH MOTIVATION

As Sections I and II discuss, logging is indispensable for ensuring the reliability and maintainability of modern software systems. Effective logs accelerate debugging, enable performance monitoring, and provide safeguards against failures and risks [28]. Conversely, misplaced or low-quality logs can prolong problem resolution, introduce operational overhead, and potentially expose security vulnerabilities. In large and rapidly evolving projects, where millions of lines of code are constantly evolving, the cost of insufficient or incorrect logging compounds quickly [10].

Despite considerable progress in automated logging, most research has addressed only isolated aspects of the problem. For example, several approaches [5], [14], [15], [26] have focused solely on log-level recommendations or insertion-point predictions, without addressing other crucial components, such as variable binding or message generation. As Section II-A discusses, more recent research [17], [18] has advanced toward end-to-end log statement generation, yet these methods remain tied to specific programming languages (most commonly Java) and rely heavily on domain-specific pretraining. Such dependence limits their generalizability and poses practical barriers for adoption in diverse industrial contexts.

Moreover, LLMs trained on vast corpora of code and text exhibit robust performance in structural code reasoning

and natural-language generation without requiring custom pretraining for each downstream task [1], [2]. This contrast raises a fundamental question: *Are domain-specific pretrained models necessary for effective log automation, or can modern code LLMs achieve competitive or superior performance out of the box?* From a practical perspective, this is a matter of accuracy and deployability regarding whether a solution can scale across projects and languages without the prohibitive cost of continuous pretraining.

This gap between reliance on pretraining and the potential of off-the-shelf LLMs motivates this work. This study investigates whether SOTA off-the-shelf models, including T5 [22], CodeT5 [25], and Qwen [2], can perform automated logging tasks without additional pretraining. By comparing these general-purpose models against pretraining-centric baselines under identical conditions, this work aims to clarify the role of pretraining in automated logging and offer guidance on when specialized models are warranted and when a lightweight adaptation of general-purpose models is sufficient.

IV. DESIGN AND IMPLEMENTATION

This section presents the design and implementation of the proposed method. This section describes the automatic log generator and its workflow, covering single- and multilog injection (Section IV-A). Then, this section details the evaluation protocol for structural elements (level, variable, and position) and message quality (Section IV-B), followed by the datasets and experimental setup (Section V-B).

A. AUTOMATIC LOG GENERATOR

The proposed automatic log generator was designed to synthesize log statements for target methods by applying logging patterns observed in similar code (Fig. 1). Given a target method without logs, this approach retrieves structurally similar methods from the same project by computing the Jaccard similarity [20] over token sets. The Jaccard similarity was selected due to its efficiency and effectiveness in capturing the lexical overlap between code methods, which was sufficient for identifying reusable logging contexts [17]. From each retrieved method, existing log statements were removed via AST parsing to ensure the structural integrity of the code. The stripped code with the original log messages and similarity scores was combined with the target method to form the input sequence.

This work adopts three SOTA code language models as the generation engines: T5 [22], CodeT5 [25], and Qwen [2]. Unlike prior studies that require domain-specific pretraining on large Java corpora, the proposed models use general-purpose pretrained checkpoints and are fine-tuned only on the logging dataset, without any additional Java-specific pretraining. For T5-based models, this work extends the vocabulary with source-code-specific tokens to better capture programming constructs and semantics.

The models were configured to generate the four critical components of a log statement: log level, variables,

¹<https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct>

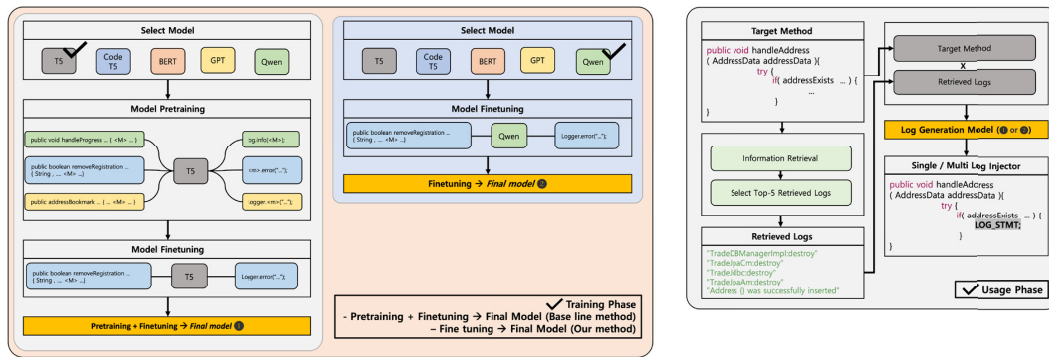


FIGURE 1. The framework consists of two phases. In the training phase (left), prior methods require domain-specific pretraining followed by finetuning to obtain a final model, whereas our approach eliminates pretraining and relies solely on lightweight finetuning of general-purpose code models. In the usage phase (right), given a target method without logs, structurally similar methods are retrieved from the project, and the log generation model predicts log statements—including level, variables, position, and message—for single or multi-log injection.

insertion position, and natural-language message. Although single-log injection is straightforward, multilog injection requires reasoning regarding the program control flow and structural context, rendering it a more challenging and realistic scenario. The final output is a completed method with synthesized log statements inserted at the predicted locations, enabling the assessment of the feasibility of log automation in diverse software settings.

B. EVALUATION OF GENERATED LOG

As Sections II-A and IV-A discuss, the proposed framework achieves competitive accuracy compared to prior methods and demonstrates cross-programming language applicability via fine-tuning alone, without additional pretraining. To assess the effectiveness of the proposed approach, we adopted the SOTA LEONID evaluation method [17]. To ensure a fair comparison, we reimplemented LEONID using the source code provided by the authors and executed it under the same experimental environment as the proposed models. This approach enables us to evaluate whether general-purpose pretrained models with supervised fine-tuning are sufficient to match or surpass systems that rely on extensive domain-specific pretraining. Although other studies (see Section II-A) provide valuable insight, their scope is often limited to individual logging elements (e.g., only levels or positions), making a direct end-to-end comparison with the proposed framework infeasible. Therefore, we restricted the primary evaluation baseline to LEONID, which supports comprehensive log statement injection.

For the single-log injection, where a target method contains exactly one log statement, we evaluated the accuracy of each individual logging element. The log levels, variables, and insertion positions were measured using the exact match against the ground truth, whereas the quality of the log message was assessed using multiple reference-based natural-language generation metrics, including BLEU [21], METEOR [3], ROUGE-RCS [16], and Levenshtein distance [12]. Fig. 2 illustrates the evaluation protocol for single-log injection, displaying the input methods, correct injection

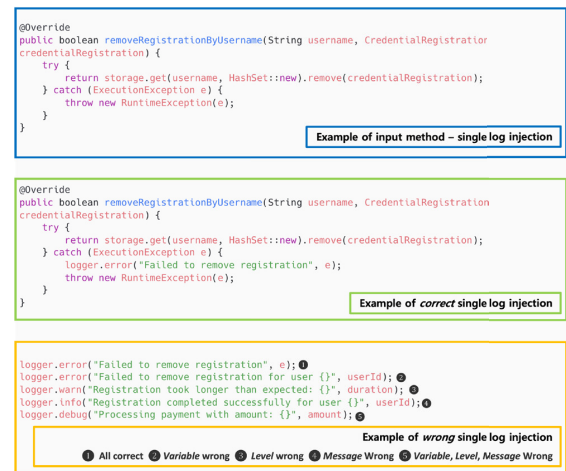


FIGURE 2. Examples of input, correct, and incorrect cases in single-log injection. Incorrect cases illustrate elementwise mismatches, such as variable, level, and message errors.

TABLE 2. Training hyperparameters for evaluated models.

Model	Epochs	Batch Size	Learning Rate	Weight Decay
T5	10	16	2e-5	0.01
CodeT5	3	16	2e-5	0.01
Qwen-1.5b	3	16	2e-4	0.001
Qwen-2.5-7b	3	4	2e-4	0.001

examples, and incorrect cases where individual elements (level, variable, and message) are mismatched.

For the multilog injection, where multiple log statements are required in the same method, the evaluation accounts for the correctness of the predicted number of log statements. Each generated statement is matched against reference logs in terms of the insertion positions, log levels, and variables using the exact match. Unlike the single-log case, we did not compute BLEU-based scores for log messages in multilog scenarios, as message quality is more difficult to quantify when multiple reference messages coexist. Fig. 3 presents the evaluation protocol for the multilog injection. Incorrect cases highlight that even small deviations, such as case sensitivity or spacing differences, are penalized under the exact-match criteria, reflecting the strictness of the evaluation.

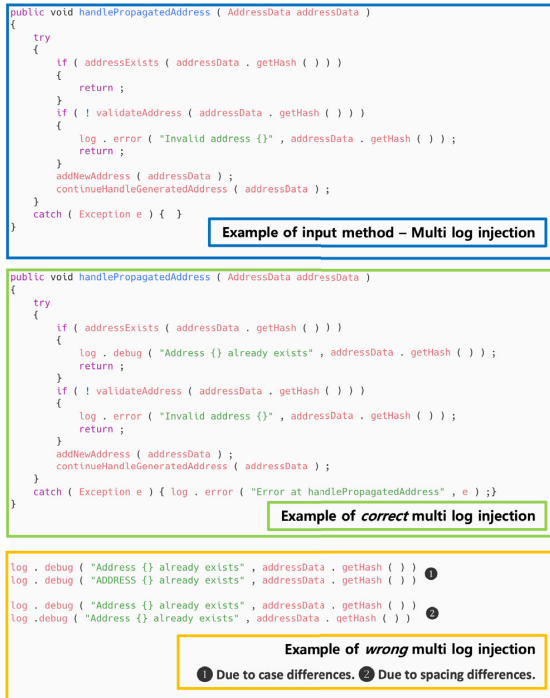


FIGURE 3. Examples of input, correct, and incorrect cases in multilog injection. Incorrect cases illustrate mismatches due to differences in case or spacing.

V. EXPERIMENTAL SETUP

A. DATASET

We evaluated the proposed models on the dataset introduced by LEONID [17]. The dataset was constructed from widely used Java projects on GitHub, selected based on popularity (measured by the minimum number of GitHub stars), activity, and code quality. Each method in these projects was annotated with corresponding log statements, including the log level, message, variable, and insertion position. This dataset has become the de facto benchmark for automated log generation. We adopted the same training, validation, and testing splits as in LEONID to ensure comparability, comprising 192,773 training instances, 24,092 validation instances, and 24,088 testing instances.

B. MODEL TRAINING SETUP

We trained four representative models without additional Java-specific pretraining: T5-base,² CodeT5-base,³ Qwen2.5-Coder-1.5B-Instruct,⁴ and Qwen2.5-Coder-7B-Instruct.⁵ All models were trained under comparable conditions, using the AdamW optimizer, with early stopping based on the validation loss. Table 2 summarizes the detailed hyperparameters.

In addition to model training, our evaluation pipeline incorporates the retrieval component used in LEONID [17]. We identify structurally similar methods from the same

project using Jaccard similarity over normalized tokens. A fixed minimum-overlap threshold of five shared tokens and the top- $k = 3$ neighbors are used, consistent with the configuration in LEONID's implementation. Token normalization includes lowercasing, punctuation removal, and Java identifier splitting. Retrieval is an essential part of the LEONID multilog pipeline, as retrieved examples provide structural context for predicting multiple insertion positions. Disabling this module causes the pipeline to fail, so we adopt it unchanged to ensure strict comparability with prior work.

Although T5 was included in the training setup, it consistently produced syntactically invalid Java code that could not be parsed by the AST-based evaluation pipeline used in LEONID. We attempted several mitigation strategies—including adding code-specific vocabulary, applying token-level normalization, and enforcing stricter decoding constraints—but the generated outputs remained unparsable. As a result, the structural and message-generation metrics could not be computed reliably for T5. Therefore, its results are reported as unavailable (“–”) in Tables 3 and 5. This limitation suggests that, without domain-specific pretraining on Java code, encoder-decoder models such as T5 struggle to maintain syntactic fidelity, which may also explain why prior work (e.g., LEONID [17]) did not consider T5 as a baseline.

VI. RESULTS

The goal of this study was to evaluate the effectiveness of general-purpose code models in supporting automated logging tasks compared to domain-specific pretrained systems. We assessed two scenarios: (i) single-log injection, measuring accuracy across log levels, variables, insertion positions, and message quality; and (ii) multilog injection, assessing the correctness of the number of log statements and the structural accuracy. The datasets described in Section V define the experimental context, and Section IV-B details the evaluation method.

This work aims to answer the following research questions:

- **RQ1:** How accurately do different models perform a single-log injection in terms of structural correctness and message quality?
- **RQ2:** How effectively can models handle multilog injection, including the number of statements and structural elements?
- **RQ3:** Beyond exact matching, how do models compare in robustness and message quality, particularly in multilog scenarios where BLEU provides a more tolerant measure?

A. RESEARCH QUESTION 1

Tables 3 and 4 report the results of single-log injection across structural elements and log-message quality. As illustrated in Fig. 2, single-log injection was evaluated along four structural dimensions: variable, level, message, and position, alongside message quality metrics, including BLEU, METEOR,

²<https://huggingface.co/google-t5/t5-base>

³<https://huggingface.co/Salesforce/codet5-base>

⁴<https://huggingface.co/Qwen/Qwen2.5-Coder-1.5B-Instruct>

⁵<https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct>

TABLE 3. Accuracy on structural elements of single-log injection.

Method	Variable	Level	Message	Position
LEONID [17]	56.80	72.83	33.09	79.61
T5 [22]	-	-	-	-
CodeT5 [25]	53.72	66.33	34.11	89.54
Qwen-1.5b [2]	51.38	65.59	29.25	94.11
Qwen-7b [2]	56.74	70.99	34.32	93.76

ROUGE-RCS, and the Levenshtein distance. Based on these criteria, we derived four critical findings.

First, although LEONID [17], the SOTA baseline, relies on extensive pretraining on Java projects, the models without domain-specific pretraining achieved comparable or superior performance. This baseline finding demonstrates that recent LLMs, pretrained on diverse code and natural language, can rival or outperform task-specific systems without being tied to a single programming language.

Second, when predicting log insertion positions, a crucial aspect for the practical usefulness of logging, the proposed models, particularly Qwen-7b and Qwen-1.5b, outperformed LEONID (93.76% and 94.11%, respectively, vs. 79.61%). As in Fig. 2, incorrect log positions can cause syntactically invalid code and compilation errors. Therefore, superior performance in position prediction underscores a decisive advantage of large code models because it ensures that generated logs integrate seamlessly into the host program.

Third, variable prediction remains a challenging subtask across all models, with accuracy values of 50% to 57%. Although Qwen-7b matched LEONID in this regard, none of the models achieved consistently high accuracy. This result indicates that grounding log statements to the correct program variables remains a problem for domain-specific and general-purpose models.

Fourth, log-message quality metrics reveal a tradeoff. Qwen and CodeT5 performed comparably or better than LEONID on the BLEU, METEOR, ROUGE-RCS, and Levenshtein distance metrics, suggesting that broad pretraining on diverse codebases enables more natural and readable messages. In contrast, LEONID achieved the highest accuracy in log-level prediction (72.83%), reflecting the stability conferred by task-specific pretraining. These results indicate a tradeoff between stability (log level and variable fidelity) and adaptability (message fluency and structural reasoning).

Finally, contrasting LEONID with the proposed off-the-shelf models clarifies the cost–benefit tradeoff. The LEONID model achieved the highest accuracy in log-level prediction; however, it relies on Java-specific pretraining and task-specific engineering. In contrast, Qwen-7b/1.5b and CodeT5, without domain-specific pretraining, match or exceed the results of LEONID on message quality and outperform it in terms of insertion position. These results indicate that the marginal gains of custom pretraining are concentrated on the level or variable stability, whereas general-purpose models deliver superior structural placement and comparable

or better messages with a much lower engineering cost and greater portability across repositories and languages.

Thus, the single-log injection experiments revealed that off-the-shelf large code models, such as Qwen, offer clear advantages. These models combine competitive message generation with superior structural accuracy, avoiding the cost and inflexibility of custom pretraining. Their decisive gains on the insertion position, the most critical element for ensuring syntactic validity and seamless integration into the host program, demonstrate that large code models can rival and surpass domain-specific pretrained systems in the critical aspects for practical logging automation.

B. RESEARCH QUESTION 2

Table 5 presents the results of multilog injection measured using the exact-match accuracy. As illustrated in Fig. 3, multilog injection requires models to predict the correct position, level, and variables for each log statement and match the exact number of log statements in a method. This strict protocol means that even minor discrepancies in a log message, such as spacing, punctuation, or variable ordering, are treated as errors. Thus, multilog injection is substantially more challenging than single-log scenarios, and Fig. 3 demonstrates how semantically valid but superficially mismatched predictions are penalized under the exact-match approach. This finding highlights a critical limitation: exact matching underestimates practical utility when multiple valid log variants exist.

Multilog injection is of practical importance, as real-world methods often require multiple logs at various stages (e.g., input validation, process monitoring, and output reporting). Failure to manage this task correctly forces developers to edit or add missing statements manually, reducing the effectiveness of automated logging systems. Although the absolute accuracy values remain low, this reflects the strictness of the exact match; in practice, many near-miss predictions are still semantically valid and interpretable by developers.

In the experiments, T5 and CodeT5 failed to achieve exact matches, consistent with their limited code-specific pretraining (see Section VI-A). In contrast, the Qwen models presented clear promise: Qwen-7b slightly outperformed LEONID (17.44% vs. 16.99%), and Qwen-1.5b performed competitively. This result indicates that large code models can approach and exceed domain-specific systems on this challenging task, despite lacking log-specific pretraining. Unlike LEONID, which requires extensive customization for Java, Qwen achieved this performance off-the-shelf, underscoring the portability and efficiency of modern code LLMs.

To the best of our knowledge, this is the first empirical study demonstrating that multilog injection is feasible with general-purpose LLMs. These findings imply that future advances could benefit from hybrid designs that integrate the retrieval of similar logging contexts with LLM generation,

TABLE 4. Natural-language quality metrics on single-log messages.

Method	BLEU-1	BLEU-2	BLEU-3	BLEU-4	METEOR	ROUGE-LCS Precision	ROUGE-LCS Recall	ROUGE-LCS F-measure	LEVENSHTEIN
LEONID [17]	50.02	44.26	39.65	35.55	51.42	43.66	42.67	42.11	50.66
T5 [22]	-	-	-	-	-	-	-	-	-
CodeT5 [25]	51.18	45.54	41.01	36.78	52.68	48.26	45.52	45.62	48.78
Qwen-1.5b [2]	52.03	45.86	40.94	36.53	50.66	45.05	44.93	43.70	53.04
Qwen-7b [2]	48.11	41.69	36.79	32.56	53.75	50.03	47.94	47.64	48.43

TABLE 5. Exact-match accuracy for multi-log injection.

Method	Leonid	T5	CodeT5	Qwen-1.5b	Qwen-7b
Exact match	16.99	-	-	13.51	17.44

improving robustness in multilog scenarios and reducing the manual effort required in practice.

C. RESEARCH QUESTION 3

Before presenting the natural-language evaluation, we first note that multi-log injection in prior works (e.g., LEONID [17] and LANCE [18]) is assessed using strict exact-match criteria. While such criteria are necessary for ensuring structural correctness, they often underestimate practical utility, as small lexical differences may not affect the semantics of a log message. To address this limitation, our evaluation includes complementary relaxed metrics for the message-generation component. These reference-based metrics (BLEU, METEOR, ROUGE-L, Levenshtein) capture semantic similarity even when outputs differ lexically, thereby providing a more balanced assessment of model performance in multi-log scenarios.

Sections VI-A and VI-B focus on the structural accuracy of log injection, and this section complements that perspective by examining the natural-language quality of the generated messages. This distinction is crucial: even if the insertion positions, variables, and levels are correct, poor or unintelligible messages can undermine the practical utility of the automated logs.

Table 6 presents the natural-language quality metrics for single-log-message generation, where LEONID [17] achieved the highest BLEU scores, yet its authors noted that the message quality is often underestimated by automatic metrics. Manual inspection revealed that many outputs penalized as errors differed only in spacing, punctuation, or capitalization, remaining semantically equivalent to the reference. The same phenomenon occurred in this work: exact-match-based metrics exaggerated errors, and many near-miss messages remained acceptable to developers.

Manual inspection at scale is impractical; hence, this work relies on established reference-based metrics (BLEU, METEOR, ROUGE-L, and Levenshtein distance) to approximate the message quality. These metrics provide complementary perspectives. For example, BLEU is highly sensitive to token overlap, often sharply decreasing for minor variations. Furthermore, METEOR and the Levenshtein distance are more tolerant, capturing semantic similarity despite lexical or structural differences. This reference-based metric per-

formance explains why CodeT5, despite its lower BLEU scores, performs well in terms of the METEOR and Levenshtein distance, indicating concise yet semantically accurate messages. The Qwen models achieved high BLEU scores by producing lexically overlapping content, but the weaker Levenshtein distance results suggest verbosity or structural drift. Although behind on BLEU, T5 attained competitive ROUGE scores, preserving critical tokens and phrases even with limited fluency. To document this phenomenon more concretely, Appendix section provides representative near-miss examples from our evaluation. Although these outputs fail under exact-match scoring, they preserve the same semantics as the reference and would be acceptable to developers in practice.

These findings align with broader observations in log research. Prior studies have emphasized that log data are unstable, and anomaly detection depends more on capturing robust patterns than on exact textual matches [27]. Large-scale evaluations of log parsing confirm that accuracy significantly deteriorates for rare or parameter-heavy logs, underscoring the need for robustness rather than correctness [9]. Recent prompt-based few-shot approaches have demonstrated that preserving semantic fidelity is more important than exact token matching [13].

Overall, these results highlight the necessity of interpreting automatic metrics in context. Developers rarely require exact replication; instead, clarity in expressing variables, system states, and error descriptions is more critical. Moreover, prior empirical studies have found that even human-written log messages are often revised or rephrased during software evolution [9], indicating that exact textual replication is rarely maintained in practice. In realistic workflows, automated systems present candidate logs for developer review. Further, near-miss outputs, which are penalized by metrics but semantically valid, retain their substantial practical value.

Therefore, the findings demonstrate that recent general-purpose models can produce log messages of meaningful quality without log-specific pretraining. Although LEONID relies on extensive domain-specific pretraining to achieve strong BLEU overlap, large code models (e.g., Qwen and CodeT5) deliver comparable or superior performance in metrics that better reflect semantic fidelity. This finding implies that future advances may benefit from hybrid designs that integrate the retrieval of similar logging contexts with LLM generation, improving robustness in multilog scenarios and reducing manual efforts in practice.

TABLE 6. Natural-language quality metrics for multi-log messages.

Method	BLEU-1	BLEU-2	BLEU-3	BLEU-4	METEOR	ROUGE-LCS Precision	ROUGE-LCS Recall	ROUGE-LCS F-measure	LEVENSHTEIN
LEONID	93.27	92.46	91.54	90.52	77.89	79.08	77.57	77.80	18.40
T5	75.66	70.39	65.99	61.57	72.41	84.93	82.63	83.35	17.55
CodeT5	89.28	85.86	82.59	79.25	79.30	84.35	82.69	83.11	14.93
Qwen-1.5b	91.04	88.49	85.91	83.08	75.27	75.98	74.49	74.64	21.67
Qwen-7b	91.54	89.10	86.64	83.94	70.27	69.98	67.89	68.19	27.95

VII. DISCUSSION AND FUTURE WORK

This study offers insight into the comparative performance of pretraining-based and general-purpose models for automated logging while addressing the broader challenges that remain in this field. The results demonstrate that large code models can rival or surpass specialized systems in several dimensions, yet they also reveal crucial limitations that motivate future research.

One limitation is the dataset, as the evaluation used LEONID, relying on a large-scale Java corpus (the de facto benchmark for log generation). Although this approach enables direct comparability, it restricts the findings to a single language ecosystem. The generalizability of the results to other programming languages (e.g., Python or C++) or industrial projects with heterogeneous logging conventions remains uncertain. Ground-truth logs are unstable, and projects adopt widely varying styles, from concise state reporting to verbose capturing of every variable. Such stylistic diversity undermines the assumption that reference logs are an absolute standard. Thus, the exact-match evaluation often penalizes semantically valid and useful generated logs because the style differs from that of the reference.

Although our evaluation is limited to Java due to dataset availability, it is important to note that the LEONID benchmark is already constructed from a diverse collection of independent GitHub repositories. Each project adopts its own coding conventions, logging practices, and structural patterns, meaning that the fine-tuning and evaluation pipelines naturally operate under a cross-project setting.

Unlike LEONID—which requires task-specific pretraining on 12.6M unlabeled Java methods to support generalization—our models rely solely on supervised fine-tuning over 192k–229k annotated instances drawn from heterogeneous projects. The ability of general-purpose LLMs such as Qwen and CodeT5 to match or exceed LEONID without any task-specific pretraining provides implicit evidence that these models generalize across projects within the scope of the available benchmark. Nevertheless, evaluating cross-language or out-of-distribution repositories remains an open challenge and represents an important direction for future work once comparable datasets become available.

The limitations of the current evaluation metrics compound this problem. The BLEU, METEOR, ROUGE-L, and Levenshtein distance metrics capture surface-level textual similarity without fully measuring whether a log message conveys the correct program state or variable values. The results illustrate

this mismatch, as CodeT5 produced semantically accurate messages that scored poorly under BLEU, whereas Qwen realized higher BLEU scores through verbose outputs that contributed little utility. These discrepancies highlight the necessity for new semantically aware metrics or human-in-the-loop evaluations that more accurately reflect developer usefulness.

In addition, our evaluation does not account for deeper semantic factors such as verbosity drift, duplicate-token inflation, or adherence to message content constraints (e.g., explicit mention of key variables, state descriptors, or error identifiers). These aspects are important for assessing the practical usefulness of developer-facing logs but are not captured by surface-level n-gram metrics. Since our study follows the LEONID protocol for strict comparability, we intentionally did not introduce new message templates or human annotation criteria. Nevertheless, developing semantically grounded evaluation frameworks—potentially combining message constraints, verbosity analysis, and human-in-the-loop assessments—represents a valuable direction for future research.

Thus, as our evaluation protocol follows LEONID, all structural and textual judgments are handled automatically using deterministic metrics, without manual adjudication or inter-annotator validation. Borderline cases—such as spacing, casing, or punctuation differences—are therefore marked as errors even when the generated message is semantically equivalent to the reference. To make these phenomena transparent, Appendix section provides representative near-miss examples that illustrate where automated metrics diverge from human judgment. While this ensures strict reproducibility of the evaluation, developing semantically grounded or human-in-the-loop assessment protocols remains an important direction for future work.

We also conducted exploratory tests using generative pretrained transformer 4.5 (GPT-4.5) to examine whether prompt-only instruction tuning could serve as a practical solution. However, the results were unstable and often not reproducible. Even in single-log scenarios, the model frequently generated multiple log statements or failed to adhere to the constraints, rendering systematic analysis infeasible. Therefore, this work does not report the quantitative results for GPT, but these preliminary observations indicate that prompt-only approaches are insufficient for logging automation, where reliability and control are crucial. Controllable generation techniques, such as lightweight fine-tuning, constraint-based decoding, or structured generation, are necessary to make such models practical.

Our study is designed around a focused objective: evaluating whether LEONID's task-specific pretraining over 12.6 million unlabeled Java methods is truly necessary, or whether modern general-purpose LLMs can achieve comparable performance using only supervised fine-tuning. To ensure a clean comparison, we intentionally kept the entire LEONID pipeline unchanged—including retrieval, task formulation, and evaluation design. Altering the pipeline (e.g., modifying retrieval settings, redefining sub-tasks, or probing architectural factors in isolation) would introduce multiple confounding effects, making it unclear whether performance differences arise from the model architecture or from changes in the task definition itself.

Even within this controlled setup, architectural patterns naturally emerge: Qwen (decoder-only) demonstrates strong structural sensitivity, particularly in insertion-position and multilog prediction; CodeT5 (encoder-decoder) provides high semantic fidelity in message generation; and LEONID remains stable in categorical sub-tasks such as variable and log-level prediction. These observations illustrate complementary strengths across model families without modifying the evaluation framework.

A particularly notable finding is that our fine-tuned general-purpose models require only 192k–229k annotated training instances, yet still match—or even surpass—LEONID, which depends on extensive task-specific pretraining over millions of unlabeled methods. This suggests that such large-scale domain-specific pretraining may no longer be necessary, substantially reducing both data and computational requirements for automated logging systems.

This comparison highlights complementary strengths across model classes. Qwen excels in structural sensitivity, achieving superior performance in insertion position and multilog scenarios. CodeT5 demonstrates strong semantic fidelity in message generation, whereas LEONID remains stable and reliable in log-level and variable prediction.

In addition, our evaluation relies on the retrieval mechanism defined in LEONID [17], which identifies structurally similar methods using Jaccard similarity over normalized tokens. We adopt the same configuration—lowercasing, identifier splitting, punctuation removal, a minimum-overlap threshold of five shared tokens, and the top-3 nearest neighbors ($k = 3$)—to ensure strict comparability with the pretraining-based baseline. Because LEONID's multilog pipeline depends on retrieved examples to supply structural context, disabling this module prevents the system from producing valid predictions, making an on/off ablation infeasible. Prior work by the same authors (LEONID and LANCE [18]) has already shown that removing retrieval leads to significant drops in multilog accuracy, indicating that its contribution has been previously validated. Nevertheless, its robustness under code duplication, vendor directories, or generated sources remains an open challenge, and exploring alternative similarity measures—such as AST-based similarity, control-flow proximity, or deduplication-

aware retrieval—offers a promising avenue for improving multilog prediction in future work.

We retain the Jaccard similarity configuration because it is the exact retrieval mechanism used in LEONID and LANCE, enabling a controlled comparison that isolates the impact of pretraining versus fine-tuning. Replacing Jaccard with embedding-based or AST-aware retrieval would modify the structural context provided to all models, introducing confounding factors that obscure the interpretation of our results. Moreover, prior work has shown that Jaccard similarity is effective for identifying structurally similar methods in the multilog setting, where token-level overlap better reflects structural proximity than semantic similarity. While exploring richer retrieval mechanisms such as AST-based similarity, control-flow proximity, or deduplication-aware retrieval is an important direction for future work, such investigation lies outside the scope of our controlled evaluation framework.

A notable limitation of this study is the relatively low accuracy in multi-log injection (<20%) across all evaluated models. This limitation is not specific to general-purpose LLMs: prior work such as LEONID reports similarly low performance in multilog settings, reflecting the inherent difficulty of predicting multiple insertion positions within a single method. This suggests that the challenge arises from the task formulation itself, rather than from any particular model architecture. While our primary goal is to assess whether task-specific pretraining is necessary—and the results indicate that general-purpose models can perform comparably to pretraining-based baselines—improving multilog robustness remains an important direction for future research. Techniques such as retrieval-enhanced generation, structured decoding constraints, or multi-position prediction mechanisms may help mitigate this challenge.

Therefore, no single model dominates across all dimensions. Hybrid designs that integrate complementary strengths may offer the most effective solutions. For example, retrieving similar logging contexts and combining them with LLM-based generation could enhance robustness in multilog injection while further reducing reliance on domain-specific pretraining.

Our evaluation corpus is constructed from multiple independent open-source repositories, each with its own coding conventions and logging styles. While such repository-level differences may influence message phrasing or stylistic variation, our study does not explicitly examine how generated logs differ across projects serving different audiences or application domains. Prior work has noted that demographic or domain-specific factors can affect textual characteristics in developer-facing messages [6], [24]. A diversity-aware analysis—examining how log messages vary across repositories or user contexts—remains outside the scope of this work but represents a promising direction for future research.

Overall, these observations indicate several promising directions for future research. Research should advance toward semantically aware evaluation methods that better

capture the practical utility of the generated logs. Expanding datasets to cover multiple languages and diverse repositories is essential for validating generalization beyond use for Java. Hybrid retrieval-generation approaches are promising for combining stability with flexibility, and controllable generation methods are necessary to address the limitations of prompt-only LLMs. Finally, automated logging should be framed as a developer-in-the-loop tool; rather than replacing human judgment, systems should present candidate logs that developers can review, edit, and adopt. Integrating these workflows into integrated development environments (IDEs) or continuous integration pipelines offers value in real-world software engineering practice. In summary, although the findings reveal that custom pretraining is not necessarily required for automated logging, achieving practical adoption requires advances in model design, evaluation methods, dataset diversity, and integration into developer workflows.

VIII. CONCLUSION

This study revisits the role of pretraining in automated logging and presents the first systematic comparison of domain-specific pretrained systems and general-purpose code LLMs. The results demonstrate that large code models, including Qwen and CodeT5, can perform comparably to or superior to SOTA pretrained systems while eliminating costly, project-specific pretraining.

Beyond accuracy, the findings have practical implications. Pretraining pipelines require considerable data collection, annotation, and computational costs, making them challenging to sustain in practice. In contrast, the results indicate that off-the-shelf models offer robust performance, particularly in insertion position prediction, which is critical for syntactic correctness and seamless integration. This approach substantially reduces the engineering overhead for deploying logging automation in real-world systems.

These results broaden the industrial applicability of logging automation, suggesting that this approach can be adopted across diverse projects and languages without prohibitive customization. For practitioners, this applicability enables the integration of automated logging into IDEs or continuous integration/continuous delivery workflows as a lightweight, developer-assist tool. For researchers, the study reframes the role of pretraining in code intelligence. Rather than a universal requirement, pretraining emerges as a targeted technique for stabilizing variables and levels, whereas general-purpose models already deliver robust outcomes in message quality and structural reasoning.

Future work should advance semantic-aware evaluation metrics, explore cross-language datasets, and investigate hybrid designs that combine retrieval with LLM-based generation. Automated logging should evolve into a developer-in-the-loop paradigm, balancing automation with human oversight to ensure reliability and practical value in software engineering practice.

DATA AVAILABILITY

All datasets, prompt example, and source code used in this study are available anonymously at <https://figshare.com/s/fb93db8e62798a485bf3>

APPENDIX

A. WHITESPACE DIFFERENCES

```
@Override
public boolean removeRegistrationByUsername(
    String username,
    CredentialRegistration credentialRegistration
) {
    try {
        return storage.get(username, HashSet::new).remove(credentialRegistration);
    } catch (ExecutionException e) {
        logger.error("Failed to remove registration", e);
        throw new RuntimeException(e);
    }
}

@Override
public boolean removeRegistrationByUsername(
    String username,
    CredentialRegistration credentialRegistration
) {
    try {
        return storage.get(username, HashSet::new).remove(credentialRegistration);
    } catch (ExecutionException e) {
        logger.error("Failed to remove registration", e);
        throw new RuntimeException(e);
    }
}
```

B. LOGGER IDENTIFIER VARIANTS

```
private Map<String, String> createCodeMap(String url) {
    LOG.info("Creating language code map from {}", url);
    try {
        Map<String, String> result =
            ImmutableMap.copyOf(CommonUtils.createCMDIComponentItemMap(url));
        return result;
    } catch (Exception e) {
        if (CommonUtils.shouldSwallowLookupErrors()) {
            LOG.warn("Ignoring exception", e);
            return Collections.emptyMap();
        } else {
            throw new RuntimeException(
                "Cannot instantiate postProcessor. URL: " + url, e);
        }
    }
}

private Map<String, String> createCodeMap(String url) {
    LOGGER.info("Creating language code map from {}", url);
    try {
        Map<String, String> result =
            ImmutableMap.copyOf(CommonUtils.createCMDIComponentItemMap(url));
        return result;
    } catch (Exception e) {
        if (CommonUtils.shouldSwallowLookupErrors()) {
            LOG.warn("Ignoring exception", e);
            return Collections.emptyMap();
        } else {
            throw new RuntimeException(
                "Cannot instantiate postProcessor. URL: " + url, e);
        }
    }
}

private Map<String, String> createCodeMap(String url) {
    get_logger().info("Creating language code map from {}", url);
    try {
        Map<String, String> result =
            ImmutableMap.copyOf(CommonUtils.createCMDIComponentItemMap(url));
        return result;
    } catch (Exception e) {
        if (CommonUtils.shouldSwallowLookupErrors()) {
            LOG.warn("Ignoring exception", e);
            return Collections.emptyMap();
        } else {
            throw new RuntimeException(
                "Cannot instantiate postProcessor. URL: " + url, e);
        }
    }
}

private Map<String, String> createCodeMap(String url) {
    DEBUG.info("Creating language code map from {}", url);
    try {
        Map<String, String> result =
            ImmutableMap.copyOf(CommonUtils.createCMDIComponentItemMap(url));
        return result;
    } catch (Exception e) {
        if (CommonUtils.shouldSwallowLookupErrors()) {
            LOG.warn("Ignoring exception", e);
            return Collections.emptyMap();
        } else {
            throw new RuntimeException(
                "Cannot instantiate postProcessor. URL: " + url, e);
        }
    }
}
```

C. CASE SENSITIVITY

```
public static Connection getConnection() {
    Connection connection = null;
    if (null != PgConnection.source) {
        try {
            connection = source.getConnection();
        } catch (SQLException e) {
            logger.warn("Something went wrong, PostgreSQL DB is not connected.");
            logger.error(e.getMessage());
        }
    }
    return connection;
}

public static Connection getConnection() {
    Connection connection = null;
    if (null != PgConnection.source) {
        try {
            connection = source.getConnection();
        } catch (SQLException e) {
            logger.warn("Something went wrong, PostgreSQL DB is not connected.");
            logger.error(e.getMessage());
        }
    }
    return connection;
}
```

D. PUNCTUATION AND EMPHASIS VARIATIONS

```
protected void returnExtractor(ContentExtractor e) {
    log.debug("Returning extractor to the pool.");
    synchronized (extractors) {
        try {
            e = new ContentExtractor();
            extractors.add(e);
        } catch (AnalysisException ex) {
            throw new RuntimeException("Cannot create extractor!", ex);
        }
        extractors.notify();
    }
}

protected void returnExtractor(ContentExtractor e) {
    log.debug("Returning extractor to the pool.");
    synchronized (extractors) {
        try {
            e = new ContentExtractor();
            extractors.add(e);
        } catch (AnalysisException ex) {
            throw new RuntimeException("Cannot create extractor!", ex);
        }
        extractors.notify();
    }
}

@Override
public void attack() {
    LOGGER.info("The troll tries to grab you!");
}

@Override
public void attack() {
    LOGGER.info("The troll tries to grab you!");
}
```

REFERENCES

- [1] J. Achiam et al., "GPT-4 technical report," 2023, *arXiv:2303.08774*.
- [2] J. Bai et al., "Qwen technical report," 2023, *arXiv:2309.16609*.
- [3] S. Banerjee and A. Lavie, "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments," in *Proc. ACL Workshop Intrinsic Extrinsic Eval. Measures Mach. Transl. Summarization*, 2005, pp. 65–72.
- [4] M. A. Batoun, M. Sayagh, R. Aghili, A. Ouni, and H. Li, "A literature review and existing challenges on software logging practices: From the creation to the analysis of software logs," *Empirical Softw. Eng.*, vol. 29, no. 4, p. 103, Jul. 2024.
- [5] Z. Ding, H. Li, and W. Shang, "LoGenText: Automatically generating logging texts using neural machine translation," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Mar. 2022, pp. 349–360.
- [6] E. Fosch-Villaronga, A. Poulsen, R. A. Søråa, and B. H. M. Custers, "A little bird told me your gender: Gender inferences in social media," *Inf. Process. Manage.*, vol. 58, no. 3, May 2021, Art. no. 102541.
- [7] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? An empirical study on logging practices in industry," in *Companion Proc. 36th Int. Conf. Softw. Eng.*, May 2014, pp. 24–33.
- [8] Y. Fu, M. Yan, P. He, C. Liu, X. Zhang, and D. Yang, "End-to-end log statement generation at block-level," *J. Syst. Softw.*, vol. 216, Oct. 2024, Art. no. 112146.
- [9] Z. Jiang, J. Liu, J. Huang, Y. Li, Y. Huo, J. Gu, Z. Chen, J. Zhu, and M. R. Lyu, "A large-scale evaluation for log parsing techniques: How far are we," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2024, pp. 223–234.
- [10] S. Kabinna, C.-P. Bezemer, W. Shang, M. D. Syer, and A. E. Hassan, "Examining the stability of logging statements," *Empirical Softw. Eng.*, vol. 23, no. 1, pp. 290–333, Feb. 2018.
- [11] S. Lal, N. Sardana, and A. Sureka, "ECLogger: Cross-project catch-block logging prediction using ensemble of classifiers," *e-Informatica Softw. Eng. J.*, vol. 11, no. 1, 2016.
- [12] V. Lcvshentsin, "Binary Coors capable or 'correcting deletions, insertions, and reversals,'" *Soviet Phys.-Doklady*, vol. 10, no. 8, pp. 1–4, 1966.
- [13] V.-H. Le and H. Zhang, "Log parsing with prompt-based few-shot learning," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 2438–2449.
- [14] Z. Li, T.-H. Chen, and W. Shang, "Where shall we log? Studying and suggesting logging locations in code blocks," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2020, pp. 361–372.
- [15] Z. Li, H. Li, T.-H. Chen, and W. Shang, "DeepLV: Suggesting log levels using ordinal based neural networks," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, May 2021, pp. 1461–1472.
- [16] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81.
- [17] A. Mastropaolo, V. Ferrari, L. Pascarella, and G. Bavota, "Log statements generation via deep learning: Widening the support provided to developers," *J. Syst. Softw.*, vol. 210, Apr. 2024, Art. no. 111947.
- [18] A. Mastropaolo, L. Pascarella, and G. Bavota, "Using deep learning to generate complete log statements," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, May 2022, pp. 2279–2290.
- [19] T. Mizouchi, K. Shimari, T. Ishio, and K. Inoue, "PADLA: A dynamic log level adapter using online phase detection," in *Proc. IEEE/ACM 27th Int. Conf. Program Comprehension (ICPC)*, May 2019, pp. 135–138.
- [20] S. Niwattanakul, J. Singthongchai, E. Naenudorn, and S. Wanapu, "Using of Jaccard coefficient for keywords similarity," in *Proc. Int. Multiconference Eng. Comput. Scientists*, vol. 1, 2013, pp. 380–384.
- [21] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics*, Philadelphia, PA, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318.
- [22] C. Raffel, N. Shazeer, A. P. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, 2019.
- [23] K. Schmidt, C. Phillips, and A. Chuvakin, *Logging Log Management: Authoritative Guide to Understanding the Concepts Surrounding Logging and Log Management*. London, U.K.: Newnes, 2012.
- [24] N. Thakur, S. Cui, K. Khanna, V. Knieling, Y. N. Duggal, and M. Shao, "Investigation of the gender-specific discourse about online learning during COVID-19 on Twitter using sentiment analysis, subjectivity analysis, and toxicity analysis," *Computers*, vol. 12, no. 11, p. 221, Oct. 2023.
- [25] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," 2021, *arXiv:2109.00859*.
- [26] K. Yao, G. B. de Pádua, W. Shang, S. Sporea, A. Toma, and S. Sajedi, "Log4Perf: Suggesting logging locations for web-based systems' performance monitoring," in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, Mar. 2018, pp. 127–138.
- [27] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J.-G. Lou, M. Chintalapati, F. Shen, and D. Zhang, "Robust log-based anomaly detection on unstable log data," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Aug. 2019, pp. 807–817.
- [28] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "Log20: Fully automated optimal placement of log printing statements under specified overhead threshold," in *Proc. 26th Symp. Operating Syst. Princ.*, Oct. 2017, pp. 565–581.



DAE-SUNG WANG was born in Jeonju, South Korea. He received the B.S. degree in software engineering from Jeonbuk National University, Jeonju, the M.S. degree in computer science and engineering from Chung-Ang University, Seoul, South Korea, where he is currently pursuing the Ph.D. degree in computer science and engineering. His research interests include software engineering, requirement engineering, and natural language processing.



CHAN-GUN LEE was born in Seoul, South Korea, in 1972. He received the B.S. degree in computer engineering from Chung-Ang University, Seoul, in 1996, the M.S. degree in computer science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, in 1998, and the Ph.D. degree in computer science from The University of Texas at Austin, Austin, TX, USA, in 2005. From 2005 to 2007, he was a Senior Software Engineer at Intel, Hillsboro, OR, USA.

Since 2007, he has been a Professor with the Department of Computer Science and Engineering, Chung-Ang University. He is the author of more than 30 articles and conference papers. His research interests include software engineering and real-time systems. He was a recipient of Korea Foundation of Advanced Studies (KFAS) Fellowship, from 1999 to 2005.



TAE-IL KIM was born in Gyeonggi-do, South Korea. He received the B.S. degree in software engineering from Jeonbuk National University, Jeonju, in 2023. He is currently pursuing the M.S. degree in computer science and engineering with Chung-Ang University, Seoul, South Korea. His research interests include software engineering, change impact analysis, and natural language processing.



HYUN-TAEK HONG received the M.S. degree in information engineering. He is currently pursuing the Ph.D. degree in computer engineering with Chung-Ang University. He is a Certified LG Software Architect, having completed the software architect course at Carnegie Mellon University. He is currently the Lead Architect with LG Electronics, with a work focuses on software defined vehicle and software platform to enhance vehicle flexibility through evolving software. He serves

as a Board Member for the Vehicle Semiconductor and System Software Society, The Korea Society of Automotive Engineers (KSAE).

...