Towards Universal Segmentation for Log Parsing

Van-Hoang Le*
University of Luxembourg
Luxembourg
vanhoang.le@uni.lu

Domenico Bianculli
University of Luxembourg
Luxembourg
domenico.bianculli@uni.lu

Huy-Trung Nguyen
Posts and Telecommunications Institute of Technology
Vietnam
trungnh@ptit.edu.vn

Abstract

Log parsing is a crucial step in log analysis facilitating program comprehension throughout software maintenance and engineering life cycles. Log parsing transforms unstructured log messages into structured data required by various downstream analysis tasks. The sheer volume of log data generated by modern software systems motivates the development of numerous log parsing techniques in the literature. However, existing log parsers still suffer from unsatisfactory accuracy, which may significantly affect the follow-up analysis such as log-based anomaly detection. We have identified two main limitations that hinder the effectiveness of existing log parsing methods: (1) under-segmentation: most log parsers leverage a fixed, predefined set of delimiters to separate a log message into a set of tokens, which may fail to split log messages correctly due to the heterogeneity of logging formats; (2) over-segmentation: using too many delimiters may lead to the over-segmentation issue, which fragments meaningful units in log messages and makes it difficult to accurately identify templates and parameters. To address these limitations, we propose SCLog, a novel syntax- and contextual-aware segmentation approach for log parsing. SCLog leverages a comprehensive set of syntax-based heuristics to segment log messages into coarse-grained tokens. To further tokenize log messages into fine-grained tokens, SCLog mines the structural patterns of tokens based on their surrounding contexts to identify the optimal delimiters for each token dynamically. We evaluate SCLog on widely-used, large-scale Loghub-2.0 datasets. The results demonstrate that SCLog significantly improves the parsing accuracy of four representative log parsers.

CCS Concepts

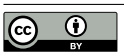
• **Software and its engineering → Software maintenance tools**.

Keywords

Log Parsing, Segmentation, Syntactic Analysis, Structural Patterns

ACM Reference Format:
Van-Hoang Le, Domenico Bianculli, and Huy-Trung Nguyen. 2026. Towards Universal Segmentation for Log Parsing. In *34th IEEE/ACM International Conference on Program Comprehension (ICPC '26), April 12–13, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3794763.3794811

*Corresponding author



ICPC '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2482-4/2026/04
https://doi.org/10.1145/3794763.3794811

1 Introduction

Software-intensive systems generate massive amounts of log data, which is printed by logging statements (e.g., `printf()`, `log.info()`) in the source code. Log data serves as runtime representations of program behavior, enabling developers to reason about execution paths, control flow, and system states that are otherwise difficult to observe in deployed environments. Analyzing log data constitutes an important form of automated program comprehension based on runtime software artifacts. The rich information in log data enables a wide range of software operations and maintenance tasks, such as detecting system anomalies [13, 22, 43], predicting failures [4, 5, 26], ensuring application security [32, 36], and diagnosing errors [1, 14, 24]. To facilitate these downstream tasks, the first and foremost step is *log parsing*, which parses raw log messages into structured format [12]. The structured log data from parsing allows the adoption of various machine learning (ML), deep learning (DL), and large language models (LLMs) techniques for effective downstream analysis tasks.

Specifically, log parsing is the task of converting each log entry into a specific log event/template associated with key parameters. As shown in Figure 1, each log entry is printed by a logging statement in the source code to record a specific system event. A log entry usually contains a header that is automatically produced by the logging framework and includes information such as component and verbosity level. In the figure, the header consists of the date ("`2025-09-17`"), time ("`20:10:46`"), level ("`TRACE`"), and component generating the log entry ("`BlockManager`"); it can be easily extracted through regular expression matching. The log message (or log content) typically consists of two parts: (1) *Template* - constant strings (or keywords) describing the system events; (2) *Parameters* - dynamic variables, which vary during runtime and reflect system runtime information. In the figure, the log message has a template, "`Task <*> downgrading write lock for <*>`", with five keywords and two parameters (i.e., "`25236`" and "`rdd_3_52_6`").

As the volume of log data grows exponentially with the increasing complexity of software systems [31], there have been tremendous efforts towards achieving the goal of automated log parsing. Since the source code is generally inaccessible during system operations and maintenance, existing log parsing methods propose to leverage *syntactic* and *semantic* patterns of logs to identify and separate static text and dynamic variables. *Syntax-based* log parsers [2, 6, 11, 17] utilize specific features or heuristics (e.g., token count, frequency, and position) to extract the constant parts of log messages as templates. In contrast, *semantic-based* log parsers [15, 23, 28] leverage pre-trained language models or large language models to recognize dynamic variables based on their semantic differences from constant keywords. The common step of existing log parsers is *segmentation*, which splits a log message

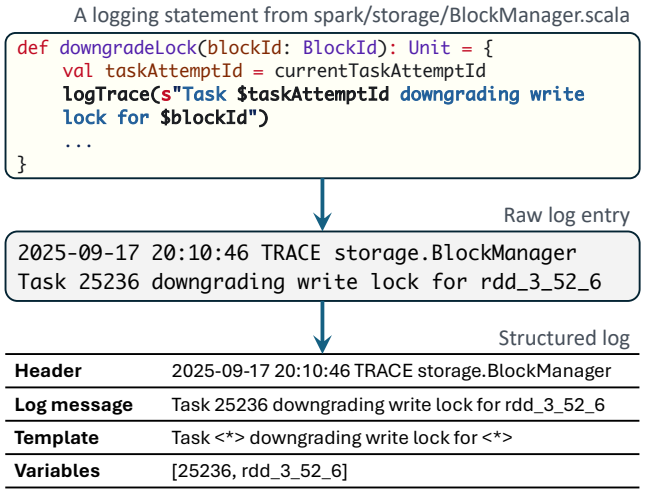


Figure 1: An illustrative example of log parsing

into smaller units called "tokens". These tokens are then analyzed to identify the structure and components of the log message (e.g., keywords and parameters) based on their syntactic and semantic differences. Despite the progress, existing log parsers still suffer from unsatisfactory accuracy [16, 18], which may significantly affect the follow-up analysis such as log-based anomaly detection [8, 19]. We have identified two main limitations that hinder the effectiveness of existing log parsing methods:

- **Under-segmentation**: Most log parsers leverage a fixed, predefined set of delimiters to separate a log message into a set of tokens. For example, Drain [11] and Logram [2] use whitespace characters (i.e., *space* and *tab*) as the delimiter. Such approaches are insufficient to handle the heterogeneity of logging formats in practice, which may lead to the under-segmentation issue where multiple parameters or keywords are grouped together into a single token. Specializing the set of delimiters for each dataset can alleviate the under-segmentation issue to some extent but requires significant manual effort and domain knowledge [7].
- **Over-segmentation**: On the one hand, using too many delimiters for syntax-based log parsers may lead to the over-segmentation issue, which fragments meaningful units in log messages. Defining a proper set of delimiters for each dataset is a non-trivial task due to the vast search space of delimiter combinations. On the other hand, to relax the dependency on predefined delimiters, some semantic-based log parsers (e.g., LogPPT [23] and LLMParser [29]) leverage pre-trained tokenizers from large language models (e.g., "`RoBERTa`", "`LLaMA`") to tokenize log messages. However, these tokenizers are designed for natural language and may split technical terms or parameters into multiple sub-tokens, resulting in over-segmentation.

Achieving accurate segmentation of log messages is vital for effective log parsing, especially in online parsing scenarios where log messages are processed in a streaming manner. To address the above limitations, in this paper, we propose SCLog, a novel **S**yntax- and **C**ontextual-aware segmentation approach for **Log** parsing. SCLog is designed with a two-fold process to uniformly segment log messages into fine-grained tokens. In the first step, SCLog preprocesses log messages by abstracting easily recognizable parameters using regular expressions. Then, it leverages a predefined set of common delimiters (e.g., whitespace characters, commas, semicolons, vertical bars, and brackets) to segment log messages into coarse-grained tokens. In the second step, SCLog further refines the segmentation of log messages by mining structural patterns of tokens based on their surrounding contexts to either merge or split the coarse-grained tokens dynamically. The proposed context-aware segmentation can be easily integrated with existing log parsers to improve their parsing accuracy without modifying their core algorithms. Additionally, we propose a context-aware similarity measure to further enhance the integration of SCLog with existing log parsers. Experimental results on widely-used, large-scale Loghub-2.0 datasets demonstrate that SCLog significantly improves the parsing accuracy and robustness of representative log parsers across diverse datasets with minimal runtime overhead.

The main contributions of this paper are summarized as follows:

(1) We conduct an empirical study to investigate the impact of log message segmentation on log parsing and identify the under-segmentation and over-segmentation issues in existing log parsers that hinder their effectiveness.
(2) We propose SCLog, a novel universal segmentation approach for log parsing that leverages both syntax-based heuristics and context-aware structural patterns to achieve accurate segmentation of log messages.
(3) We evaluate SCLog on widely-used, large-scale Loghub-2.0 datasets. The results demonstrate that SCLog significantly outperforms state-of-the-art log parsers in terms of parsing accuracy and robustness across diverse datasets.

2 Background

2.1 Preliminaries

We first introduce the key concepts used in the rest of the paper.

Log Message and Token. *A log message L is a sequence of tokens, i.e., $L = [t_1, t_2, \ldots, t_n]$, where each token t_i is a contiguous substring of L separated by delimiters.*

For instance, considering the log entry in Figure 1, the log message is "`Task 25236 downgrading write lock for rdd_3_52_6`", which can be segmented into seven tokens: "`Task`", "`25236`", "`downgrading`", "`write`", "`lock`", "`for`", "`rdd_3_52_6`".

Delimiter. *A delimiter is a character or a sequence of characters that separates tokens in a log message.*

For example, in the log message "`Task 25236 downgrading write lock for rdd_3_52_6`", the whitespace character is used as the delimiter to separate tokens.

Structural Pattern. *A structural pattern of a token t_i refers to its structural characteristics (e.g., length, character types, position).*

For example, considering the token "`rdd_3_52_6`" in the log message in Figure 1, some of its structural patterns include: (1) *length:* 10 characters; (2) *format:* "`F_F_F_F`"; and (3) *prefix:* "`rdd_`".

Context of a Token. *The context of a token t_i in a log message L is defined as its structural patterns along with the structural patterns of its neighbors.*

2.2 Log Parsing

Log parsing is one of the first steps for log analysis tasks [44]. It extracts the static log template parts and the corresponding dynamic parameters/variables from free-text raw log messages. The most straightforward way of performing log parsing relies on handcrafted regular expressions or grok patterns to extract log templates and parameters [44]. However, manually writing regular expressions to parse a huge volume of logs is time-consuming and error-prone [44]. Some studies [34, 40] extract the log templates from logging statements in the source code to compose regular expressions for log parsing. However, it is not applicable in practice since the source code is often unavailable, especially for third-party libraries [44]. To achieve the goal of automated log parsing, many data-driven approaches have been proposed in literature to parse logs without the need for source code or predefined patterns [12], which can be mainly categorized into two groups: *syntax-based* and *semantic-based* log parsers. Figure 2 illustrates a typical workflow of log parsing.

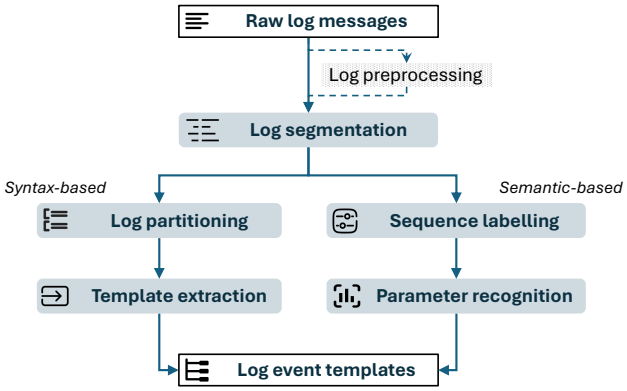


Figure 2: A typical workflow of log parsing

Syntax-based log parsers [2, 11, 41] assume that log templates inherit some common patterns which emerge constantly across the entire log dataset. They employ (1) *frequent pattern mining* to find constantly occurring patterns as log templates [2, 9], (2) *clustering* to group similar logs and extract the constant parts as templates [17, 39], or (3) *heuristics* to extract common templates based on unique characteristics of log messages [6, 11]. For example, Logram [2] finds frequent n-gram patterns that emerge constantly across the entire log dataset as templates. Drain [11] assumes that all log parameters within specific templates possess an identical number of tokens and uses token count and token position as heuristics to parse logs.

Semantic-based log parsers [15, 23, 28] leverage pre-trained language models or large language models to capture the semantic meanings of log messages and semantically differentiate log parameters from constant keywords. For example, UniParser [28] trains a bi-directional LSTM model to learn the semantic representations of log messages and identify parameters based on their semantic differences from constant keywords. LogPPT [23] uses a pre-trained `RoBERTa` model to predict parameter tokens based on their semantic meanings via a novel prompt-tuning approach. LLMParser [29] uses large language models (e.g., `T5`, `LLaMA`, and `ChatGLM`) for log parsing via few-shot tuning.

To evaluate the effectiveness of log parsers, several metrics have been proposed in literature [16, 18], with *Group Accuracy* (GA) and *Parsing Accuracy* (PA) being the most widely used ones. Grouping accuracy and its variants (e.g., F1-score of Group Accuracy - FGA) evaluate whether log messages sharing the same template also share the template in the ground truth [18]. The parsing accuracy family (e.g., F1-score of Template Accuracy - FTA) estimates the number of templates that are identical to the oracle [16]. In the rest of the paper, we leverage four widely-used metrics, i.e., GA, PA, FGA, and FTA, to evaluate the effectiveness of log parsers. Details of these metrics can be found in Section 5.1.2.

2.3 Log Segmentation

Given a log message L, segmentation splits L into a sequence of fine-grained tokens $[t_1, t_2, \ldots, t_n]$. These tokens are then analyzed to identify the structure and components of the log message (e.g., keywords and parameters) based on their syntactic or semantic features. For example, the log message in Figure 1, i.e., "`Task 25236 downgrading write lock for rdd_3_52_6`", can be segmented into seven tokens: "`Task`", "`25236`", "`downgrading`", "`write`", "`lock`", "`for`", and "`rdd_3_52_6`". Based on these tokens, log parsers can identify the log template as "`Task <*> downgrading write lock for <*>`" associated with the parameters as "`25236`" and "`rdd_3_52_6`".

3 Motivation: An Empirical Study of Log Segmentation

The segmentation quality of log messages has a significant impact on the effectiveness of log parsing, which has been overlooked by existing log parsers. In this section, we empirically study this impact and identify the challenges of log message segmentation.

3.1 Impact of Segmentation on Log Parsing

In this section, we conduct an empirical study to investigate the impact of log message segmentation on the effectiveness of log parsing. Specifically, we evaluate the parsing accuracy of two representative log parsers: Drain [11] (syntax-based) and LogPPT [23] (semantic-based) using different segmentation strategies on the widely-used *Loghub-2k* benchmark [44] consisting of 16 log datasets from various systems (e.g., distributed systems, supercomputers, operating systems). Apart from the default segmentation strategies used by these log parsers (i.e., whitespace characters for Drain and pre-trained `RoBERTa` tokenizer [27] for LogPPT), we evaluate two additional segmentation strategies: (1) using a fixed set of common delimiters (i.e., whitespace characters, equal sign, comma, colon, and semicolon) to segment log messages; (2) using dataset-specific delimiters borrowed from previous studies [7, 41] to segment log messages. Table 1 shows the average performance difference in the four metrics of Drain and LogPPT using different segmentation strategies, and presents the differences in *percentage points (pp)*.

From the results, we can see that using a fixed set of common delimiters for segmentation (i.e., in the case of Drain_C and LogPPT_C) can improve the parsing accuracy of both log parsers compared to their default segmentation strategies. Specifically, Drain_C achieves

Table 1: Impact of segmentation strategies on log parsing

	Average GA	Average FGA	Average PA	Average FTA
Drain	0.841	0.763	0.395	0.341
Drain$_{\mathcal{C}}$	0.863 ↑2.2 pp	0.794 ↑3.1 pp	0.474 ↑7.9 pp	0.365 ↑2.4 pp
Drain$_{\mathcal{D}}$	0.847 ↑0.6 pp	0.798 ↑3.5 pp	0.390 ↓0.5 pp	0.345 ↑0.4 pp
LogPPT	0.759	0.825	0.749	0.641
LogPPT$_{\mathcal{C}}$	0.787 ↑2.8 pp	0.839 ↑1.4 pp	0.696 ↓5.3 pp	0.628 ↓1.3 pp
LogPPT$_{\mathcal{D}}$	0.819 ↑6.0 pp	0.833 ↑0.8 pp	0.448 ↓24.8 pp	0.492 ↓14.9 pp

$\mathcal{C}$: using a fixed set of common delimiters for segmentation
$\mathcal{D}$: using dataset-specific delimiters for segmentation

an average GA and PA of 0.863 and 0.474, which are 2.2 *pp* and 7.9 *pp* higher than those of Drain with whitespace segmentation. In contrast, using dataset-specific delimiters for segmentation (i.e., Drain$_{\mathcal{D}}$ and LogPPT$_{\mathcal{D}}$) does not consistently improve the parsing accuracy of both log parsers. For example, Drain$_{\mathcal{D}}$ achieves an average GA and PA of 0.847 and 0.390, which are only 0.6 *pp* higher and 0.5 *pp* lower than those of Drain with whitespace segmentation. These results indicate that the segmentation quality of log messages has a significant impact on the effectiveness of log parsing. Using more delimiters for segmentation could potentially improve the parsing accuracy of existing log parsers.

To further evaluate the impact of segmentation on log parsing, we analyze the differences in parsing accuracy of Drain using different segmentation strategies on each individual dataset. Figures 3 and 4 show the parsing accuracy of Drain and LogPPT using three segmentation strategies (i.e., default, common delimiters, and dataset-specific delimiters) on each dataset.

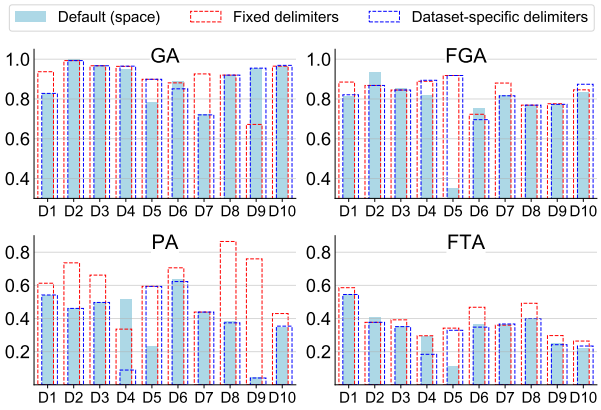


Figure 3: Impact of segmentation on log parsing with Drain

We can see that, on most datasets, using a fixed set of common delimiters for segmentation (in the case of Drain$_{\mathcal{C}}$) can consistently improve the parsing accuracy. For example, on Android (D1), HealthApp (D5), and OpenSSH (D7), using common delimiters for segmentation outperforms whitespace segmentation by up to 56 *pp* (FGA). However, this is not the case for segmentation with dataset-specific delimiters (i.e., Drain$_{\mathcal{D}}$). In contrast, on some datasets (e.g., Hadoop (D4) and HPC (D6)), using more delimiters actually drops the parsing accuracy compared to whitespace segmentation. The reason is that naively using too many delimiters for segmentation

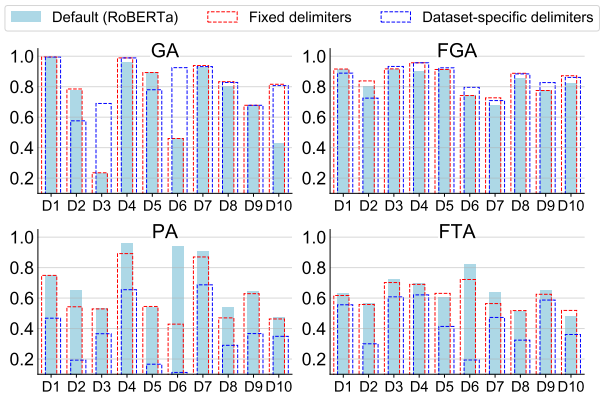


Figure 4: Impact of segmentation on log parsing with LogPPT

may fragment one meaningful unit in log messages into multiple tokens, leading to structural changes in log messages and thus impairing the parsing accuracy. In the next section, we discuss the challenges of log message segmentation in detail.

> **Finding 1**: The segmentation quality of log messages has a significant impact on the accuracy of log parsing. Using more delimiters for segmentation could potentially improve the parsing accuracy of existing log parsers yet it is non-trivial to select the optimal combination from a vast search space of delimiter combinations.

3.2 Failure Case Analysis of Log Segmentation

In this section, we analyze the failure cases of log message segmentation and identify two main challenges:

3.2.1 Under-segmentation. Under-segmentation occurs when a log message is not segmented into sufficiently fine-grained tokens, leading to the merging of multiple distinct components into a single token. Figure 5 shows an example of under-segmentation from the Linux log dataset. In this example, we present one scenario where under-segmentation occurs and another scenario where it does not occur for comparison.

We can see that when using the blank space as the delimiter (top half of the figure), the two log messages are segmented into five tokens; all of the tokens are different, suggesting a low similarity value. This leads to the incorrect grouping of the tokens, resulting in the two log messages being assigned to two different templates $\ddot{\mathcal{T}}$ and $\overline{\mathcal{T}}$, shown at the very top of the figure. On the other hand, when using more delimiters (such as blank space, equal sign, and parentheses in this example, as shown in the bottom half of the figure) for segmentation, the two log messages are segmented into more fine-grained tokens and have a higher similarity value of 0.5, i.e., five common tokens out of ten. This higher similarity value leads to the correct grouping, resulting in the two log messages being assigned to the same template $\mathcal{T}$: "`audit(<*>): item=<*> name=<*> inode=<*> dev=<*>`". Naively using more delimiters for segmentation can alleviate the under-segmentation issue to some extent. However, the bottleneck is that there are many delimiters to choose from in practice, and it is non-trivial to select

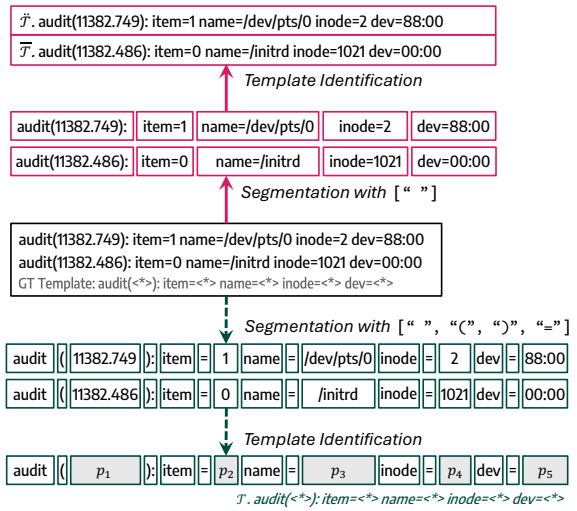
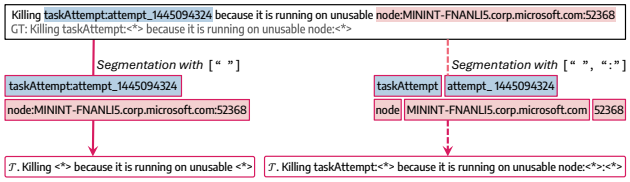


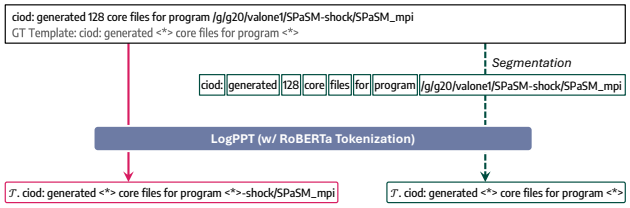
Figure 5: An illustration of under-segmentation (green: correct parsing; red: incorrect parsing)

the optimal combination from a vast search space of delimiter combinations. Moreover, using too many delimiters may lead to the over-segmentation issue, which we discuss next.

3.2.2 Over-segmentation. Over-segmentation occurs when a log message is segmented into excessively fine-grained tokens, leading to the fragmentation of meaningful components into multiple tokens. Figure 6 shows an example of over-segmentation, occurring both with syntax-based and semantic-based log parsers.



a) Over-segmentation with syntax-based log parsing



b) Over-segmentation with semantic-based log parsing

Figure 6: An illustration of over-segmentation (green: correct parsing; red: incorrect parsing)

From Figure 6(a), we can see that, given a single log message, the use of delimiters should be flexible based on the context of each token. For instance, using whitespace as the delimiter fails to segment two tokens, i.e., "`taskAttempt:attempt_1445094324`" and "`node:MININT-FNANLI5.corp.microsoft.com:52368`". These tokens, in fact, contain two distinct keywords (i.e., "`taskAttempt`" and "`node`") that should be separated by the colon delimiter for correct parsing. However, naively using both whitespace and colon as delimiters for this example leads to the over-segmentation issue of the second token, which is fragmented into three sub-tokens (i.e., "`node`", "`MININT-FNANLI5.corp.microsoft.com`", "`52368`"). This over-segmentation makes log parsers fail to extract the correct template. This scenario, which occurs frequently in real-world log datasets (e.g., Hadoop, HPC), highlights the difficulty of defining a proper set of delimiters for log parsers due to the context-dependent nature of log messages.

Another scenario of over-segmentation occurs within semantic-based log parsers, as shown in Figure 6(b). We can see that when using the pre-trained `RoBERTa` tokenizer for segmentation, the last token "`/g/g20/valone1/SPaSM-shock/SPaSM_mpi`" is over-fragmented into multiple sub-tokens, resulting in an incorrect template. Similar to the previous scenario, this over-segmentation issue makes LogPPT fail to identify certain tokens as parameters due to their semantic similarity to constant keywords (e.g., "`shock`"). On the other hand, when we add an intermediate segmentation step using whitespace before using LogPPT (with the `RoBERTa` tokenizer), the log message is correctly segmented into eight tokens, leading to the correct template. This example highlights that solely relying on pre-trained tokenizers from language models for segmentation may lead to over-segmentation, which impairs the parsing accuracy of semantic-based log parsers.

> **Finding 2**: Segmentation issues in log parsing mainly arise from two challenges: (1) under-segmentation, where multiple parameters or keywords are grouped together into a single token due to the heterogeneity of logging formats; (2) over-segmentation, where meaningful units in log messages are fragmented into multiple tokens due to the use of too many delimiters.

4 Approach

In this paper, to address the limitations of existing log parsing methods in terms of log segmentation, we propose SCLog, a simple yet effective universal segmentation approach for log parsing. We hypothesize that although log messages exhibit diverse formats and structures across different systems, they often share common syntactic features and structural patterns due to the nature of recording values of runtime variables. Therefore, by mining such patterns, we can accurately segment log messages into fine-grained tokens that facilitate effective log parsing. To this end, we design SCLog using a two-fold process to segment log messages into fine-grained tokens. First, SCLog leverages a commonly-used set of syntax-based heuristics to segment log messages into coarse-grained tokens (§ 4.1). Then, to further refine the segmentation of log messages, SCLog mines several structural patterns of tokens based on their surrounding contexts to either merge or split the coarse-grained tokens dynamically (§ 4.2). The intuitive idea is that log messages often exhibit repetitive structural patterns to record runtime information, and these patterns can provide valuable clues for identifying optimal delimiters for each token. By analyzing the contexts of tokens within log messages, SCLog can effectively capture these structural patterns and use them to guide the segmentation process. Additionally, we propose a context-aware similarity measure to enhance

the integration of SCLog with existing log parsers (§ 4.3). This measure allows SCLog to adapt its segmentation strategy based on the specific characteristics of the log parser it is integrated with to improve the parsing accuracy.

4.1 Coarse-grained Segmentation

The first step of SCLog is to segment a log message into coarse-grained tokens using a set of syntax-based heuristics. This step aims to quickly identify potential token boundaries based on common delimiters used in log messages. The coarse-grained segmentation serves as a foundation for the subsequent context-aware refinement process. In this step, SCLog simply segments the input log message using a predefined set of common delimiters, including whitespace characters, commas, semicolons, vertical bars, and brackets. These delimiters are frequently used in logging practices to separate different components of log messages [7]. By applying these delimiters, SCLog generates an initial set of coarse-grained tokens that capture the main structure of the log message. We remark this step may still result in under-segmentation or over-segmentation due to the limitations of fixed delimiters. Therefore, the next step focuses on refining the segmentation using context-aware structural patterns.

4.2 Context-aware Segmentation

After obtaining the coarse-grained tokens from the previous step, SCLog further refines the segmentation by analyzing the structural patterns of tokens based on their surrounding contexts. This step aims to (1) merge tokens that are likely to belong to the same parameter or keyword, and (2) split tokens that contain multiple parameters or keywords. To this end, we design two operations on each coarse-grained token, namely *token folding* and *token splitting*, which are guided by the context-aware structural patterns.

4.2.1 Token Folding. A log message may contain repetitive variables that occur consecutively, such as IDs and IP addresses. These variables are often separated into multiple tokens during the coarse-grained segmentation step due to the presence of delimiters, which can lead to over-segmentation and significant differences in log lengths, resulting in failing to group similar logs together [7]. For example, the following two log messages from HDFS[1]:

```
BLOCK* ask 10.250.17.177 to delete blk_857 blk_912
BLOCK* ask 10.250.10.213 to delete blk_857 blk_912 blk_194
```

contains consecutive "`blk_<id>`" variables corresponding to different data blocks in HDFS. While the number of blocks may vary, these log messages share the same template (i.e., "`BLOCK* ask <*> to delete <*>`"). However, during the coarse-grained segmentation step, these log messages may be segmented into different numbers of tokens due to the varying number of blocks, leading to failure in grouping them together.

To address this issue, we propose a *token folding* operation that merges consecutive tokens exhibiting similar structural patterns into a single token. In particular, this operation considers two structural patterns of tokens: *(1) the set of special characters* (i.e., "_" in the above example) and *(2) the position of the shared pattern* (i.e., the prefix "`blk_`" in the above example). If two or more consecutive tokens share such characteristics, we merge them into a single token. Algorithm 1 illustrates this process in detail. Specifically, we iterate through each coarse-grained token and extract its structural patterns (lines 2–14). If the current token shares the same patterns as the previous token(s), we group them together (lines 6–9). Otherwise, we merge the tokens in the current group and add them to the refined token list (lines 11–14). Finally, we return the refined tokens after processing all coarse-grained tokens (lines 15–17).

Algorithm 1: Token Folding Algorithm

Input: Coarse-grained tokens $T = \{t_1, t_2, \ldots, t_n\}$; set of special characters S
Output: Refined tokens T' with merged similar tokens

1 Initialize an empty list T' and *current_group*
2 **foreach** $t \in T$ **do**
3 **if** *current_group is empty* **then**
4 Add t to *current_group*
5 **else**
  /* Extract patterns from t
  ch_t: list of special characters in t
  pf_t: prefix before the first special character */
6 $ch_t \leftarrow \{\, t_i \mid i \in [1, |t|],\ t_i \in S \,\}$
7 $pf_t \leftarrow \textbf{SPLIT}(t, ch_t[0])[0]$
8 **if** $ch_t == ch_g$ ***and*** $pf_t == pf_g$ **then**
9 Add t to *current_group*
10 **else**
11 Merge tokens in *current_group* and add to T'
12 Reset *current_group* and add t
  /* Update patterns for *current_group* */
13 $(ch_g, pf_g) \leftarrow (ch_t, pf_t)$
14 **if** *current_group is not empty* **then**
15 Merge tokens in *current_group* and add to T'
16 **return** T'

4.2.2 Token Splitting. In contrast to token folding, some coarse-grained tokens may contain multiple individual units that should be separated for accurate log parsing. This phenomenon occurs particularly commonly in logging practices where developers concatenate keywords and parameters without using uniform, proper delimiters. We use the following log messages from Hadoop[1] to further discuss this issue and illustrate our proposed solution:

```
MergerManager: memoryLimit=130652568, maxSingleShuffleLimit
↪=32663142, mergeThreshold=86230696, ioSortFactor=10
Killing taskAttempt:attempt_1445094324 because it is running
↪ on unusable node:MININT-FNANLI5.corp.microsoft.com:52368
```

In the above log messages, some coarse-grained tokens (e.g., "`memoryLimit=130652568`") contain multiple individual units that should be separated for accurate log parsing. However, simply using a fixed set of delimiters may not be sufficient to handle such cases, as even in the same dataset, different log messages may use different separators (in this case, "=" and ":") to concatenate keywords and parameters. Some might argue that using more delimiters during the coarse-grained segmentation step can alleviate this issue. However, as discussed in Section 3.2.2, different tokens in the same log message may require different delimiters for accurate segmentation (e.g., the colon ":" in the first log message). Therefore, we propose a *token splitting* operation that dynamically identifies optimal delimiters for each token based on its surrounding contexts.

[1]We shorten numbers in the log messages for better readability.

We observe that, given a log message, commonalities and variabilities often coexist among the structural patterns of its tokens. For example, in the above log messages, the pattern "F=F" appears multiple times, where the prefix varies (e.g., "memoryLimit", "maxSingleShuffleLimit"), but the delimiter "=" remains consistent. This reflects the nature of logging practices, where developers often record several *distinct* runtime variables within a single logging statement [17]. Therefore, by analyzing the contexts of tokens within a log message, we can effectively capture these structural patterns and use them to guide the token splitting process. To this end, we define three structural patterns of tokens for the splitting operation: *(1) the first delimiter character and its position within the token, (2) the prefix before the delimiter*, and *(3) the suffix after the delimiter*. After extracting these patterns for each token, SCLog identifies potential delimiters that frequently appear in similar contexts and uses them to split the tokens accordingly. Specifically, we first group tokens based on their delimiter character and its position. Then, within each group that has multiple tokens, we split those tokens that have different prefixes and non-empty suffixes using the identified delimiter. For those groups with only one token, we do not perform any splitting to avoid unnecessary fragmentation as there is insufficient evidence to support the splitting operation based on the aforementioned observations. We consider several common delimiters (e.g., "=", ":", "-") for this operation to cover a wide range of logging practices [7].

Taking the first log message as an example, SCLog identifies four tokens containing the delimiter "=" at the same position with different prefixes (e.g., "memoryLimit", "maxSingleShuffleLimit") and non-empty suffixes. Therefore, it splits these tokens into individual units based on the delimiter "=". Algorithm 2 formally illustrates this process in detail. Specifically, we iterate through each refined token from the previous step, §4.2.1, and extract its structural patterns (lines 2–12). After collecting all patterns, we check if there are multiple unique prefixes for the tokens sharing the same delimiter (lines 13–14). If so, we split those tokens using the identified delimiter and return the set of fine-grained tokens (lines 16–17). Otherwise, we return the original refined tokens without any splitting (line 15). We invoke this algorithm for each considered delimiter to ensure comprehensive token splitting.

Algorithm 2: Token Splitting Algorithm

Input: Refined tokens from Algorithm 1, T'; delimiter d
Output: Fine-grained tokens $\overline{T}$ with context-aware splits

```
1  Initialize an empty list patterns
   /* Extract structural patterns from tokens                    */
2  foreach t_i ∈ T' do
3      if d ∉ t_i then
4          continue
5      st ← SPLIT(t_i, d)        // Split token by d (keep delimiter)
6      pos ← index of d in st
7      if multiple delimiters in st then
8          Merge all tokens from pos + 1 to end
9      pf ← st[pos − 1] if pos > 0 else ∅
10     sf ← st[pos + 1] if pos < |st| − 1 else ∅
11     if sf ≠ ∅ and pf ≠ ∅ then
12         Append (t_i, pos, pf, sf, st) to patterns
   /* Apply context-aware splitting                              */
13 prefixes ← unique({pf | (t, pos, pf, sf, st) ∈ patterns})
14 if |prefixes| < 2 then
15     return T'        // All tokens have the same prefix: no split
16 Split all tokens in patterns using delimiter d and update T' to get
     the set of fine-grained tokens, T̄
17 return T̄
```

4.3 Context-aware Similarity Measure

In addition to the segmentation process, we further propose a context-aware similarity measure to enhance the integration of SCLog with existing log parsers. The idea is to leverage the similarities in both the tokens and mined delimiters to compute a more accurate similarity score between log messages, which can help improve the grouping accuracy of log parsers. Specifically, we define the context-aware similarity between two log messages $P = \{p_1, p_2, \ldots, p_n\}$ and $Q = \{q_1, q_2, \ldots, q_n\}$ as follows:

$$
\begin{aligned}
SIM(P,Q) &= sim_{token}(P,Q) + \lambda \times sim_{delim}(P,Q) \\
sim_{token}(P,Q) &= \frac{\sum_{i=1}^{n} score(p_i,q_i), \forall p_i \text{ is token}}{number\_of\_tokens\_in\_P} \\
sim_{delim}(P,Q) &= \frac{\sum_{i=1}^{n} score(p_i,q_i), \forall p_i \text{ is delimiter}}{number\_of\_delimiters\_in\_P}
\end{aligned} \tag{1}
$$

where $score(p_i, q_i) = 1$ if $p_i = q_i$ else 0. Here, $sim_{token}(P, Q)$ measures the similarity based on the tokens of the log messages, while $sim_{delim}(P, Q)$ measures the similarity based on the delimiters mined during the context-aware segmentation process. λ is a weighting factor that balances the contributions of token similarity and delimiter similarity. By incorporating both aspects into the similarity measure, we can capture more nuanced relationships between log messages, leading to improved grouping ability in log parsing.

5 Evaluation

In this section, we report on the experimental evaluation of SCLog. We aim to assess the effectiveness and efficiency of SCLog when integrated with existing log parsers. Specifically, we use the following research questions (RQs) to guide our evaluation:

- **RQ1. What is the performance of SCLog when combined with log parsers?** In this RQ, we evaluate the parsing accuracy of several representative log parsers enhanced with SCLog on widely-used log datasets. We also evaluate the robustness of SCLog across diverse datasets and analyze its runtime overhead.
- **RQ2. What is the contribution of each component in SCLog to the overall performance?** In this RQ, we conduct ablation studies to analyze the impact of each component in SCLog on the parsing performance. We also investigate the performance of SCLog under different experimental settings.

5.1 Benchmark and Settings

5.1.1 Datasets. We conduct experiments on 14 large-scale public log datasets from the Loghub-2.0 benchmark [16]. This benchmark consists of log data of 14 different systems, including distributed systems, supercomputers, operating systems, mobile systems, server

applications, and standalone software. Each dataset in the benchmark comprises 3.6 million log messages on average. In total, the benchmark contains approximately 3,500 unique log templates.

5.1.2 Evaluation Metrics. Following recent studies [18, 28, 35, 44], we use four metrics in our evaluation, including:

- **Group Accuracy (GA)**: Group Accuracy [44] is the most commonly used metric for log parsing. Group Accuracy considers template identification as a clustering process in which log messages with different log events are clustered into different groups [18]. The GA metric is defined as the ratio of "correctly parsed" log messages over the total number of log messages, where a log message is considered "correctly parsed" if and only if it is grouped with other log messages consistent with the ground truth.
- **F1 score of Grouping Accuracy (FGA)**: FGA [15] is a template-level metric that measures the ratio of correctly grouped templates. It is computed as the harmonic mean of precision and recall of grouping accuracy, where the template is considered as correct if log messages of the predicted template have the same group of log messages as the oracle.
- **Parsing Accuracy (PA):** Parsing Accuracy (or Message-Level Accuracy [28]) is defined as the ratio of "correctly parsed" log messages over the total number of log messages, where a log message is considered to be "correctly parsed" if and only if every token of the log message is correctly identified as template or variable. This metric is much stricter than Group Accuracy since any incorrectly parsed token will lead to the wrong parsing result for the whole log message.
- **F1 score of Template Accuracy (FTA)**: Similar to FGA, FTA [18] is a template-level accuracy computed as the harmonic mean of precision and recall of Template Accuracy. A template is regarded as correct if and only if it satisfies two requirements: log messages of the predicted template have the same group of log messages as the oracle, and all tokens of the template are the same as the oracle template. Given its definition, FTA is the most stringent and comprehensive metric among the four metrics.

5.1.3 Parser Selection. As the main contribution of this paper is a novel, universal segmentation approach for log parsing, we focus on evaluating the effectiveness of SCLog in improving the parsing accuracy of existing log parsers. Therefore, we select four representative log parsers as the base parsers to integrate with SCLog. Particularly, we select three syntax-based log parsers, including Drain [11] for heuristics-based parsing, AEL [17] for clustering-based parsing, and LFA [33] for frequent pattern mining-based parsing. In addition, we select one semantic-based log parser, LogPPT [23], which leverages pre-trained language models for log parsing. This selection covers typical log parsing techniques in the literature with state-of-the-art performance for each category. For a fair comparison, we use the same hyperparameter settings and preprocessing steps (with regular expressions from prior work [37]) for each base parser as reported in recent benchmarks [16, 37]. For LogPPT, we follow a recent study [25] to enhance its performance with optimal data sampling, prompt tuning, and caching mechanisms. Besides, as LogPPT induces randomness in its parsing process (i.e., data sampling and model training), we ran it five times for each dataset and reported the average performance to avoid randomness bias.

5.1.4 Environment Settings. We implement SCLog in Python 3.13. The experiments are conducted on a server equipped with an Intel Core Ultra 9 CPU @ 5.1 GHz and 156 GB of RAM, and an NVIDIA RTX 5880 GPU with 48 GB of memory (for LogPPT). We use $\lambda = 0.1$ (see Section 4.3) for the context-aware similarity measure.

5.2 RQ1: Performance of SCLog with Existing Log Parsers

5.2.1 Methodology. To answer RQ1, we evaluate the accuracy of the selected log parsers with and without SCLog on the Loghub-2.0 benchmark datasets. We combine SCLog with two variants of each base parser: (1) the original version of the base parser without any modification; (2) the improved version of the base parser that uses regular expressions to preprocess log messages before parsing, as suggested in recent studies [37]. As SCLog focuses on improving the segmentation of log messages, its integration should improve the parsing accuracy of both variants of the base parsers. We report the average performance of each parser regarding the four evaluation metrics across all datasets. In addition, we analyze the robustness of SCLog by reporting the parsing accuracy of each parser on individual datasets. Finally, we measure the runtime overhead introduced by SCLog when integrated with each base parser.

5.2.2 Results. We first report the average ***accuracy*** in terms of the GA, FGA, PA, and FTA metrics of each log parser with and without SCLog across all datasets in Table 2. We can observe that except for a slight decrease in GA for AEL, SCLog consistently improves the parsing accuracy of all base parsers across all metrics. Specifically, with Drain [11, 37] as the base parser, SCLog improves the average FGA, PA, and FTA by 9.4 *pp*, 11.8 *pp*, and 7.5 *pp* when no preprocessing is applied, and by 1.8 *pp*, 5.9 *pp*, and 2.6 *pp* when preprocessing is applied. Of all syntax-based log parsers, LFA [33, 37] benefits the most from SCLog with an improvement of up to 23.6 *pp* in PA and 10.2 *pp* in FTA. For the semantic-based log parser LogPPT [23, 25], SCLog also improves the average GA, PA, and FTA by 1.3 *pp*, 4.1 *pp*, and 2.4 *pp*, respectively. The experimental results indicate that SCLog can effectively enhance the segmentation of log messages, leading to significant improvements in parsing accuracy across various log parsers using major techniques (i.e., syntax-based with heuristics, clustering, frequent pattern mining, and semantic-based) in the literature. It is worth noting that the improvements in PA and FTA are more significant than those in GA and FGA, with FTA being the most stringent metric. This indicates that SCLog is particularly effective in improving the fine-grained parsing accuracy of log messages.

We next evaluate the performance of SCLog in terms of ***robustness*** across all log datasets. Figure 7 shows the accuracy distribution of each log parser with and without SCLog. The results demonstrate that SCLog consistently improves the parsing robustness of all base parsers across diverse datasets. It yields a notable increase in terms of the median PA value for all base parsers. For instance, with Drain as the base parser, SCLog increases the median FGA and FTA value from 0.460 to 0.663 and from 0.170 to 0.272, respectively. Similar trends are observed for other base parsers. Moreover,

Table 2: The average performance of log parsers with and without SCLog

	Average GA		Average FGA		Average PA		Average FTA	
	w/o SCLog	*w/ SCLog*	*w/o SCLog*	*w/ SCLog*	*w/o SCLog*	*w/ SCLog*	*w/o SCLog*	*w/ SCLog*
Drain-ori	0.810	0.840$_{\uparrow 3.0\ pp}$	0.490	0.584$_{\uparrow 9.4\ pp}$	0.409	0.527$_{\uparrow 11.8\ pp}$	0.225	0.300$_{\uparrow 7.5\ pp}$
Drain-pre	0.847	0.847$_{=}$	0.815	0.833$_{\uparrow 1.8\ pp}$	0.645	0.704$_{\uparrow 5.9\ pp}$	0.590	0.616$_{\uparrow 2.6\ pp}$
AEL-ori	0.767	0.754$_{\downarrow 1.3\ pp}$	0.523	0.580$_{\uparrow 5.7\ pp}$	0.393	0.500$_{\uparrow 10.7\ pp}$	0.225	0.287$_{\uparrow 6.2\ pp}$
AEL-pre	0.791	0.781$_{\downarrow 1.0\ pp}$	0.682	0.687$_{\uparrow 0.5\ pp}$	0.607	0.663$_{\uparrow 5.6\ pp}$	0.511	0.530$_{\uparrow 1.9\ pp}$
LFA-ori	0.596	0.603$_{\uparrow 0.7\ pp}$	0.507	0.529$_{\uparrow 2.2\ pp}$	0.239	0.299$_{\uparrow 6.0\ pp}$	0.098	0.139$_{\uparrow 4.1\ pp}$
LFA-pre	0.593	0.723$_{\uparrow 13.0\ pp}$	0.633	0.728$_{\uparrow 9.5\ pp}$	0.238	0.474$_{\uparrow 23.6\ pp}$	0.158	0.260$_{\uparrow 10.2\ pp}$
LogPPT	0.911	0.924$_{\uparrow 1.3\ pp}$	0.888	0.888$_{=}$	0.730	0.771$_{\uparrow 4.1\ pp}$	0.644	0.668$_{\uparrow 2.4\ pp}$

SCLog significantly reduces the variance in PA across datasets for all base parsers, indicating that it can effectively handle the heterogeneity of logging formats in different systems, which is crucial for practical applications of log parsing.

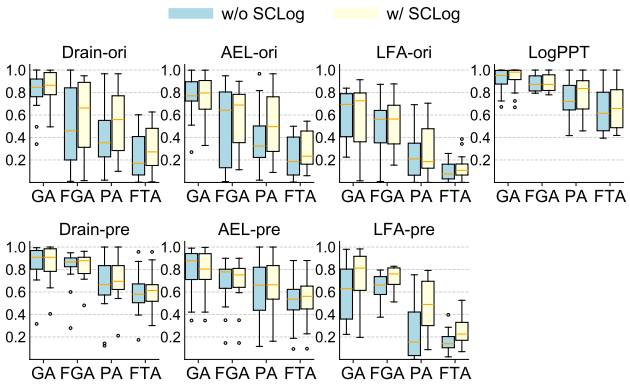


Figure 7: Parsing accuracy distribution of log parsers with and without SCLog

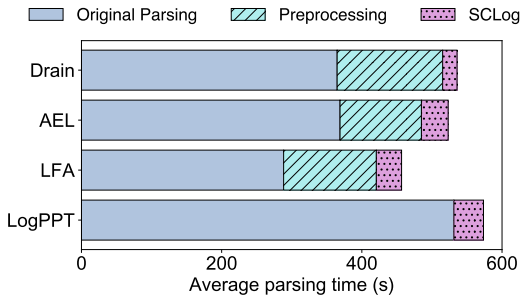


Figure 8: Runtime overhead of log parsers with and without SCLog

Lastly, we measure the ***efficiency*** of SCLog by analyzing the runtime overhead introduced when integrated with each base parser. Figure 8 shows the runtime of each log parser with and without SCLog across all datasets. In this figure, we report the original runtime of each base parser, the runtime with preprocessing, and the runtime with SCLog. The runtime overhead introduced by SCLog is relatively small compared to the total parsing time of each base parser. On average, SCLog introduces an additional overhead of 5.5% to 12.3% across all base parsers. Notably, the overhead is more pronounced for parsers with lower original runtimes, such as LFA, where SCLog introduces an overhead of 12.3%. It is worth noting that the overhead introduced by SCLog is less than that introduced by preprocessing steps for all base parsers, indicating that SCLog is an efficient solution for enhancing existing log parsers without incurring significant computational costs.

> **Answer to RQ1:** SCLog significantly improves the parsing accuracy of all base parsers across all evaluation metrics, with the most significant improvements observed in terms of FGA, PA, and FTA. The improvements are consistent across diverse datasets, demonstrating the robustness of SCLog. The runtime overhead introduced by SCLog is minimal, making it a practical solution for enhancing existing log parsers.

5.3 RQ2: Contribution of Each Component in SCLog

5.3.1 Methodology. To answer RQ2, we conduct ablation studies to analyze the contribution of each component in SCLog to the overall performance using the following variants:

- **SCLog$_{\mathcal{W}}$**: In this configuration, we only use whitespace characters as the delimiters for the initial segmentation step in SCLog. This allows us to assess the impact of using a predefined set of common delimiters versus only whitespace characters on the parsing accuracy.
- **SCLog$_{C}$**: In this configuration, we disable the context-aware segmentation component in SCLog. This allows us to assess the impact of context-aware segmentation on the parsing accuracy.
- **SCLog$_{\mathcal{S}}$**: In this configuration, we disable the context-aware similarity measure in SCLog. This allows us to evaluate the contribution of the context-aware similarity measure to the overall performance.
- **SCLog$_{\mathcal{D}}$**: In this configuration, we only use the predefined set of common delimiters for segmentation without any further refinement. This allows us to assess the effectiveness of the two-step segmentation process in SCLog.

We integrate each configuration of SCLog with the selected base parsers and evaluate their parsing accuracy on the Loghub-2.0 benchmark datasets. Due to space constraints, we only present the results of SCLog integrated with Drain (original) [11] and

LogPPT [23] in this RQ. The results for other base parsers are consistent and can be found in the supplementary material.

5.3.2 Results. Table 3 shows the average parsing accuracy of each configuration of SCLog when integrated with base parsers across all datasets. We can observe that each component in SCLog contributes positively to the overall performance. For example, without the coarse-grained segmentation component (i.e., SCLog$_{\mathcal{W}}$), the average FTA decreases by 7.6 *pp* and 3.4 *pp* when integrated with Drain and LogPPT, respectively. Without the context-aware segmentation component (i.e., SCLog$_{C}$), the average FTA decreases by 4.0 *pp* with LogPPT as the base parser. Without the context-aware similarity measure (i.e., SCLog$_{\mathcal{S}}$), we also observe a significant drop in parsing accuracy across all metrics for both base parsers. Finally, when only using the predefined set of common delimiters for segmentation (i.e., SCLog$_{\mathcal{D}}$), we observe the most significant decrease in parsing accuracy across all metrics for both base parsers.

Table 3: Performance of SCLog with different settings

	Average GA	Average FGA	Average PA	Average FTA
Drain $_{\text{SCLog}}$	0.840	0.584	0.527	0.300
- *SCLog*$_{\mathcal{W}}$	0.804$_{\downarrow 3.6\text{ pp}}$	0.486$_{\downarrow 9.8\text{ pp}}$	0.407$_{\downarrow 12.0\text{ pp}}$	0.224$_{\downarrow 7.6\text{ pp}}$
- *SCLog*$_{C}$	0.801$_{\downarrow 3.9\text{ pp}}$	0.568$_{\downarrow 1.6\text{ pp}}$	0.521$_{\downarrow 0.6\text{ pp}}$	0.286$_{\downarrow 1.4\text{ pp}}$
- *SCLog*$_{\mathcal{S}}$	0.797$_{\downarrow 4.3\text{ pp}}$	0.503$_{\downarrow 8.1\text{ pp}}$	0.516$_{\downarrow 1.1\text{ pp}}$	0.262$_{\downarrow 3.8\text{ pp}}$
- *SCLog*$_{\mathcal{D}}$	0.787$_{\downarrow 5.3\text{ pp}}$	0.539$_{\downarrow 4.5\text{ pp}}$	0.498$_{\downarrow 2.9\text{ pp}}$	0.257$_{\downarrow 4.3\text{ pp}}$
LogPPT $_{\text{SCLog}}$	0.924	0.888	0.771	0.668
- *SCLog*$_{\mathcal{W}}$	0.907$_{\downarrow 1.7\text{ pp}}$	0.877$_{\downarrow 1.1\text{ pp}}$	0.726$_{\downarrow 4.5\text{ pp}}$	0.634$_{\downarrow 3.4\text{ pp}}$
- *SCLog*$_{C}$	0.913$_{\downarrow 1.1\text{ pp}}$	0.882$_{\downarrow 0.6\text{ pp}}$	0.752$_{\downarrow 1.9\text{ pp}}$	0.628$_{\downarrow 4.0\text{ pp}}$
- *SCLog*$_{\mathcal{S}}$	0.889$_{\downarrow 3.5\text{ pp}}$	0.855$_{\downarrow 3.3\text{ pp}}$	0.759$_{\downarrow 1.2\text{ pp}}$	0.641$_{\downarrow 2.7\text{ pp}}$
- *SCLog*$_{\mathcal{D}}$	0.870$_{\downarrow 5.4\text{ pp}}$	0.832$_{\downarrow 5.6\text{ pp}}$	0.740$_{\downarrow 3.1\text{ pp}}$	0.620$_{\downarrow 4.8\text{ pp}}$

Answer to RQ2: Overall, each component in SCLog contributes positively to the parsing accuracy. The combination of coarse-grained segmentation, context-aware segmentation, and context-aware similarity measure is crucial for achieving optimal performance in log parsing.

6 Discussion

6.1 Prospects and Limitations of SCLog

6.1.1 Prospects. SCLog has demonstrated its effectiveness in enhancing the performance of existing log parsers through context-aware segmentation. Its adaptability to various log parsing algorithms and datasets highlights its potential as a universal segmentation approach. There are several reasons that make SCLog a promising solution for log parsing. First, SCLog's syntax-aware segmentation allows it to effectively handle diverse logging formats by leveraging a comprehensive set of delimiters. This adaptability is crucial in real-world scenarios where log formats can vary significantly across different systems and applications. Second, the context-aware segmentation strategy enables SCLog to dynamically identify optimal delimiters based on the structural patterns of tokens, which helps mitigate both under-segmentation and over-segmentation issues. This dynamic approach enhances the accuracy of log parsing by ensuring that tokens are segmented in a manner that reflects their true semantic meaning. Third, SCLog's modular design allows it to be easily integrated with various existing log parsers, making it a versatile tool that can enhance the performance of a wide range of log analysis systems.

6.1.2 Limitations. The main limitation of SCLog lies in its reliance on the mining of structural patterns within single log messages. In scenarios where log messages are extremely short or lack sufficient contextual information, SCLog may struggle to identify optimal delimiters, potentially leading to suboptimal segmentation and parsing accuracy. To address this limitation, we can consider mining inter-message structural patterns by analyzing multiple log messages together. Specifically, the current approach focuses on splitting a single log message based on its own structural patterns (§4.2.2) assuming that logging patterns exhibit variabilities for recording distinct runtime variables. In contrast, by examining a collection of log messages, we can identify common structural patterns where the same variables are logged across different messages, leading to commonalities that can inform better segmentation decisions. This collective analysis can help uncover patterns that may not be evident when considering individual log messages in isolation, thereby enhancing the segmentation process. To validate this idea, we conducted a preliminary experiment by extending SCLog to incorporate inter-message structural pattern mining, denoted as SCLog$_{I}$. We evaluated SCLog$_{I}$ using the AEL log parser. Table 4 shows the results.

Table 4: Parsing accuracy with inter-message structural pattern mining

	Average GA	Average FGA	Average PA	Average FTA
AEL$_{\text{SCLog}_I}$	0.756	0.609	0.531	0.304
AEL	0.767$_{\uparrow 1.1\text{ pp}}$	0.523$_{\downarrow 8.6\text{ pp}}$	0.393$_{\downarrow 11.8\text{ pp}}$	0.225$_{\downarrow 7.9\text{ pp}}$
AEL$_{\text{SCLog}}$	0.754$_{\downarrow 0.2\text{ pp}}$	0.580$_{\downarrow 2.9\text{ pp}}$	0.500$_{\downarrow 3.1\text{ pp}}$	0.287$_{\downarrow 1.7\text{ pp}}$

SCLog$_{I}$: SCLog with inter-message structural pattern mining

It can be observed that SCLog $_{I}$ achieves better parsing accuracy than SCLog, showcasing an improvement of 7.9 *pp* and 1.7 *pp* in Average FTA compared to AEL and AEL$_{\text{SCLog}}$. The preliminary results indicate the potential of inter-message structural pattern mining in further enhancing log parsing accuracy. However, this design unintentionally hinders the online parsing capability of SCLog, as it requires access to multiple log messages for effective segmentation compared to the current streaming fashion. Hence, more research is needed to explore efficient techniques for mining inter-message structural patterns while preserving the online parsing capability. In future work, we will systematically investigate this direction to further enhance the segmentation and parsing accuracy of SCLog.

6.2 Threats to Validity

While our study demonstrates promising results, several threats to validity should be considered as follows:

(1) **Data quality:** The quality of log data used in our experiments may affect the generalizability of our findings. In this paper, we used public log datasets [16] for our evaluation.

These datasets, although in large-scale, may contain errors in the ground truth labels. In future work, further validation on these datasets is needed to ensure the correctness of the ground truth labels.

(2) **Parameter settings:** The performance of SCLog may be influenced by the choice of parameters used in the base log parsers. To mitigate this threat, we used the default parameter settings recommended by the original authors of each log parser for a fair comparison. However, it is possible that different parameter settings could yield different results [3]. Future work could explore the sensitivity of SCLog to various parameter configurations.

(3) **Generalizability of SCLog:** While we evaluated SCLog with four state-of-the-art log parsers, there may be other log parsing algorithms that could benefit from SCLog's segmentation approach. For example, some log parsers [9, 38, 39] do not consider similarity measures in their parsing process, which may limit the effectiveness of SCLog's context-aware segmentation. Future work could explore the integration of SCLog with a broader range of log parsing algorithms to assess its generalizability.

(4) **Impact of SCLog on downstream tasks:** While our study focused on improving log parsing accuracy, it is important to assess the impact of SCLog on downstream log analysis tasks. While log mining is only effective when the parsing accuracy is high enough [10], there might be no strong correlation between parsing accuracy and the performance of downstream tasks such as anomaly detection [19]. Future work could investigate how the improved parsing accuracy achieved by SCLog translates to enhanced performance in various log analysis applications.

7 Related Work

Log parsing has been an active research area in the field of log analysis, with numerous approaches proposed to improve the accuracy and efficiency of log parsing. Traditional log parsers leverage data-driven techniques to analyze the syntactic structure of log messages and extract templates and parameters [2, 6, 11, 30], while recent advancements in language models have led to the development of semantic-based log parsing methods [15, 23, 28]. Despite the progress, automated log parsing remains non-trivial due to the diverse and complex nature of log data [16, 42].

7.1 Benchmarking Log Parsers

In order to understand the strengths and weaknesses of existing log parsers, several benchmarking studies have been conducted [3, 16]. Zhu et al. [44] evaluated 13 log parsers on Loghub-2k, which contains 16 datasets with 2,000 log messages each. Khan et al. [18] studied the appropriateness of evaluation metrics for log parsing and proposed guidelines for selecting suitable metrics. Following the guidelines, Jiang et al. [16] conducted a large-scale benchmarking study of 14 log parsers on Loghub-2.0, which contains 14 datasets with 3.6 million log messages per dataset on average. Dai et al. [3] investigated the impact of parameter settings on the performance of log parsers. These benchmarking studies provide valuable insights into the performance of existing log parsers and highlight the need for more robust and accurate log parsing techniques.

7.2 Improving Log Parsing Accuracy

While numerous log parsing techniques have been proposed, only a few target at uniformly improving the parsing accuracy of existing log parsers. Qin et al. [37] proposed a preprocessing framework that abstracts variable parts in log messages using a set of predefined regular expressions before applying existing log parsers. While this approach can improve parsing accuracy to some extent, it relies heavily on the quality of predefined regular expressions, which may not generalize well to diverse log formats as well as requires significant manual effort and domain knowledge to create and maintain. Fu et al. [7] investigated the use of fixed and dataset-specific delimiters for log message segmentation and their impact on log parsing accuracy. They proposed *Drain+*, an enhanced version of the Drain log parser that incorporates dataset-specific delimiters for segmentation. While *Drain+* shows improved parsing accuracy on certain datasets, it still relies on a fixed set of delimiters segmentation for all log messages within a dataset, which may not effectively address the segmentation issues identified in our study.

In contrast to these approaches, SCLog offers a universal, more dynamic, and context-aware segmentation strategy that combines syntax-based heuristics with context-aware analysis to effectively segment log messages. This allows SCLog to adaptively identify optimal delimiters for each log message, thereby addressing the limitations of under-segmentation and over-segmentation and significantly improving the parsing accuracy of existing log parsers across diverse datasets.

8 Conclusion

Log parsing is a foundation step to transform unstructured log messages into structured data for various downstream log analysis tasks. In this paper, we have empirically studied the impact of log message segmentation on the performance of log parsers. We identified two main limitations of existing log parsers, i.e., under-segmentation and over-segmentation, which significantly hinder their parsing accuracy. To address these limitations, we have proposed SCLog, a universal segmentation approach that combines syntax- and context-aware strategies to effectively segment log messages for log parsing. SCLog combines syntax-based heuristics and context-aware strategies to dynamically and effectively divide log messages into smaller units (tokens), enabling a more accurate parsing process. Extensive experiments with four state-of-the-art log parsers on large-scale, widely-used Loghub-2.0 datasets have demonstrated that SCLog significantly improves the parsing accuracy and robustness of existing log parsers across diverse datasets. Still, SCLog has some limitations, such as its reliance on the mining of structural patterns within single log messages, which may affect its performance in scenarios with short log messages.

In future work, we plan to explore approaches to mine inter-message structural patterns by considering multiple log messages together to further enhance the segmentation and parsing accuracy.
Data Availability: The source code of SCLog is available on Figshare [21] together with our experimental data [20].

Acknowledgments

Part of this work was supported by the Luxembourg National Research Fund (FNR), grant reference C22/IS/17373407/LOGODOR.

References

[1] Chaima Boufaied, Claudio Menghi, Domenico Bianculli, and Lionel Briand. 2023. Trace Diagnostics for Signal-based Temporal Properties. *IEEE Transactions on Software Engineering* 49, 5 (May 2023), 3131 – 3154. doi:10.1109/TSE.2023.3242588

[2] Hetong Dai, Heng Li, Che Shao Chen, Weiyi Shang, and Tse-Hsun Chen. 2020. Logram: Efficient log parsing using n-gram dictionaries. *IEEE Transactions on Software Engineering* (2020).

[3] Hetong Dai, Yiming Tang, Heng Li, and Weiyi Shang. 2023. PILAR: Studying and mitigating the influence of configurations on log parsing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 818–829.

[4] Anwesha Das, Frank Mueller, and Barry Rountree. 2020. Aarohi: Making real-time node failure prediction feasible. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1092–1101.

[5] Anwesha Das, Frank Mueller, Charles Siegel, and Abhinav Vishnu. 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*. 40–51.

[6] Min Du and Feifei Li. 2016. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 859–864.

[7] Ying Fu, Meng Yan, Jian Xu, Jianguo Li, Zhongxin Liu, Xiaohong Zhang, and Dan Yang. 2022. Investigating and improving log parsing in practice. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1566–1577.

[8] Ying Fu, Meng Yan, Zhou Xu, Xin Xia, Xiaohong Zhang, and Dan Yang. 2023. An empirical study of the impact of log parsers on the performance of log-based anomaly detection. *Empirical Software Engineering* 28, 1 (2023), 6.

[9] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. 2016. Logmine: Fast pattern recognition for log analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 1573–1582.

[10] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. 2016. An evaluation study on log parsing and its use in log mining. In *2016 46th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 654–661.

[11] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 33–40.

[12] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R Lyu. 2021. A survey on automated log analysis for reliability engineering. *ACM computing surveys (CSUR)* 54, 6 (2021), 1–37.

[13] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. 2016. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*. IEEE, 207–218.

[14] Tong Jia, Lin Yang, Pengfei Chen, Ying Li, Fanjing Meng, and Jingmin Xu. 2017. Logsed: Anomaly diagnosis through mining time-weighted control flow graph in logs. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. IEEE, 447–455.

[15] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R Lyu. 2024. LILAC: Log Parsing using LLMs with Adaptive Parsing Cache. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 137–160.

[16] Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jiazhen Gu, Zhuangbin Chen, Jieming Zhu, and Michael R Lyu. 2024. A Large-Scale Evaluation for Log Parsing Techniques: How Far Are We?. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[17] Zhen Ming Jiang, Ahmed E Hassan, Parminder Flora, and Gilbert Hamann. 2008. Abstracting execution logs to execution events for enterprise applications (short paper). In *2008 The Eighth International Conference on Quality Software*. IEEE, 181–186.

[18] Zanis Ali Khan, Donghwan Shin, Domenico Bianculli, and Lionel Briand. 2022. Guidelines for assessing the accuracy of log message template identification techniques. In *Proceedings of the 44th International Conference on Software Engineering*. 1095–1106.

[19] Zanis Ali Khan, Donghwan Shin, Domenico Bianculli, and Lionel C Briand. 2024. Impact of log parsing on deep learning-based anomaly detection. *Empirical Software Engineering* 29, 6 (2024), 139.

[20] Van-Hoang Le, Domenico Bianculli, and Huy-Trung Nguyen. 2026. Towards Universal Segmentation for Log Parsing - Full Supplementary Material. https://doi.org/10.6084/m9.figshare.30448163.

[21] Van-Hoang Le, Domenico Bianculli, and Huy-Trung Nguyen. 2026. Towards Universal Segmentation for Log Parsing - SCLog Source Code. https://doi.org/10.6084/m9.figshare.31228420.

[22] Van-Hoang Le and Hongyu Zhang. 2021. Log-based Anomaly Detection Without Log Parsing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 492–504.

[23] Van-Hoang Le and Hongyu Zhang. 2023. Log parsing with prompt-based few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2438–2449.

[24] Van-Hoang Le and Hongyu Zhang. 2024. PreLog: A Pre-trained Model for Log Analytics. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–28.

[25] Van-Hoang Le, Hongyu Zhang, and Yi Xiao. 2025. Unleashing the True Potential of Semantic-Based Log Parsing with Pre-Trained Language Models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 711–711.

[26] Qingwei Lin, Ken Hsieh, Yingnong Dang, Hongyu Zhang, Kaixin Sui, Yong Xu, Jian-Guang Lou, Chenggang Li, Youjiang Wu, Randolph Yao, 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*. 480–490.

[27] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[28] Yudong Liu, Xu Zhang, Shilin He, Hongyu Zhang, Liqun Li, Yu Kang, Yong Xu, Minghua Ma, Qingwei Lin, Yingnong Dang, et al. 2022. Uniparser: A unified log parser for heterogeneous log data. In *Proceedings of the ACM Web Conference 2022*. 1893–1901.

[29] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2024. Llmparser: An exploratory study on using large language models for log parsing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[30] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. 2018. A search-based approach for accurate identification of log message formats. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 167–16710.

[31] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael Rung-Tsong Lyu, and Hua Cai. 2013. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems* 24, 6 (2013), 1245–1255.

[32] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. 2019. Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 1795–1812.

[33] Meiyappan Nagappan and Mladen A Vouk. 2010. Abstracting log lines to log event types for mining software system logs. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 114–117.

[34] Meiyappan Nagappan, Kesheng Wu, and Mladen A Vouk. 2009. Efficiently extracting operational profiles from execution logs using suffix arrays. In *2009 20th International Symposium on Software Reliability Engineering*. IEEE, 41–50.

[35] Sasho Nedelkoski, Jasmin Bogatinovski, Alexander Acker, Jorge Cardoso, and Odej Kao. 2020. Self-attentive classification-based anomaly detection in unstructured logs. In *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE, 1196–1201.

[36] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H Chin, and Sumayah Alrwais. 2015. Detection of early-stage enterprise infection by mining large-scale log data. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 45–56.

[37] Qiaolin Qin, Roozbeh Aghili, Heng Li, and Ettore Merlo. 2025. Preprocessing is All You Need: Boosting the Performance of Log Parsers With a General Preprocessing Framework. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 310–320.

[38] Risto Vaarandi. 2003. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*. Ieee, 119–126.

[39] Risto Vaarandi and Mauno Pihelgas. 2015. Logcluster-a data clustering and pattern mining algorithm for event logs. In *2015 11th International conference on network and service management (CNSM)*. IEEE, 1–7.

[40] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 117–132.

[41] Siyu Yu, Pinjia He, Ningjiang Chen, and Yifan Wu. 2023. Brain: Log parsing with bidirectional parallel tree. *IEEE Transactions on Services Computing* (2023).

[42] Chenbo Zhang, Wenying Xu, Jinbu Liu, Lu Zhang, Guiyang Liu, Jihong Guan, Qi Zhou, and Shuigeng Zhou. 2025. LogBase: A Large-Scale Benchmark for Semantic Log Parsing. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 2091–2112.

[43] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, 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*. 807–817.

[44] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. 2019. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 121–130.