Journal of Information Processing Vol.34 913–924 (Sep. 2026)

[DOI: 10.2197/ipsjjip.34.913]

Regular Paper

Efficient Identification of Original Logging Functions from Log Messages for Network Troubleshooting

Satoru Kobayashi[1,a)] Gaspard Damoiseau-Malraux[2,b)] Kensuke Fukuda[3,c)]

Received: December 11, 2025, Accepted: June 10, 2026

Abstract: System logs are valuable for network troubleshooting, but log messages often lack sufficient detail for operators to diagnose failures, requiring investigation of source code. While regular expression-based matching can identify the source code locations corresponding to log messages, this approach suffers from a long processing time as it must attempt matching against all patterns. We propose SCOLM, a method for efficiently identifying logging function origins from log messages. SCOLM achieves both efficiency and applicability through complementary use of word-segmented prefix trees (WSPT) and regular expressions. We evaluate SCOLM using ground truth data obtained through source code instrumentation and emulation network execution. The results demonstrate that SCOLM achieves approximately 78 times speedup compared to existing approaches without missing any correct origins while significantly reducing false positive candidates.

Keywords: log analysis, source code analysis, logging functions, regular expression, prefix tree

1. Introduction

Large-scale networks require continuous operation through network management techniques that utilize operational data. System logs are particularly valuable for detecting signs of failures and investigating their causes, as they allow developers to record detailed information about system behavior across many components.

However, in troubleshooting using system logs, the information recorded in log messages alone is often insufficient to fully understand the actual behavior of the system. Log messages are initially designed for debugging by developers and often lack sufficient detail for operators to diagnose failures. This limitation is particularly common in open-source software (OSS), which is widely used in network operations [20]. OSS log messages are often not refined due to limited development resources. Consequently, operators often need to investigate the source code to understand the system behavior during failures. However, it is not realistic to comprehensively investigate and understand all source code. For efficient troubleshooting, it is necessary to focus on the specific source code parts related to the observed log messages.

When searching for source code parts related to failures, it is useful to identify the location of the logging function in the source code corresponding to the output log message. **Figure 1** shows a simplified diagram of the process of pinpointing log messages to the source code. Understanding the execution flow of a program is crucial for troubleshooting, and identifying which logging function calls correspond to which log messages helps operators understand the system behavior during failure.

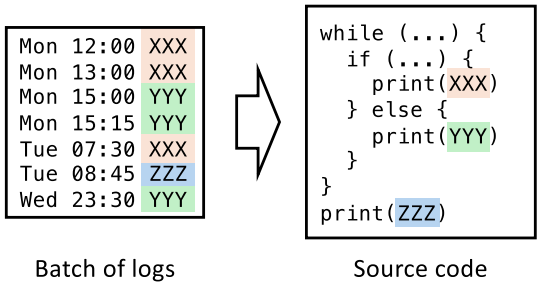

Fig. 1 Simplified diagram of pinpointing log batches to source code.

However, identifying these locations is challenging with conventional search methods. Log output formats in source code contain variable specifiers (e.g., `%s`, `%d` in C) that are replaced with actual values at runtime, creating a gap between the format string and the resulting log message. Conventional keyword-based search requires users to manually extract fixed parts of log messages, which is inefficient when the extracted keywords are insufficient to uniquely identify the target logging function.

In this paper, we propose SCOLM (Source Code Origins of Log Messages), a method for automatically identifying logging function origins from log messages. SCOLM extracts logging functions and their format strings from source code through static analysis. Given a log message as input, it matches the message against extracted templates and returns the corresponding source code locations. The key technical contribution is a hybrid matching approach that combines word-segmented prefix trees (WSPT) for search space reduction with regular expressions (regex) for precise matching, achieving fast processing without missing cor-

[1] Hiroshima City University, Graduate School of Information Sciences, Hiroshima 731–3194, Japan
[2] Sorbonne Université, CNRS, LIP6, Paris, France
[3] National Institute of Informatics, Information Systems Architecture Science Research Division, Chiyoda, Tokyo 101–8430, Japan
[a)] kobayashi-s@hiroshima-cu.ac.jp
[b)] gaspard.dama@lip6.fr
[c)] kensuke@nii.ac.jp

© 2026 Information Processing Society of Japan

rect origins.

We first review existing log matching approaches and introduce the idea of a hybrid approach combining WSPT and regular expressions (Section 2), then present the design of SCOLM based on this approach (Section 3). We have implemented SCOLM and evaluated it in terms of both processing time and matching quality using three software packages: FRRouting (an open-source router), Avahi (a service discovery daemon), and dhcpcd (a DHCP client) (Section 4). For the FRRouting dataset, we obtained ground truth data through source code instrumentation to enable accuracy evaluation. The results demonstrate that SCOLM achieves approximately 78 times speedup compared to regex-only approaches, without missing any correct origins (100% recall) while reducing false positive candidates (Section 5). We also discuss how to address potential matching failures and the limitations of the current approach (Section 6).

The contributions of this paper are as follows [*1].

(1) An architecture for automatically identifying logging function origins from log messages using static source code analysis and log matching (Section 3).

(2) A dynamic template addition mechanism that enables WSPT to be applied to source code templates, achieving both efficiency and applicability with approximately 78 times speedup compared to regex-only approaches (Section 5).

(3) Comprehensive evaluation of both processing time and matching quality using ground truth data obtained through source code instrumentation and emulation network execution, confirming that SCOLM achieves both fast processing and high matching quality (Sections 4 and 5).

2. Background

2.1 Log Matching

Log matching in our context serves two primary purposes: (1) providing troubleshooting information to operators and (2) serving as preprocessing for automated log analysis methods (e.g., log classification, execution flow extraction). In both cases, the system must process large volumes of log data. Therefore, we require that the proposed method achieves fast matching for high-volume inputs.

In the following, we first describe two existing approaches for log matching: regular expression-based matching and word-segmented prefix tree (WSPT) based matching. We then present our proposed hybrid approach that combines both methods to achieve fast processing with high matching precision.

2.1.1 Regular Expression Approach

Existing research mainly uses matching techniques based on regular expressions [26]. In this method, templates are extracted from logging functions in source code (hereafter, source code templates), and variable specifiers contained in these templates are typically replaced with regular expression patterns. This regular expression pattern can specify a narrow range of patterns if, for example, it is known that the type of the variable to be embedded is numeric, only allowing strings containing numbers. On the other hand, if the variable type is string, it is necessary to use a pattern that assumes strings containing almost any characters without restrictions. This causes the search space for strings matching the regular expression to become very large, leading to an increase in processing time.

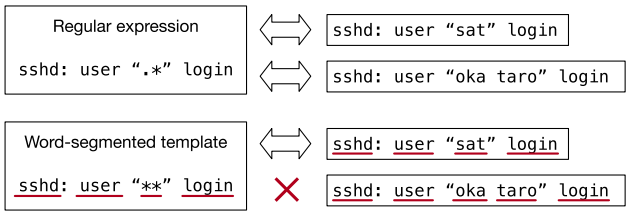

Fig. 2 Difference in matching behaviors between regular expressions and word-segmented templates.

Furthermore, in regular expression-based matching, it is necessary to attempt matching for as many patterns as there are, so in large-scale software where the number of patterns becomes large, the processing time for matching increases even further. When the log message can match multiple regular expression patterns, it is not possible to terminate the search when a matching pattern is detected, because the log message may match other patterns as well. This causes the processing time to increase in proportion to the number of patterns, and in large-scale software, the processing time for matching can become quite long. Note that the primary bottleneck is the number of matching attempts, not the time for individual pattern matching. Therefore, optimizing individual regex patterns does not fundamentally solve this scalability issue.

2.1.2 Word-segmented Prefix Tree Approach

As an alternative to regular expression-based matching, the word-segmented prefix tree (WSPT) approach [13] has been proposed for high-speed log matching. For example, as shown in **Fig. 2**, while variable parts (such as `.*`) in regular expressions can contain multiple words, in WSPT, one variable wildcard (`**`) in the log template (word-segmented template, WS-template) corresponds to one word, and multiple words separated by whitespace are not included.

In this way, WSPT consistently handles log messages under word segmentation, making it possible to fix the number of words in variable parts of log templates. By fixing the number of words, it becomes possible to use a prefix tree at the word level, enabling efficient log template search through fast matching. Specifically, when the number of template types is 10,000, WSPT has been confirmed to be capable of matching at a processing time of less than 1/10 compared to regular expressions [13].

The WSPT is implemented as a prefix tree where each node represents a word in the template. The tree structure allows efficient template search through word-level matching. **Figure 3** shows an example of the WSPT tree structure. In this example, the tree contains templates for messages like "User ** logged in", "User ** created", and "User not found", where ** represents wildcards matching any single word. The tree structure enables efficient matching by sharing common prefixes among templates.

[*1] This paper is an extended version of the work published in COMPSAC 2025 [5]. The main extensions include comprehensive accuracy evaluation using ground truth data (Section 5.2). This evaluation enables us to confirm that SCOLM achieves equivalent accuracy to baseline methods while providing significant speedup, identify potential failure cases, and discuss improvement strategies (Section 6.1). Based on these findings, we also refined the overall method description.

© 2026 Information Processing Society of Japan

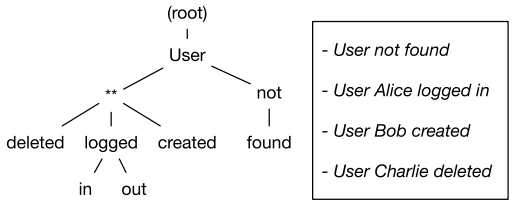

Fig. 3 Example of WSPT tree structure.

Note that a single log message may match multiple templates (i.e., multiple terminal nodes) depending on the wildcard positions, as the tree traversal considers both exact word matches and wildcard matches at each step.

However, WSPT lacks applicability to source code templates. This is because WSPT has the constraint of "consistently handling log messages under word segmentation": one wildcard in a word-segmented template must correspond to exactly one word. In source code templates, variable specifiers (e.g., `%s`) may embed an unknown number of words at runtime, making it impossible to determine the correct word-segmented template in advance.

2.1.3 Proposed Approach: Hybrid Matching

As described in the previous subsections, the regular expression (regex) table approach and WSPT each face distinct challenges. The regex table approach requires matching against all patterns for each log message, resulting in a long processing time. As explained above, WSPT lacks applicability to source code templates due to the unknown number of words in variable parts.

The proposed method addresses both challenges through a complementary combination of the two approaches. From the regex perspective, WSPT narrows down the candidate patterns that need to be tested, significantly reducing processing time. From the WSPT perspective, when a log message is matched against a regex pattern, the number of words in variable parts becomes determinable from the actual log content, enabling WSPT's applicability to source code templates by generating valid word-segmented templates that can be registered for future use.

Beyond processing time and template generation, each approach applies different types of matching constraints, leading to different overmatching behaviors. The regex table approach applies constraints on variable types and content patterns (e.g., `\d+` for numbers, `.*` for arbitrary strings) but does not consider word-level structure, potentially matching patterns where variable parts span different numbers of words. For example, a pattern `^Error: (.*)$` can match both `"Error: failed"` (single-word variable) and `"Error: timeout in system"` (three-word variable), even though these may correspond to different logging functions with different word structures. WSPT, conversely, applies constraints on word-level structure (number of words and positions of fixed words) but cannot distinguish patterns with the same structure while different variable types or content constraints. For instance, a WSPT template `Error ** occurred` cannot differentiate between numeric errors (`"Error 123 occurred"`) and string-based errors (`"Error timeout occurred"`), even though they may originate from different logging functions with different variable types.

Table 1 Comparison of three log matching approaches.

Method	Processing time	Source code templates	Overmatching
Regex table	Large	Available	Medium
WSPT	Small	Not available	Large
Proposed method	Small	Available	Small

By combining both approaches in a two-stage matching process, the proposed method applies both types of constraints simultaneously: WSPT filters candidates based on word-level structure, then regex matching verifies variable types and content patterns. This intersection-based approach requires patterns to satisfy both structural and content-level constraints, minimizing overmatching by eliminating false positives that would pass only one type of constraint. **Table 1** summarizes the characteristics of the three approaches, indicating source code template availability.

This hybrid matching method is particularly effective in environments where many messages of the same type are output. In general software, the number of log entries is extremely large compared to the number of log output formats. For example, in 15 months of SINET4 log data, while there are 35 million log entries, the number of different types of log templates is only 1,788 [13]. In such data, the approach of performing matching using regular expressions for every log entry for all regular expression patterns results in an extremely long processing time. In contrast, with the proposed method, once matching using regular expressions has been performed for a log, subsequent logs can be matched at high speed using WSPT, and the larger the ratio of log entry numbers to corresponding log output format types, the greater the proportion that can use WSPT.

Furthermore, among the logging functions in the source code, there are many logging functions that are called only in exceptional situations such as exception handling or debug processing, which rarely occur in general use. In other words, this "number of log output format types corresponding to log entries" is expected to be even smaller than the actual number of logging functions contained in the source code, which can further enhance the efficiency of the proposed method. In this way, the proposed matching method is designed to work particularly effectively for the task of associating source code and log messages, such as in this research.

2.2 Related Work

The task of associating output log messages with logging functions in source code has been performed as part of many existing studies [3]. In existing research, this association is mainly performed for two purposes.

The first purpose is to extract log output formats for log message classification. To classify log messages referring to the same type of event, classification based on log output formats can be effective. This approach is mainly used as preprocessing for log analysis. Although many methods such as unsupervised classification of log messages [8], [14], [25] or estimation of output formats [12], [23] have been proposed without using source code, using source code allows the use of the actual formats used for log message generation, enabling more reliable classification. Xu et al. [26] perform anomaly detection of distributed systems us-

© 2026 Information Processing Society of Japan

ing principal component analysis, and use log classification by source code templates as part of the features. Schipper et al. [21] implement and evaluate an analysis platform to make Xu et al.'s method applicable to general log analysis. Pecchia et al. [17] perform an investigation of logging usage in software development, and use association with source code for log data classification as part of this investigation.

The other purpose is to map log events onto the execution flow. By associating output log messages with logging functions in the source code, they can be handled as mapped onto the execution flow in the source code. This makes it possible to aggregate the obtained log events into sequential data as execution flows, or to analyze them using static analysis approaches for software. This approach is mainly used for system debugging, testing, and troubleshooting using logs. Chen et al. [4] develop LogCoCo for automatic code coverage calculation, and perform association between software execution logs and source code locations for this calculation. Yuan et al. [27] enable the use of debugging logs by automatically extracting execution paths and the locations of logging functions contained therein for automatic diagnosis of software bugs. Shang [22] organizes issues in log utilization from the perspective of the gap between development and operation in software, and mentions the usefulness of log-to-source-code association. Fu et al. [6] use association between logs and source code to learn normal execution flows as finite state automata for anomaly detection of distributed systems through log analysis. Bao et al. [2] similarly propose an anomaly detection method using probabilistic suffix trees for execution flows in distributed systems. Ghanbari et al. [7] perform outlier detection of execution flows using features generated from combinations of logging functions and execution times. Zhao et al. [28] propose lprof, a mechanism to extract execution flows without modifying the existing source code.

As described above, the task of associating log messages with logging functions in source code is performed as preprocessing for many automated analysis technologies. However, efficient methods for performing this association have not been sufficiently studied. In many of these technologies, the association is performed following Xu et al.'s method [26]. That is, source code templates are created from logging functions, and matching between these templates and log messages is performed using regular expressions. However, this log message association becomes a task that requires a long processing time depending on the number of target log messages and the number of output format types (number of logging functions). In regular expression-based matching, the search space for variable parts in the format is large, resulting in a long processing time, and when dealing with large-scale log data and source code, it can have an impact on practicality. In contrast, this study proposes a more efficient log message and logging function matching method.

3. Design of SCOLM

3.1 System Overview

The pipeline for pinpointing logging functions from log messages can be divided into two main steps: (1) template generation and (2) log matching. Template generation is performed as a preprocessing phase, where we extract logging function information from source code and prepare templates for matching. Log matching is performed during online processing, where the system takes a log message as input and returns its origin(s) in the source code.

Figure 4 shows the overall process. In the preprocessing step, we parse the source code to extract logging function calls and their format strings, then generate two types of templates: regular expression patterns for precise matching and word-segmented templates for fast candidate filtering. In the online matching step, we use WSPT to narrow down candidate patterns, then apply regular expression matching to verify the candidates.

This hybrid approach achieves both efficiency and applicability by addressing the two challenges discussed in Section 2.1: (1) WSPT's applicability to source code templates is enabled through dynamic template addition (Section 3.3.2), and (2) overmatching is minimized by applying both structural and content-level constraints in two-stage matching (Section 3.3.1).

SCOLM requires access to source code for template generation, making it applicable to open-source software and in-house developed software. Preprocessing is performed when the source code changes, while online matching can be applied to logs as they are collected during system operation.

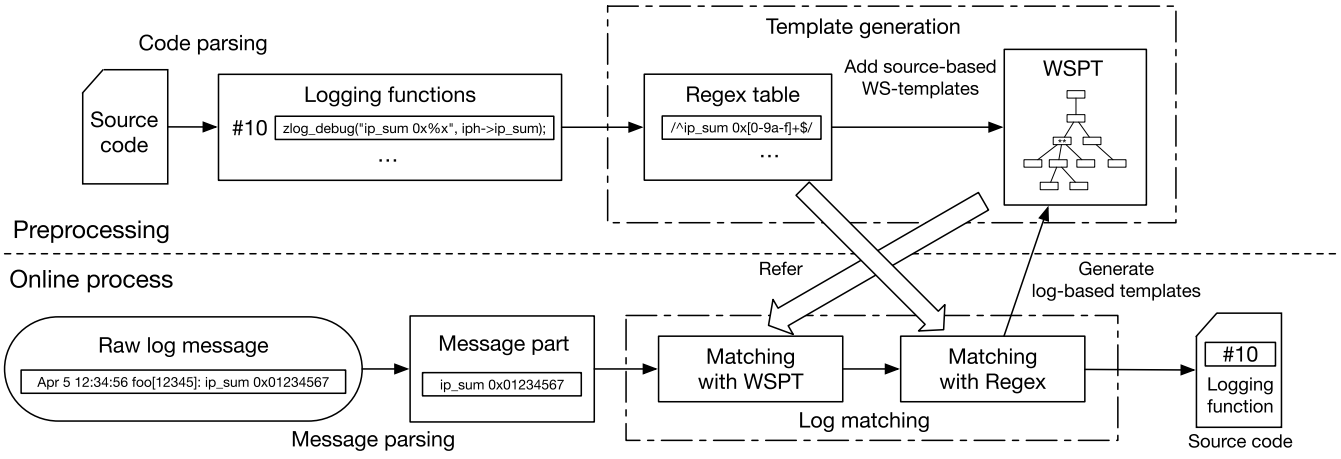

Fig. 4 Overview of the proposed method.

© 2026 Information Processing Society of Japan

3.2 Template Generation

The main purpose of the template generation step is to prepare datasets for the matching algorithm. This process is decomposed into three steps: code parsing, regex pattern generation, and WS-template generation.

3.2.1 Code Parsing

Code parsing relies on the assumption that logging functions include a basic blueprint of the log messages. SCOLM requires that the logging function arguments include a format string similar to `printf`'s. After the extraction, this format string is going to be the base material for template generation.

We parse the source code by identifying files and lines that contain calls to logging functions (specified by the user). The arguments of logging function calls are extracted, the most important being the format string. Optionally, the other arguments can be parsed in order to generate even more precise templates that depend on the actual values passed to the function.

For example, in FRRouting, the logging functions include `flog_err_sys`, `flog_err`, `flog_warn`, `zlog_debug`, `zlog_err`, `zlog_notice`, `zlog_info`, and `zlog_warn`. By specifying this list, SCOLM can extract all logging function calls from the FRRouting source code.

3.2.2 Regular Expression Table

The regex table is created by transforming all format strings into their regex equivalent, such that any log generated from the format string must match the regular expression. Placeholder parts in the regex pattern should be accurate "translations" of their C-style format specifiers in the string (`%d`, `%s`, `%.2f`, etc.). It is important to construct the most precise expression possible to minimize the tolerance of the expression. In addition, precise regex will decrease lookup speed by avoiding costly backtracking.

To achieve precise regex patterns, the system utilizes type information of variables to be embedded. This type information can be determined from the format specifiers in C-style format strings. For example, specifiers like `%d` will only produce (signed) integers matching `/-?\d+/`. Hexadecimal specifiers like `%x` produce patterns matching `/[0-9a-f]+/`. In contrast, other specifiers like `%s` can hardly be translated to anything more specific than `/.*?/` (lazy matching) without further knowledge of the corresponding argument, as they accept arbitrary strings. Lazy matching is used to reduce backtracking overhead during regex evaluation.

For example, the following logging function call in FRRouting: `zlog_debug("ip_sum 0x%x", iph->ip_sum);` is converted to the following regex pattern: `/^ip_sum 0x([0-9a-f]+)$/`.

If implemented correctly, regular expressions created by this process necessarily match any log generated by its corresponding format string. The system stores these regex patterns along with their source code location information for later retrieval during the matching process. Specifically, for each regex pattern, the system records the file name and line number where the corresponding logging function call appears in the source code. This information is maintained as a list, where multiple logging functions may generate the same regex pattern, and thus multiple source code locations can be associated with a single pattern.

3.2.3 WSPT Initialization

If we were to start matching logs with a WSPT initialized as empty, the system would have to go through the regex table for every type of log message it has not seen before. Of course, as the matching process goes on, the WSPT would eventually fill with all appropriate word-segmented templates, but at the cost of a high initial processing time.

To reduce this initial overhead, we initialize the WSPT by pre-filling it with source-based templates (heuristically generated from source code) before starting the online process. This WSPT initialization step generates templates based on the assumption that format specifiers in the format string will require only one wildcard in the word-segmented template. Although this is not true for all log messages, the assumption should cover a majority of them.

The WSPT is filled by generating basic word-segmented templates from the source code: in the extracted format string, every word that contains a format specifier would be replaced by a wildcard during template generation (e.g., `msg size=%zu` becomes `msg **`). As described in Section 2.1, the WSPT uses a prefix tree structure (see Fig. 3) where terminal nodes correspond to complete templates. In our proposed method, each terminal node is associated with a set of regex patterns that match the same word-segmented pattern. When multiple places in the source code create the same word-segmented template, we associate all corresponding regex patterns with the terminal node, and regex matching will later identify which specific origin(s) match the actual log message. Afterwards, during the online processing step, additional templates may be dynamically added to the WSPT when needed to better fit the dataset, as described in Section 3.3.

3.3 Log Matching

3.3.1 Matching Procedure

The log matching process follows a two-stage approach: WSPT-based candidate narrowing followed by regex matching. For each input log message, the system first searches the WSPT to identify matching word-segmented templates and retrieve their associated regex patterns. The system then performs regex matching only on this narrowed subset of candidates. If WSPT matching fails, the system falls back to full regex table search and dynamically adds new templates to the WSPT for future use, as described below. **Figure 5** illustrates the overall matching algorithm flow.

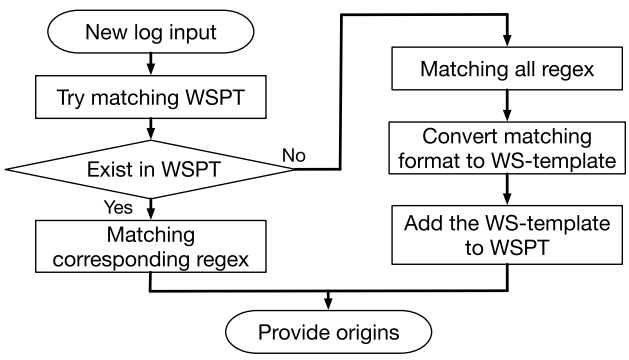

Fig. 5 Flow of the log matching algorithm.

© 2026 Information Processing Society of Japan

The WSPT search operates on word-segmented log messages. The system uses customizable word segmentation rules to handle cases where symbols are used inconsistently across log messages. For example, colons may appear both as part of IPv6 addresses (where they should not cause word breaks) and as word delimiters. By applying the same segmentation rules consistently to both log messages and regex patterns (when converting them to word-segmented templates), the system ensures that wildcards in templates can be treated as words, enabling proper matching behavior.

The WSPT search procedure follows a tree traversal approach. For an input log message, the system first divides it into words using the segmentation rules. Then, starting from the root node of the prefix tree, the system traverses nodes corresponding to each word in sequence. At each step, the system considers both exact word matches and wildcard matches (**), potentially creating multiple search paths. When the final word is reached, multiple terminal nodes may be reached, as a single log message can match multiple word-segmented templates depending on wildcard positions. The system collects the union of all regex pattern sets associated with these matched terminal nodes.

After narrowing down candidates through WSPT search, the system performs regex matching on the collected subset of patterns. All regex patterns that successfully match the log message are returned as the final result. This two-stage approach significantly reduces computation compared to matching against all regex patterns directly.

3.3.2 Dynamic Template Addition

When WSPT matching fails, the system falls back to full regex table search and converts the matching regex patterns to word-segmented templates for future use. The conversion process operates in three steps: (1) divide the input log into words on predefined whitespace characters; (2) extract the positions corresponding to variable parts in the matching regular expression; (3) replace all words that correspond to variable parts (either fully or partially) with wildcards (**). The system assembles the list of words and wildcards to form a word-segmented template that matches the log message.

The conversion process handles two main cases that differ between regex patterns and word-segmented templates. The first case is when a single variable part in the regex pattern contains multiple words, requiring multiple consecutive wildcards in the word-segmented template. The second case is when a variable part is only a portion of a single word. For example, AS numbers like “AS65000” correspond to format patterns like “AS%d”, which produces regex patterns like “AS[0-9]+”. While the variable part in the regex only matches the numeric portion, the word-segmented template must treat the entire word including the “AS” prefix as a single wildcard, since word segmentation cannot split within a word.

After generating the word-segmented template, the system adds it to the WSPT through a tree traversal procedure. Starting from the root node, the system follows nodes corresponding to each word in the template from left to right. For each word, if a corresponding node exists, the system moves to that node; otherwise, it creates a new node. This process continues for all words in the template. Finally, the terminal node corresponding to the last word is associated with the regex pattern(s) that generated this word-segmented template. Since one log message may match multiple regex patterns, one terminal node can be associated with multiple regex patterns.

4. Evaluation Setup

4.1 Evaluation Design

The primary goal of our evaluation is to demonstrate that SCOLM achieves both fast processing and high matching accuracy in identifying logging function origins from log messages. To comprehensively validate SCOLM, we conduct two types of evaluation: performance evaluation and accuracy evaluation.

The performance evaluation aims to demonstrate the efficiency of SCOLM’s hybrid matching approach combining WSPT and regular expressions. We measure the processing time required to match log messages against logging functions extracted from source code, and compare SCOLM with baseline approaches. The key question is whether the WSPT-based search space reduction effectively accelerates the matching process compared to regex-only approaches, and how much speedup can be achieved in practice.

The accuracy evaluation aims to verify that SCOLM correctly identifies the true origins of log messages. Unlike performance evaluation where processing time can be directly measured, accuracy evaluation requires ground truth data indicating which logging function in the source code generated each log message. Obtaining such ground truth is challenging because it requires knowing the exact correspondence between runtime log output and source code locations.

To address this challenge, we employ a source code instrumentation approach. We insert unique identifiers into each logging function call in the source code, then compile and execute the modified program to collect logs with embedded identifiers. Since each log message contains its unique identifier, we can establish the true correspondence between log messages and their source code locations. The same instrumented dataset is used for both performance evaluation (with identifiers removed before processing) and accuracy evaluation (with identifiers serving as ground truth). Details of the instrumentation process and dataset characteristics are described in Section 4.5.

4.2 Implementation

We implemented SCOLM as described in Section 3. It is written in Python 3.10 (using the `re` module for regex matching) and is publicly available as open-source software [*2].

For source code parsing, we use Ctags [24], a widely-used tool for creating indexes of names found in source code across various programming languages. This allows the SCOLM implementation to be applied to different programming languages without language-specific parsers.

For word segmentation, we use log2seq [11], which provides flexible customization of segmentation rules. This flexibility is important for handling cases where symbols are used inconsis-

[*2] The source code is available at: https://github.com/3atlab/scolm

© 2026 Information Processing Society of Japan

tently across log messages (e.g., colons appearing both within IPv6 addresses and as word delimiters).

4.3 Methods for Comparison

To evaluate the effectiveness of different components in SCOLM, we compare five different matching approaches:

Regex-only: This is the baseline method corresponding to existing approaches. It only uses regular expression patterns without WSPT. For every input log message, the system attempts matching against all regex patterns extracted from the source code. This method is expected to have the longest processing time.

SCOLM: This is the full proposed method as described in Section 3.

SCOLM-empty: This is a variant of SCOLM without WSPT initialization. The WSPT starts empty, and templates are added dynamically only when they are discovered through regex matching during online processing. This method evaluates the contribution of WSPT initialization by comparing with SCOLM.

SCOLM-oracle: This is a reference method for evaluating the performance upper bound of the hybrid approach. The WSPT is pre-trained using all input logs, enabling all logs to be matched using WSPT without fallback to regex table search. Since all logs are processed entirely through WSPT matching, this represents the performance that a hypothetical WSPT-only approach would achieve if it were applicable to this problem setting.

SCOLM-similarity: This is a variant for accuracy improvement, discussed in Section 6.1.

4.4 Evaluation Metrics

We use different metrics for performance evaluation and accuracy evaluation.

4.4.1 Performance Metrics

For performance evaluation, we measure processing time. We distinguish between **preprocessing time** (source code parsing and template generation, performed offline) and **processing time** (online log matching for all log entries in the dataset). Additionally, we report **WSPT hit rate** (the percentage of logs matched using WSPT without fallback) as supplementary data to interpret performance variations.

4.4.2 Accuracy Metrics

For accuracy evaluation, we use different metrics depending on the availability of ground truth data obtained through source code instrumentation.

For datasets with ground truth, we measure **recall** (the percentage of logs where the correct logging function is included in the returned candidates) and **set precision** (which additionally penalizes excessive candidates by averaging 1/n where n is the number of candidates). Set precision better represents overall matching quality as it captures both correctness and precision.

For comparisons involving datasets without ground truth, we report **candidate distribution** (the distribution of the number of candidates: 0, 1, 2, 3+), which can be measured without ground truth and enables comparison across datasets.

4.5 Dataset

We evaluate SCOLM using log data collected from emulated networks running open-source software. Our evaluation environment runs on Ubuntu Server 22.04 (x86_64) with an AMD Ryzen 7 7700 processor and 64 GB of memory.

Table 2 Dataset characteristics.

Dataset	Source Code		Log Data		
	Version	Regex Templates	WSPT Templates	Log Entries	Avg Words
FRR	8.5	6,365	6,365	52,805	12.9
Avahi	0.8	366	363	1,971	13.0
dhcpcd	10.1.0	586	580	5,509	8.8

For the primary evaluation, we use FRRouting [1] version 8.5, an open-source routing software written in C, which is widely used in both commercial and research network environments, making it a representative target for our use case. To collect log data, we construct Docker-based emulation networks using netroub [18], [19], an emulation platform for network trouble scenarios that uses Containerlab [16] and dot2net [10] to build virtual network topologies and execute failure scenarios.

We use the bgp_features sample network topology provided by dot2net, and execute 25 different failure scenarios provided by netroub. These scenarios include various realistic network problems such as device failures, traffic delays, and packet loss or corruption.

To evaluate the generalizability of our approach, we also conduct evaluation on two additional software packages with smaller codebases: Avahi version 0.8 (a service discovery daemon) and dhcpcd version 10.1.0 (a DHCP client). **Table 2** summarizes the characteristics of all three datasets.

For the FRR dataset, we apply the source code instrumentation approach described in Section 4.1 to obtain ground truth. We modify all 6,365 logging function calls in the FRRouting version 8.5 source code by inserting unique identifiers. For example, a logging function call like `zlog_debug("BGP session established")` is modified to include an identifier such as `zlog_debug("[ID:12345] BGP session established")`. We then compile and execute the instrumented program on the same emulation network, collecting logs with embedded identifiers.

This instrumented dataset contains the same 52,805 log entries as shown in Table 2, with each entry traceable to its exact source code origin. Importantly, identifiers do not influence matching results, as they are removed during header processing. This allows us to use the same dataset for both performance evaluation (identifiers removed) and accuracy evaluation (identifiers as ground truth).

5. Evaluation Results

5.1 Performance Evaluation

Table 3 shows the performance comparison among the five methods across all three datasets (metrics are defined in Section 4.4).

Comparison with baseline. The results demonstrate that SCOLM achieves significant speedup compared to the baseline Regex-only approach. For the FRR dataset with 52,805 log entries, SCOLM processes all logs in 1.76 seconds, achieving approximately 78 times speedup compared to Regex-only's 137.75

© 2026 Information Processing Society of Japan

Table 3 Processing time comparison across datasets.

Dataset	Method	Preprocessing (s)	Total Time (s)	Median (ms)	Max (ms)	WSPT Hit Rate
FRR	Regex-only	5.41	137.75	2.609	15.149	—
	SCOLM	5.76	1.76	0.033	7.628	100.0%
	SCOLM-empty	5.75	1.90	0.036	10.411	99.9%
	SCOLM-oracle	7.68	1.77	0.034	0.581	100.0%
	SCOLM-similarity	137.14	4.05	0.077	56.724	99.9%
Avahi	Regex-only	0.43	0.26	0.1295	0.6340	—
	SCOLM	0.44	0.06	0.0300	0.1037	100.0%
	SCOLM-empty	0.43	0.06	0.0293	0.4241	99.7%
	SCOLM-oracle	0.43	0.06	0.0298	0.0710	100.0%
dhcpcd	Regex-only	0.73	3.59	0.5519	5.3053	—
	SCOLM	0.75	0.40	0.0231	1.7574	96.6%
	SCOLM-empty	0.74	0.42	0.0226	3.6137	90.5%
	SCOLM-oracle	0.74	0.40	0.0229	1.6174	96.6%

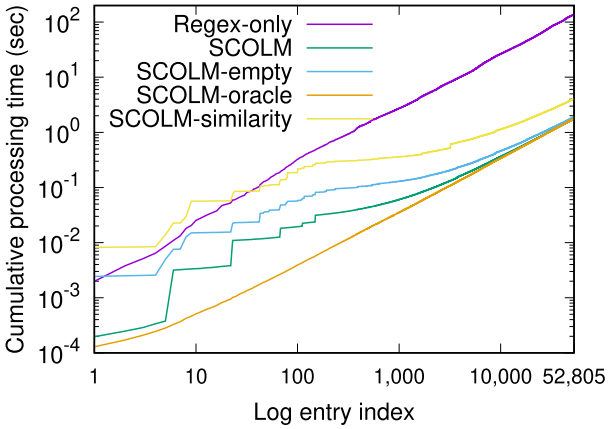

Fig. 6 Processing time transition during log matching.

seconds. The median processing time per log message, which reflects the typical cost for the majority of logs (regex matching for Regex-only, WSPT matching for SCOLM), is reduced from 2.609 ms to 0.033 ms. Similar speedups are observed for the other two datasets: Avahi shows 4.3 times speedup (0.26 s → 0.06 s), and dhcpcd shows 9.0 times speedup (3.59 s → 0.40 s).

The WSPT hit rate indicates the percentage of logs matched using WSPT without fallback to regex table search. A 100.0% hit rate (achieved for FRR and Avahi) means that the source-based templates from WSPT initialization completely covered all input logs. The miss rate (1 - hit rate), which represents the percentage of logs requiring regex table search, is approximately proportional to the total processing time. For example, dhcpcd's 3.4% miss rate (96.6% hit rate) results in longer processing time compared to the 100% hit rate cases, demonstrating this relationship.

Effect of WSPT initialization. Comparing SCOLM with SCOLM-empty reveals the contribution of WSPT initialization. SCOLM-empty, which starts with an empty WSPT, shows slightly longer processing times (1.90 s vs. 1.76 s for FRR) and lower WSPT hit rates (99.9% vs. 100.0% for FRR) compared to SCOLM. This indicates that WSPT initialization effectively reduces the number of logs that require fallback to regex table search, especially for the initial portion of log processing.

Processing time transition. **Figure 6** shows the cumulative processing time as logs are processed sequentially for the FRR dataset. This visualization reveals the internal behavior of SCOLM's hybrid matching approach.

Regex-only shows linear growth throughout, as it performs full regex table search for every log message. In contrast, SCOLM shows a distinct two-phase behavior. In the initial phase, SCOLM occasionally encounters logs that do not match any existing WSPT templates, requiring costly regex table search and subsequent dynamic template addition. As more logs are processed, the WSPT accumulates diverse templates, and the frequency of such costly operations decreases significantly. This results in the curve's transition from a steeper slope in the early phase to a much gentler slope in the later phase, approaching the efficiency of WSPT-only matching.

SCOLM-empty exhibits a more pronounced initial phase with steeper slope, as it must learn all templates from scratch.

Upper bound of hybrid approach. SCOLM-oracle represents the hypothetical WSPT-only approach that achieves the upper bound of hybrid approach performance. For the complete FRR dataset, SCOLM achieves nearly identical total processing time (1.76 s vs. 1.77 s) to SCOLM-oracle, demonstrating that when processing a sufficient number of logs, SCOLM achieves performance comparable to the theoretical best case where all logs are matched using WSPT.

As shown in Fig. 6, while SCOLM and SCOLM-oracle show a significant performance gap in the initial phase, this gap becomes relatively smaller as more logs are processed, and their cumulative processing times nearly converge. This confirms that the dynamic template addition mechanism effectively learns the necessary templates from the input logs, approaching the performance of perfect pre-training.

Table 4 Accuracy evaluation on FRR dataset.

Method	Recall	Set Precision	Avg Candidates
Regex-only	100.0%	77.6%	1.50
SCOLM	100.0%	87.6%	1.25
SCOLM-empty	99.2%	86.7%	1.27
SCOLM-oracle	100.0%	87.6%	1.25
SCOLM-similarity	100.0%	86.9%	1.28

5.2 Accuracy Evaluation

Table 4 shows the accuracy evaluation results for the FRR dataset using ground truth data obtained through source code instrumentation (described in Section 4.1). The metrics are defined in Section 4.4.

Comparison with baseline. SCOLM achieves 100.0% recall, meaning the correct logging function is always included in the returned candidate set. This matches the baseline Regex-only

© 2026 Information Processing Society of Japan

Table 5 Candidate distribution across three datasets.

Dataset	Method	Number of Candidates			
		0	1	2	3+
FRR	Regex-only	0.0%	57.0%	38.1%	4.8%
	SCOLM	0.0%	75.3%	24.6%	0.1%
	SCOLM-empty	0.0%	74.3%	24.6%	1.1%
	SCOLM-oracle	0.0%	75.3%	24.6%	0.1%
Avahi	Regex-only	0.0%	100.0%	0.0%	0.0%
	SCOLM	0.0%	100.0%	0.0%	0.0%
	SCOLM-empty	0.0%	100.0%	0.0%	0.0%
	SCOLM-oracle	0.0%	100.0%	0.0%	0.0%
dhcpcd	Regex-only	9.0%	71.3%	19.7%	0.0%
	SCOLM	9.0%	71.7%	19.2%	0.0%
	SCOLM-empty	9.0%	73.2%	17.8%	0.0%
	SCOLM-oracle	9.0%	71.7%	19.2%	0.0%

method, confirming that the WSPT-based search space reduction does not sacrifice correctness. SCOLM-oracle and SCOLM-similarity also achieve 100.0% recall.

While all methods achieve high recall, they differ significantly in precision. Regex-only shows 77.6% set precision with an average of 1.50 candidates per log, indicating that for many logs, multiple logging functions have regex patterns that match the same log message, leading to ambiguity. SCOLM improves set precision to 87.6% with 1.25 average candidates through WSPT-based filtering of candidates based on word-level structure.

Effect of WSPT initialization on recall. SCOLM-empty shows slightly lower metrics (99.2% recall, 86.7% set precision, 1.27 average candidates) compared to SCOLM. The 0.8% recall loss occurs due to a specific failure mechanism: when multiple regex patterns can match the same log message, SCOLM-empty may register only one of these patterns in the WSPT (through an earlier log), making other matching patterns inaccessible for subsequent logs. For example, if a generic pattern like `^(.*?) rcvd (.*?)$` matches first and is added to the WSPT, later logs that should match more specific patterns (such as OSPF6 protocol-specific patterns) fail because only the generic pattern is tested as a candidate. Since the WSPT template is associated exclusively with the first matched pattern, the correct specific pattern never gets a chance to match. WSPT initialization mitigates this issue by ensuring that specific patterns have their templates registered from the beginning (through source-based templates), preventing generic patterns from dominating the matching process. This demonstrates that WSPT initialization not only improves performance but also contributes to matching reliability by avoiding such pattern accessibility problems.

Precision limitations. The main cause of multiple candidates (reduced precision) is the presence of logging functions with identical or very similar format strings in the source code. For example, multiple error handling paths may use the same generic error message format like "`Failed to process: \%s`". In such cases, without additional context from the log message, it is impossible to uniquely determine which specific logging function generated the message. This is a fundamental limitation of matching based solely on log message content, and all methods face this ambiguity equally.

5.3 Candidate Distribution Analysis

Table 5 shows the candidate distribution across all three datasets. This analysis evaluates whether SCOLM generalizes to different software beyond FRR.

FRR dataset. As discussed in Section 5.2, SCOLM successfully reduces ambiguous candidates compared to Regex-only: the percentage of logs with unique matches increases from 57.0% to 75.3%, while logs with three or more candidates decrease from 4.8% to 0.1%. All logs are successfully matched (0% unmatched), demonstrating that the proposed approach provides appropriate candidates with improved precision.

Successful generalization: Avahi. For the Avahi dataset, all methods achieve perfect matching results: 100.0% of logs have exactly one candidate, with no unmatched logs or ambiguous matches. This demonstrates that SCOLM successfully generalizes to software with well-structured logging practices, where logging functions use sufficiently distinct format strings.

Limitation case: dhcpcd. The dhcpcd dataset reveals a limitation of the proposed approach: 9.0% of logs remain unmatched (0 candidates) across all methods. This occurs because dhcpcd uses some logging mechanisms that do not follow the standard `printf`-style format string pattern. Some log messages are constructed dynamically through string concatenation rather than through format strings in logging function calls. Since SCOLM's template extraction relies on detecting format strings, it cannot extract templates for these dynamically constructed messages.

Despite this limitation, dhcpcd shows reasonable matching performance for the remaining 91% of logs. Among matched logs, SCOLM maintains comparable detection rates to the baseline (71.7% vs. 71.3% for single candidates) while showing similar precision characteristics (19.2% vs. 19.7% for two candidates).

6. Discussion

6.1 Addressing Matching Failures

While the proposed method (SCOLM) achieves 100% recall on the FRR dataset, SCOLM-empty shows 0.8% matching failures (99.2% recall). As explained in Section 5.2, these failures occur when generic regex patterns prevent more specific patterns from being tested. This reveals a potential weakness: the proposed method could experience similar failures on datasets where WSPT initialization does not cover all necessary templates.

To demonstrate that recall can be improved when the basic approach encounters failures, we investigate a pattern similarity-based enhancement technique. We apply this technique to SCOLM-empty as a test case.

Similarity-based matching approach. As discussed in Section 5.2, when one pattern is registered in the WSPT, other patterns that could also match become inaccessible. To address this issue, ideally we would group all regex patterns that can match the same log message. However, this requires solving the regex intersection problem, which is PSPACE-complete [9], [15] and computationally intractable.

SCOLM-similarity addresses this challenge by grouping patterns based on similarity rather than exact matching capability, assuming that similar patterns are likely to match common log messages.

To measure pattern similarity, we extract the fixed word portions from each pattern and compute their longest common subse-

© 2026 Information Processing Society of Japan

quence (LCS), which captures both the common words and their order while ignoring variable parts.

Furthermore, analysis of the SCOLM-empty failure cases revealed that many matching failures involve patterns in inclusion relationships, where a generic pattern with few fixed words can match logs that should match more specific patterns with many fixed words. To prioritize detection of such inclusion relationships, we apply the Simpson coefficient (also known as the overlap coefficient) to the LCS-based similarity:

$$\text{Simpson}(P_1, P_2) = \frac{|\text{LCS}(W_1, W_2)|}{\min(|W_1|, |W_2|)}, \quad (1)$$

where W_1 and W_2 are sequences of fixed words extracted from patterns P_1 and P_2 (excluding variable parts), and $\text{LCS}(W_1, W_2)$ denotes their longest common subsequence. The Simpson coefficient assigns high scores to pattern pairs in inclusion relationships: when the fixed words of one pattern form a subsequence of another pattern's fixed words, the similarity score approaches 1.

In the preprocessing step, for each regex pattern, we calculate the Simpson coefficient with all other patterns and record those exceeding a threshold (0.6) as similar patterns. This creates an association list for each pattern. For example, if we have three patterns (A, B, C), the association lists might look like:

```
Pattern A: [A, B]
Pattern B: [A, B, C]
Pattern C: [B, C].
```

Note that this relationship is not symmetric: pattern A may be similar to B, but B may not be similar to A from its perspective. This differs from traditional clustering, as the grouping is pattern-specific rather than forming disjoint clusters.

During online processing, when a log message matches a regex pattern during fallback, the system associates the generated WSPT template with the union of the matched pattern and all its similar patterns. When a subsequent log triggers a WSPT hit, all associated patterns are tested as candidates.

Effectiveness and trade-offs. Table 4 shows the results for SCOLM-similarity. SCOLM-similarity achieves 100.0% recall, successfully recovering from the 0.8% failure rate observed in SCOLM-empty. The set precision is 86.9%, comparable to other methods, with an average of 1.28 candidates.

As shown in Table 3, SCOLM-similarity has significantly longer preprocessing time (137.14 seconds vs. 5.76 seconds for SCOLM) due to pairwise similarity computations, and longer online processing time (4.05 seconds vs. 1.76 seconds) because each WSPT hit requires testing multiple associated patterns rather than a single pattern. The WSPT hit rate remains high at 99.9%.

When considering both preprocessing and online processing time, the total processing time of SCOLM-similarity is comparable to Regex-only (the baseline). However, the scalability characteristics differ. SCOLM-similarity is advantageous when the number of log entries increases, as WSPT accelerates online matching. Conversely, Regex-only is preferable when the source code grows larger, as SCOLM-similarity's preprocessing overhead scales quadratically with the number of patterns.

This pattern similarity-based approach presents a trade-off between recall and processing time: it recovers matching failures at the cost of increased processing time. For SCOLM, which already achieves 100% recall through WSPT initialization, this additional complexity is unnecessary. However, for scenarios where WSPT initialization cannot be used, such as online learning scenarios or when the one-variable-one-word assumption is less applicable, SCOLM-similarity provides an effective fallback option.

6.2 Limitations

While SCOLM demonstrates effective processing time and matching quality for identifying logging function origins from log messages, several limitations remain.

Applicability to non-standard logging patterns. As demonstrated by the dhcpcd evaluation results in Section 5.3, SCOLM's static analysis approach cannot handle software that uses non-standard logging patterns. Specifically, when log messages are constructed dynamically through string concatenation or other methods rather than through `printf`-style format strings in logging function calls, SCOLM cannot extract the necessary templates. This results in matching failures for approximately 9% of dhcpcd logs.

This limitation is fundamental to the static analysis approach because SCOLM relies on detecting format strings at logging function call sites in the source code. When the message content is determined at runtime through complex control flow or dynamic string operations, static analysis alone cannot capture all possible message patterns.

Language and framework dependencies. SCOLM currently targets C-language software with `printf`-style logging functions. While this covers a significant portion of network infrastructure software (including FRRouting, Avahi, and dhcpcd), many modern applications use different programming languages and logging frameworks.

Extending SCOLM to other languages requires adapting the code parsing and template extraction components. For languages like Python, Java, or Go, the logging mechanisms differ significantly from C's printf-style formatting. Each language has its own formatting conventions: Python's logging module uses percent-style or brace-style formatting, Java uses various logging frameworks (Log4j, SLF4J, etc.), and Go uses `fmt.Printf` or structured logging libraries. All of these would require language-specific parsing logic and format string interpretation. Moreover, the increasing adoption of structured logging frameworks that output JSON or other structured formats rather than plain text messages presents an additional challenge, as these would require a fundamentally different matching approach.

Handling of ambiguous matches. As shown in Table 5, even when SCOLM successfully identifies logging function origins, approximately 25% of FRR logs return multiple candidates due to identical or very similar format strings used by different logging functions. This is an inherent limitation of matching based solely on log message content.

7. Conclusion

This paper presented SCOLM, a system for automatically iden-

© 2026 Information Processing Society of Japan

tifying logging function origins in source code from log messages. We proposed a hybrid log matching method that combines word-segmented prefix trees (WSPT) for search space reduction with regular expressions for precise matching. We implemented SCOLM and conducted comprehensive evaluation on three open-source software packages: FRRouting, Avahi, and dhcpcd. SCOLM achieved approximately 78 times speedup compared to the regex-only baseline approach while maintaining 100% recall. Additionally, SCOLM demonstrates better reliability with higher set precision (87.6%) compared to the baseline approach (77.6%), reducing false positive candidates. These results demonstrate that SCOLM achieves both efficiency and reliability, enabling practical log-to-code tracing for network troubleshooting.

Future work includes extending SCOLM to support additional programming languages and logging frameworks, and addressing the limitations identified in Section 6.2. Beyond these technical enhancements, integrating SCOLM into development tools such as IDE extensions or log analysis platforms would enable developers to trace log messages back to source code during troubleshooting, bringing this approach into practical development workflows. SCOLM provides a practical solution for connecting runtime log messages to their source code origins, supporting more effective network troubleshooting and log-based system analysis.

Acknowledgments This work was partially supported by ROIS NII Open Collaborative Research 2025 (251S1-22594) and JSPS KAKENHI Grant Number JP25K15079.

References

[1] FRRouting: frr: The FRRouting Protocol suite, available from ⟨https://github.com/FRRouting/frr/⟩ (accessed 2024-12-28).
[2] Bao, L., Li, Q., Lu, P., Lu, J., Ruan, T. and Zhang, K.: Execution anomaly detection in large-scale systems through console log analysis, *Journal of Systems and Software*, Vol.143, pp.172–186 (2018).
[3] Bushong, V., Ramakrishnan, C., Sridharan, M. and Chandra, S.: Matching Patterns in the Era of Data-Driven Software Engineering, *Proc. 2020 IEEE International Conference on Software Maintenance and Evolution* (*ICSME*), pp.1–12 (2020).
[4] Chen, B. and Jiang, Z.M.: Automated Log Analysis: A Survey, *Proc. 2018 IEEE International Conference on Software Quality, Reliability and Security* (*QRS*), pp.1–12 (2018).
[5] Damoiseau-Malraux, G., Kobayashi, S. and Fukuda, K.: Automatically pinpointing original logging functions from log messages for network troubleshooting, *Proc. 49th Annual IEEE Computers, Software, and Applications Conference* (*COMPSAC*), pp.766–775 (2025).
[6] Fu, Q., Lou, J.-G., Wang, Y. and Li, J.: Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis, *Proc. 2009 IEEE International Conference on Data Mining* (*ICDM*), pp.149–158 (2009).
[7] Ghanbari, S., Hashemi, A.B. and Amza, C.: Stage-aware anomaly detection through tracking log points, *Proc. 15th International Middleware Conference*, pp.253–264 (2014).
[8] He, P., Zhu, J., Zheng, Z. and Lyu, M.R.: Drain: An online log parsing approach with fixed depth tree, *Proc. 2017 IEEE International Conference on Web Services* (*ICWS*), pp.33–40 (2017).
[9] Jin, H., Hahn, J. and Han, Y.-S.: RegexPSPACE: A Benchmark for Evaluating LLM Reasoning on PSPACE-complete Regex Problems (2025).
[10] Kobayashi, S.: dot2net: A labeled graph approach for template-based configuration of emulation networks, available from ⟨https://github.com/cpflat/dot2net⟩ (accessed 2024-09-30).
[11] Kobayashi, S.: log2seq: parse syslog-like messages into word sequences, available from ⟨https://github.com/amulog/log2seq⟩ (accessed 2024-01-24).
[12] Kobayashi, S., Otomo, K., Fukuda, K. and Esaki, H.: Towards an NLP-based Log Template Generation Algorithm for System Log Analysis, *Proc. 9th International Conference on Future Internet Technologies* (*CFI*), pp.1–4 (2014).
[13] Kobayashi, S., Yamashiro, Y., Otomo, K. and Fukuda, K.: amulog: A general log analysis framework for comparison and combination of diverse template generation methods, *International Journal of Network Management*, Vol.32, No.4, p.e2195 (2022).
[14] Makanju, A., Zincir-Heywood, A.N. and Milios, E.E.: A Lightweight Algorithm for Message Type Extraction in System Application Logs, *IEEE Trans. Knowledge and Data Engineering*, Vol.24, No.11, pp.1921–1936 (2012).
[15] Meyer, A.R. and Stockmeyer, L.J.: The equivalence problem for regular expressions with squaring requires exponential space, *13th Annual Symposium on Switching and Automata Theory* (*swat 1972*), pp.125–129 (1972).
[16] Nokia: containerlab: Orchestrating and managing container-based networking labs, available from ⟨https://github.com/srl-labs/containerlab⟩ (accessed 2024-09-30).
[17] Pecchia, A., Cinque, M., Carrozza, G. and Cotroneo, D.: Industry Practices and Event Logging: Assessment of a Critical Software Development Process, *Proc. 37th International Conference on Software Engineering* (*ICSE*), pp.169–179 (2015).
[18] Regal-Mezin, C.: netroub: Platform to emulate network trouble scenarios to collect synthetic operational data, available from ⟨https://github.com/3atlab/netroub⟩ (accessed 2024-09-30).
[19] Regal-Mezin, C., Kobayashi, S. and Yamauchi, T.: netroub: Towards an Emulation Platform for Network Trouble Scenarios, *Proc. CoNEXT Student Workshop 2023* (*CoNEXT-SW*), pp.17–18 (2023).
[20] Saadon, G., Haddad, Y. and Simoni, N.: A survey of application orchestration and OSS in next-generation network management, *Computer Standards & Interfaces*, Vol.62, pp.17–31 (2019).
[21] Schipper, D., Aniche, M. and van Deursen, A.: Tracing Back Log Data to its Log Statement: From Research to Practice, *2019 IEEE/ACM 16th International Conference on Mining Software Repositories* (*MSR*), pp.545–549 (2019).
[22] Shang, W., Jiang, Z.M., Hemmati, H., Adams, B., Hassan, A.E. and Martin, P.: Assisting developers of Big Data analytics applications when deploying on Hadoop clouds, *Proc. 34th International Conference on Software Engineering* (*ICSE*), pp.402–411 (2012).
[23] Thaler, S., Menges, F. and Lückerath, D.: Towards Automated Log Template Extraction, *Proc. 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops* (*ICSTW*), pp.1–6 (2017).
[24] Universal Ctags: ctags, available from ⟨https://github.com/universal-ctags/ctags⟩ (accessed 2024-01-24).
[25] Vaarandi, R. and Pihelgas, M.: LogCluster - A Data Clustering and Pattern Mining Algorithm for Event Logs, *Proc. 11th International Conference on Network and Service Management* (*CNSM*), pp.1–8 (2015).
[26] Xu, W., Huang, L., Fox, A., Patterson, D. and Jordan, M.I.: Detecting large-scale system problems by mining console logs, *Proc. ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pp.117–132 (2009).
[27] Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y. and Pasupathy, S.: SherLog: Error diagnosis by connecting clues from run-time logs, *ACM SIGARCH Computer Architecture News*, Vol.38, No.1, pp.143–154 (2010).
[28] Zhao, X., Zhang, Y., Lion, D., Ullah, M.F., Luo, Y., Yuan, D. and Stumm, M.: lprof: A Non-intrusive Request Flow Profiler for Distributed Systems, *11th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI 14*), pp.629–644 (2014).

Satoru Kobayashi received his Ph.D. degree in information science and technology from the University of Tokyo, Tokyo Japan, in 2018. He is an Associate Professor at Graduate School of Information Sciences, Hiroshima City University. His research interests are network management and data mining.

© 2026 Information Processing Society of Japan

Gaspard Damoiseau-Malraux is a Ph.D. student in computer science at Sorbonne Université, France. He received his Master's degree in cryptography and high performance computing at the same university in 2025. His research interests include cryptography, historical cryptography and iterative methods.

Kensuke Fukuda is a Professor at Information Systems Architecture Research Division, National Institute of Informatics and Department of Informatics, School of Multidisciplinary Sciences, The Graduate University for Advanced Studies (SOKENDAI). He received his Ph.D. in 1999 from Keio University. His research interests span Internet traffic analyses, anomaly detection, modeling networks and QoS over the Internet.

© 2026 Information Processing Society of Japan