

Unsupervised, Accurate, and Efficient Log Parsing Using Smaller Open-Source Large Language Models

ZEYANG MA, Software Performance, Analysis, and Reliability (SPEAR) lab, Concordia University, Canada

DONG JAE KIM, DePaul University, USA

TSE-HSUN (PETER) CHEN, Software Performance, Analysis, and Reliability (SPEAR) lab, Concordia University, Canada

Log parsing transforms unstructured logs into structured templates for downstream analysis. Syntax-based parsers are fast but lose accuracy on logs that deviate from predefined rules. Recently, large language models (LLMs) based log parsers have shown superior parsing accuracy but face three issues: (1) manual labeling for fine-tuning or in-context learning, (2) high cost from large volumes and limited context size of LLMs, and (3) privacy risks with commercial models. We present LibreLog, an unsupervised approach using open-source LLMs to enhance privacy and reduce cost while achieving state-of-the-art accuracy. LibreLog groups logs with a fixed-depth tree, then parses each group via: (i) similarity scoring-based retrieval augmented generation, (ii) self-reflection to refine templates, and (iii) a template memory to reduce LLM queries. On LogHub-2.0, LibreLog achieves GA 87.2, PA 85.4, FGA 82.3, and FTA 65.1, PA and FTA outperforming prior state-of-the-art LLM-based parsers by 13.7% and 6.9%, respectively. LibreLog processes all logs in 5.94 hours, a 1.7 times speedup over the fastest LLM parser. Using a larger LLM only for self-reflection further improves PA to 86.3 and FTA to 68.3 with a moderate runtime cost increase (31%). In short, LibreLog addresses privacy and cost concerns of using commercial LLMs while achieving state-of-the-art parsing efficiency and accuracy.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**.

Additional Key Words and Phrases: Log parsing, Log analysis, Software logs

1 INTRODUCTION

Real-world software systems generate large amounts of logs, often hundreds of gigabytes or even terabytes per day [14, 25, 76]. These logs provide developers with invaluable runtime information, essential for understanding system execution and debugging. To manage and analyze this vast amount of data, researchers and practitioners have proposed many automated approaches, such as monitoring [10, 65], anomaly detection [36, 62], and root cause analysis [51, 69]. However, as shown in Figure 1, logs are semi-structured, containing a mixture of static text and dynamically generated variables (e.g., port number 62267), which makes direct analysis challenging.

Log parsing is a critical first step in log analysis that transforms unstructured logs into log templates, dividing logs into static parts (static messages) and dynamic parts (variables). As illustrated in Figure 1, log templates represent the event structure of logs, providing a standardized format that simplifies further analysis. By distinguishing between static and dynamic components, log parsing enables more efficient and accurate downstream tasks [30, 37, 58]. Given the sheer volume and diversity of generated logs, prior research has proposed various syntax-based parsers for efficient and effective log parsing. These parsers, such as Drain [18] and AEL [24],

Authors' addresses: Zeyang Ma, Software Performance, Analysis, and Reliability (SPEAR) lab, Concordia University, Montreal, Canada, dkim121@depaul.edu; Dong Jae Kim, DePaul University, Chicago, USA, dkim121@depaul.edu; Tse-Hsun (Peter) Chen, Software Performance, Analysis, and Reliability (SPEAR) lab, Concordia University, Montreal, Canada, peterc@encs.concordia.ca.

Please use nonacm option or ACM Engage class to enable CC licenses



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 1049-331X/2026/1-ART

<https://doi.org/10.1145/3796239>

Log messages					
2015-10-18	18:01:51	INFO	main	org.apache.hadoop.http.HttpServer2	Jetty bound to port 62267
2015-10-18	18:01:51	INFO	main	org.apache.hadoop.http.HttpServer2	Jetty bound to port 62258
2015-10-18	18:01:52	INFO	main	org.apache.hadoop.yarn.WebApps	Web app /mapreduce started at 62267
2015-10-18	18:01:53	INFO	main	org.apache.hadoop.app.RMContainerRequestor	nodeBlacklistingEnabled: true
2015-10-18	18:01:53	INFO	IPC Server	org.apache.hadoop.ipc.Server	IPC Server listener on 62270 : starting

Parsed templates					
2015-10-18	18:01:52	INFO	main	org.apache.hadoop.http.HttpServer2	Jetty bound to port <*>
2015-10-18	18:01:52	INFO	main	org.apache.hadoop.http.HttpServer2	Jetty bound to port <*>
2015-10-18	18:01:52	INFO	main	org.apache.hadoop.yarn.WebApps	Web app <*> started at <*>
2015-10-18	18:01:53	INFO	main	org.apache.hadoop.app.RMContainerRequestor	nodeBlacklistingEnabled:<*>
2015-10-18	18:01:53	INFO	IPC Server	org.apache.hadoop.ipc.Server	IPC Server listener on <*>: starting

Fig. 1. An example of log parsing result from Hadoop.

use manually crafted heuristics or predefined rules to identify and extract log templates. Although promising, these log parsers often experience decreased accuracy when processing logs that deviate from predefined rules [25, 29, 70, 76].

Recent advances in large language models (LLMs) have enabled researchers to leverage these models for log parsing [11, 26, 33, 34, 44, 66]. LLMs exhibit superior capabilities in understanding and generating text, making them particularly effective for parsing semi-structured log data. Consequently, LLM-based log parsers often achieve higher accuracy than traditional syntax-based parsers [26, 34, 44]. However, the sheer volume of log data and the limited context size of LLMs lead to increased parsing costs, both in terms of time and money, as token consumption grows linearly with log size. This makes practical adoption challenging. Additionally, these parsers frequently require manually derived log template pairs for in-context learning, adding significant manual overhead.

A further complication arises from the reliance on commercial LLMs like ChatGPT by many LLM-based log parsers [26, 33, 66]. While powerful, using commercial models poses potential privacy risks, as logs often contain sensitive information about the software’s runtime behavior and data. Uploading logs and other sensitive information (e.g., code for refactoring and bug fixes) to commercial LLMs can expose a company’s sensitive data to potential privacy breaches [1].

To address these challenges, we propose an unsupervised log parsing technique, LibreLog, which does not need any manual labels. LibreLog leverages smaller-size open-source LLMs (e.g., Llama3-8B [4]) to enhance privacy and reduce operational costs while achieving state-of-the-art parsing accuracy and efficiency. Inspired by the effective grouping capabilities of syntax-based unsupervised log parsing methods [18], LibreLog first groups logs that share syntactic similarity in the static text, but vary in the dynamic variable, using a fixed-depth grouping tree. Then, LibreLog parses logs within individual groups through three key steps: (i) LibreLog uses similarity scoring-based **retrieval augmented generation (RAG)** to select the most diverse logs based on Jaccard similarity within each log group. This step helps LLMs separate dynamic and static text by highlighting variability in dynamic variables among logs in the same group. (ii) LibreLog uses **self-reflection** [59] to improve LLM responses, thereby improving parsing results. (iii) LibreLog uses **log template memory** to store parsed log templates. This approach allows logs to be parsed by first matching them with stored templates, minimizing the number of LLM queries and significantly enhancing parsing efficiency.

The paper makes the following contributions:

- We introduce LibreLog, an unsupervised log parsing technique that effectively addresses the limitations of existing LLM-based and syntax-based parsers.
- LibreLog employs open-source LLMs, specifically Llama3-8B, to enhance data privacy and reduce operational costs associated with commercial models.
- Through extensive evaluations on over 50 million logs from LogHub2.0 [25], LibreLog demonstrated 85.4% parsing accuracy, which is 13.7% or higher compared to state-of-the-art LLM-based log parsers (i.e., LUNAR [21], LILAC [26] and LLMParser [44]). Moreover, LibreLog spends a total of 5.94 hours for all 50 million logs, which is 1.79 to 40 times faster, showcasing its superior efficiency and effectiveness.
- LibreLog’s self-reflection mechanism helps improve parsing accuracy by over 7%, showcasing the effectiveness of our prompting technique.
- Strengthening LibreLog’s self-reflection module using a larger LLM (i.e., LLaMA 70B) further boosts accuracy (about 5%) while incurring moderate overhead (about 31%).
- Our experiment using four small-sized LLMs shows that Llama3-8B achieves the best overall result, highlighting its potential in log analysis.
- We manually investigate the remaining mismatches, classifying the dominant failure modes and outlining directions for future improvement.

This work extends our previous work [45]. First, on top of message-level evaluation metrics GA/PA, we introduced two template-level metrics, FGA and FTA. Template-level metrics address the sensitivity of GA/PA to imbalanced data, providing a comprehensive view of evaluation. Second, we introduce a new research question that investigates how replacing the fixed-depth grouping tree with density-based clustering affects accuracy and efficiency. Third, we add another research question that leverages a larger LLM only during self-reflection, exploring how larger LLMs impact the effectiveness and efficiency. Finally, we conduct an error analysis of the remaining mismatched logs and identify the principal failure modes to guide future improvements.

Paper Organization. Section 2 discusses background and related work. Section 3 provides the design details of LibreLog. Section 4 outlines evaluation setup. Section 5 presents evaluation results. Section 6 discusses error analysis and threats to validity. Section 7 concludes the paper.

Data Availability: We made our source code and experimental results publicly available at: <https://github.com/zeyang919/LibreLog>

2 BACKGROUND AND RELATED WORK

In this section, we discuss the background of LLM and its privacy concerns. We then discuss related log parsing research.

2.1 Background

Large Language Models. Large Language Models (LLMs), primarily built on the transformer architecture [4, 9, 52], have significantly advanced the field of natural language processing (NLP). These LLMs, such as the widely recognized GPT-3 model with its 175 billion parameters [9], are trained on diverse text data from various sources, including source code. The training involves self-supervised learning objectives that enable these models to develop a deep understanding of language and generate text that is contextually relevant and semantically coherent. LLMs have shown substantial capability in tasks that involve complex language comprehension and generation, such as code recognition and generation [5, 39]. Due to logs being semi-structured texts composed of natural language and code elements, researchers have adopted LLMs to tackle log analysis tasks, such as anomaly detection [36, 40, 62], root cause analysis [51, 54, 55], and log parsing [11, 26, 33, 34, 44, 66]. Log parsing is one of

the primary tasks of focus in this area, given its crucial role for more accurate and insightful downstream log analysis [30, 58].

Privacy Issues Related to LLM. While LLMs demonstrate remarkable capabilities in processing and generating natural language and code, their application on sensitive data such as logs presents notable privacy risks, particularly with commercial models such as ChatGPT [2, 9]. One major concern is that data transmitted to these models—such as system logs—could be retained and used in the model’s further training cycles without explicit consent or knowledge of the data owners [7]. More importantly, sensitive data uploaded to the LLM providers could potentially be exposed through inadvertent data leaks or malicious attacks [22], posing significant privacy risks. To avoid such risks, an industry norm is to restrict the use of commercial LLMs despite their advanced capabilities. For example, Samsung bans ChatGPT and other commercial chatbots after a sensitive code leak [1]. Major financial institutions like Citigroup and Goldman Sachs have restricted the use of ChatGPT due to concerns over data privacy and security [3]. In contrast, open-source LLMs, such as those developed by Meta’s Llama series [4, 52], offer greater privacy and security. Users can adopt the LLMs for local deployment to ensure data privacy, aligning with stringent data protection standards. Thus, open-source LLMs are more secure and trustworthy for handling confidential data such as logs[47, 68].

Efficiency of LLM Inference. The rapid release of ever-larger LLMs has pushed accuracy to new highs [4, 17, 33], but each leap in parameter count brings a commensurate rise in inference time and resource demand. Although modern LLMs deliver impressive accuracy, their inference cost grows with both model size (number of parameters) and token footprint (prompt length \times call count) [32]. Recent benchmarking shows that end-to-end latency grows from well under a second for 6–7 B-parameter models to more than 30 seconds when 175 B-parameter models serve the same prompt on the same device, and the energy cost rises proportionally [12]. To keep LLMs usable in production, researchers have explored distillation and quantization to shrink model size [31], prompt compression [38] and retrieval to shorten inputs [73], and caching or speculative decoding to cut repeated calls [8]. Efficiency is particularly critical for latency-sensitive workloads, such as question answering, real-time translation, and anomaly detection, where even a few hundred milliseconds of extra latency degrades user experience or delays alerts [28]. Log parsing shares this constraint in operational monitoring pipelines [26]. Engineers require near-real-time template extraction on millions of lines, so LLM-based log parsing should keep the base model modest and minimize the number of LLM invocations.

2.2 Related Work

Current automated log parsers can be broadly categorized into two types: syntax-based log parsers and semantic-based log parsers. Syntax-based log parsers [14, 15, 18, 24] typically employ heuristic rules or conduct comparisons among logs to identify common components that serve as templates. Semantic-based log parsers [26, 34, 42, 44] focus on analyzing the textual content within logs to distinguish between static and dynamic segments (i.e., using LLMs), thereby deriving the log templates. Semantic-based parsers often require a data-driven approach to better grasp the semantic nuances inherent in the specific system logs they analyze. Below, we discuss related work and the limitations of these two groups of parsers.

Syntax-based Log parsing approaches. Syntax-based log parsers [14, 15, 18, 24] generally utilize manually crafted heuristics or compare syntactic features between logs to extract log templates. Different from general text data, log messages have some unique characteristics. Heuristic-based log parsers extract log templates by identifying features in the logs. For example, AEL [24] uses heuristics to remove potential dynamic variables and extract log templates. Drain [18] employs a fixed-depth parsing tree structure alongside specifically designed parsing rules (i.e., top-k prefix tokens) to identify common templates. However, these log parsers often suffer from decreased accuracy when processing logs that do not conform to the predefined rules.

Logs with the same log template share the same static messages in the log. Based on this observation, several log parsers leverage frequent pattern mining [14, 15] to parse the logs by identifying common textual content within logs. For instance, Spell [15] uses the Longest Common Subsequence to parse logs, and Logram [14] identifies frequent $n - gram$ patterns within logs, using these recurring patterns to parse logs. While these frequent pattern mining-based parsers do not require manually defined rules, the templates they generate are highly dependent on the structure of the input logs. Logs with complex structures may lead to poor frequent pattern mining results, resulting in low parsing accuracy. ***In short, while syntax-based parsers benefit from simplicity and efficiency in identifying common templates, their performance varies depending on the structure of logs.***

Semantic-based log parsing approaches. Semantic-based log parsers [11, 26, 33, 34, 44, 66] use language models to analyze the semantics of the log messages for log parsing. Recently, they have shown superior parsing accuracy compared to syntax-based log parsers, largely due to significant advancements in language models. For instance, models like ChatGPT [33] can analyze the context of log messages and dynamically generate log templates without prior knowledge, enhancing accuracy and adaptability across different log formats. DivLog [66] enhances log parsing by extracting similar logs from a candidate set of labeled logs for in-context learning using GPT-3 [9]. Due to the high cost of commercial LLMs such as ChatGPT, LILAC [26] enhances the efficiency of LLM-based log parsing by incorporating an Adaptive Parsing Cache that stores parsing results. LILAC adopts in-context learning with log-parsing demonstrations (i.e., manually created log templates) for enhanced parsing accuracy. LogParser-LLM [74] improves efficiency by pairing an off-the-shelf LLM with a lightweight template memory, reducing redundant prompts while maintaining accuracy. LUNAR [21] partitions logs and only queries LLM once for logs belonging to the same cluster, which improves the parsing efficiency. Lemur [71] combines entropy-based sampling with chain-of-thought merging to refine LLM outputs, achieving competitive accuracy at a modest computational cost.

Some parsers also aim to use open-source LLMs for log parsing. Hooglle [11] adopted an LLM pre-trained on labeled logs for log parsing. LogPPT [34] utilizes a masked language model (RoBERTa [41]) and adopts few-shot learning to classify tokens in log messages based on few-shot examples. As an initial attempt to apply LLMs for log parsing, LogPPT showed improved accuracy over traditional syntax-based log parsers. LLMParse [44] explores the performance of various LLMs after a few-shot fine-tuning on log parsing. Results indicate that fine-tuning small open-source LLMs with a few demonstrations can also achieve high log parsing accuracy. UNLEASH [35] demonstrates that large pre-trained language models (RoBERTa [41]), when used without task-specific fine-tuning, can outperform existing semantic parsers through better contextual understanding.

Although the results are promising, recent works in semantic-based log parsers have three main limitations: 1) privacy and monetary costs of using commercial LLMs, 2) requiring manually derived log templates for LLMs to learn, and 3) inefficient when parsing large amounts of logs. First, most log parsers are based on commercial LLMs such as ChatGPT, which makes real-world adoption a challenge due to the privacy issues and monetary costs of parsing large volumes of logs. Second, many parsers, especially the ones that aim to improve efficiency and accuracy (e.g., LILAC [26]) or the ones that use smaller open-source models (e.g., LogPPT [34] and LLMParse [44]) require some log-template pairs as the demonstration. Deriving such templates requires significant manual efforts, and the provided demonstrations may affect the parser’s accuracy on logs with unseen templates. Third, since LLMs typically have hundreds of millions or even hundreds of billions of parameters, each inference incurs significant cost. Existing parsers, such as LogPPT [34] and LLMParse [44], require LLMs to process each log message individually. A large number of LLM queries introduces significant overhead, making it difficult to meet the efficiency requirements of practical log parsing tasks.

In this paper, we propose LibreLog that addresses the three above-mentioned limitations. We deployed a relatively small open-source LLM (i.e., Llama3-8B [4]) on log parsing to avoid privacy issues and monetary costs.

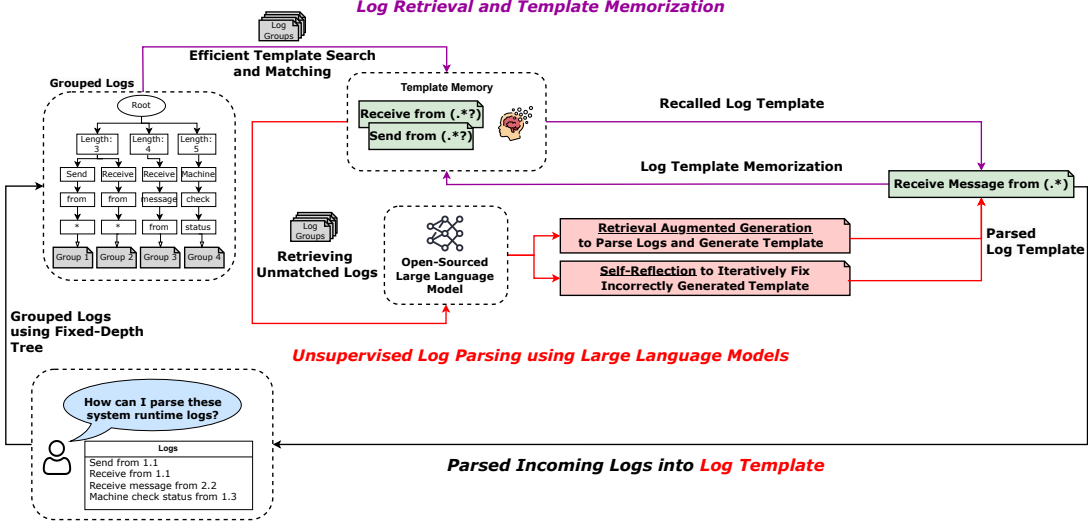


Fig. 2. An overview of LibreLog.

Additionally, LibreLog enhances LLM-based log parsing by capitalizing on the commonalities and variabilities within logs to provide a demonstration-free prompt for the LLM. At the same time, template memory is introduced to store parsed log templates for subsequent log matching to avoid repeated LLM queries and improve efficiency.

3 APPROACH

In this section, we introduce LibreLog, an efficient unsupervised log parser, leveraging memory capabilities and advanced prompting techniques to maximize efficiency and parsing accuracy. LibreLog leverages a smaller-sized open-source LLM to enhance privacy and reduce operational costs. Figure 2 illustrates the overall architecture of LibreLog, which primarily comprises of three components: (i) **log grouping**, which groups logs that share a commonality in their text. Such log groups can then be used as input to LLM to uncover dynamic variables. (ii) An **unsupervised LLM-based log parser** that uses retrieval-augmented generation (RAG), followed by an iterative self-reflection mechanism to accurately parse the grouped logs into log templates. (iii) An **efficient log template memory**, which memorizes the parsed log templates for future query. The core idea is to enhance efficiency by storing parsed log templates in memory, thereby avoiding the need for repeated LLM queries.

3.1 Log Grouping Based on Commonality

LibreLog achieves unsupervised and zero-shot log parsing by first applying an effective grouping strategy. This strategy aims to group logs that share commonality in their static text, yet are different in their dynamic variables. Such log groups can then be used as input to LLMs to generate log templates by prompting LLMs to identify the dynamic variables among logs in the same group. To group the logs, we adapt the efficient unsupervised methodology proposed by Drain [18], which applies a fixed-depth parsing tree and parsing rules (i.e., K prefix tokens) to identify log groups. The fixed depth in our grouping tree provides a structured and predictable framework that enhances efficiency. By limiting the depth, we reduce the complexity of the tree traversal, which speeds up the grouping process.

Our fixed-depth tree implementation for grouping consists of three key steps: (i) group by **length**, (ii) group by **K prefix tokens**, and (iii) group by **token string similarity**. In step (i), we first group the logs based on *token length*, which partitions the logs into subsets of logs that are similar in token length. This initial grouping significantly reduces the computational complexity in the subsequent grouping phases. In step (ii), the grouped logs are then kept at a fixed depth which stores K prefix tokens. Since logs are initially grouped based on *token length*, truncating K prefix tokens (default is the first three tokens of the log) can limit the number of nodes visited during the subsequent traversal process for step (iii), significantly improving grouping efficiency. Before step (iii), it is important to note that we abstract the numerical literals in the logs with a wildcard symbol (*). This is done to prevent the issue of grouping explosion in step (iii), which can make grouping inefficient. Finally, in step (iii), we calculate the similarity between the new logs and the log groups stored in the fixed-depth tree. This step determines whether the incoming log fits into an existing group or necessitates the creation of a new log group. If a suitable group is found based on the similarity threshold, i.e., $\frac{\text{\# of common tokens}}{\text{total number of tokens}} > 0.5$, the log is inserted into existing log groups. If not, a new group is created, and the tree is dynamically updated to accommodate this new log pattern. This adaptive approach ensures that our system evolves with the incoming data, continuously optimizing both the accuracy and efficiency of the log grouping process.

3.2 LLM-based Unsupervised Log Parsing

Our prompts to LLMs contain representative logs (based on variability) retrieved from each log group (from Section 3.1) to guide LLMs in separating dynamic variables and static text. Figure 3 illustrates the prompt template that LibreLog uses. Below, we discuss the composition of our prompt in detail.

Prompt Instruction. In the instruction part of our prompt, we define the goal of the log parsing task to the LLM (highlighted in green in Figure 3). We emphasize that all the provided logs should share one common template that matches all selected logs. This specification is crucial to ensure that the LLM can effectively identify the commonalities and variability within the provided logs, thereby preventing any difficulties in parsing due to inconsistent log templates.

Standardizing LLM Response by Input and Output Example. Since our LLM is not instruction fine-tuned [44], it is crucial to clearly describe our task instruction and include an input-output example in the prompt. This explicit guidance helps the LLM understand the desired input and output formats. As shown in Figure 3, we provide one example to illustrate the input/output form. The example remains unchanged for all systems. This approach effectively guides the LLM in understanding the objective and input-output formats without the need of instruction fine-tuning or labeled data.

Retrieval-Augmented Log Parsing. To parse logs accurately, we select representative logs that showcase variabilities within a log group based on commonality. By presenting the LLM with logs sharing the same structure but varying in dynamic variables, it can more effectively distinguish between fixed and dynamic elements to identify the log template. We developed a retrieval argument generation (RAG) approach based on Jaccard Similarity [63]. Jaccard similarity measures the similarity between two given sets by calculating the ratio of the number of elements (e.g., tokens) in their intersection to the number of elements in their union. For log data, each log is split into a set of tokens (i.e., words), and then these tokens are used to determine the sizes of the intersection and union. The resulting ratio is the Jaccard similarity between two given logs, with a ratio closer to one indicating higher similarity. We aim to identify the logs with the greatest variability within the same group. Hence, we select logs with the lowest Jaccard similarity score. This approach helps create accurate log templates by focusing on logs that are most indicative of the entire group’s characteristics.

Our selection process starts by selecting the longest log (based on the number of characters) within the group as the initial reference. We then calculate the Jaccard similarity between this log and every other log in the group. The log with the lowest similarity to the reference log is added to the selection set. We continue computing

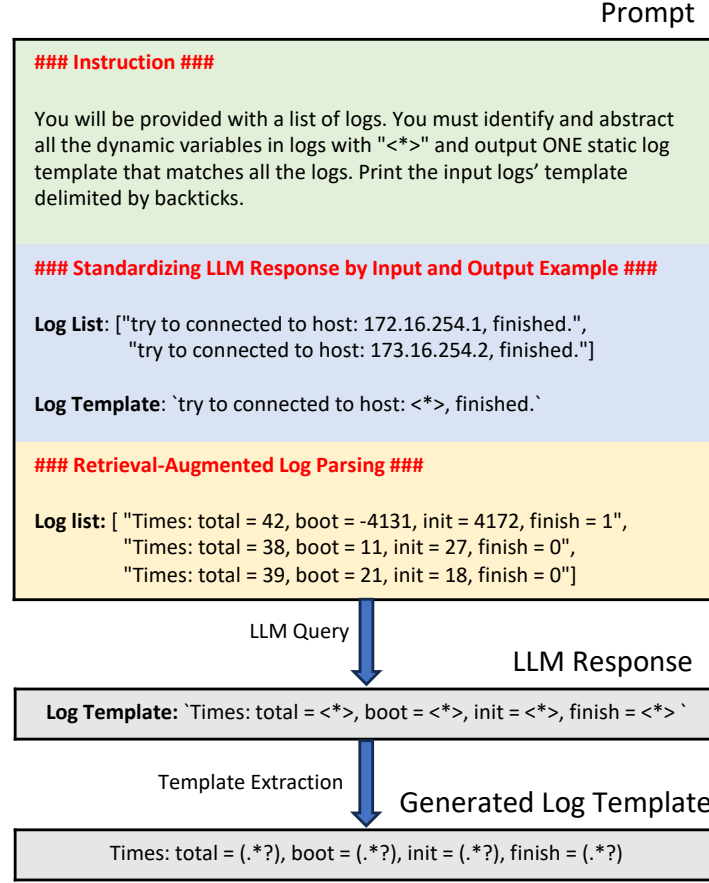


Fig. 3. Example of the prompt template used for LibreLog. The green block illustrates the task instruction provided to the LLM. The blue block highlights the input and output examples used to standardize the log response format. The yellow block depicts the retrieval-augmented selection process that enhances log parsing accuracy by incorporating representative variability.

pairwise Jaccard similarity between the selected logs and the remaining unselected logs, sequentially adding the log with the lowest similarity. This iterative process is repeated until K (default $K = 3$) logs have been selected, ensuring the selected logs effectively represent both the commonality and diversity within the log group. Given the computation costs and to ensure efficiency, we randomly select at most 200 logs from each group (or all the logs if the number is less than 200) for our log selection process.

Specifically, the logs selected from the log group are listed in the format of a Python list within the prompt for parsing. We use a prefix (i.e., 'Log list:') to help the LLM identify the logs that require parsing (highlighted in yellow in Figure 3). This consistency in input format, mirroring the "Input and Output Example", also guides the LLM to respond with the log template in a fixed format as demonstrated in the example, facilitating accurate template generation and extraction.

Post-processing Template Standardization. We use a post-processing technique to further standardize the log template generated by LLM. We employ string manipulation techniques to remove non-template content from the response (i.e., prefixes and backticks). To facilitate the verification of the accuracy of log templates, we replace the placeholder "<*>" within the templates with the regular expression pattern "(. *?)". The regex template enables a direct matching process when comparing the generated templates with logs, and can be directly applied to abstract logs.

Self-Reflection for Verifying Log Template. After generating a log template, we verify whether the template can match each log within the group. If a log is correctly matched by a log template, we consider it to be parsed successfully. The log template is then added to the log template memory for future use. After all logs in the group have been checked, any unparsed logs undergo a self-reflection process [59], which aims to revise the templates and improve parsing results. Similar to the initial parsing attempt, we first select these unparsed logs and then utilize the prompt described in Figure 3 to generate a new log template using LLMs. This step is repeated until all logs in the group can be matched/parsed by the generated templates. Note that, to prevent the LLM from entering a parsing loop (i.e., repeatedly generating incorrect templates), we limit the self-reflection process to three iterations.

3.3 Template Memory for Efficient Log Parsing

Repeatedly using LLM to parse logs with identical groupings and templates significantly increases the frequency of LLM queries, thereby reducing the efficiency of the log parsing process. To address this issue, we introduce **log template memory** in LibreLog, which stores the parsed log templates for future parsing, avoiding redundant LLM queries.

Efficient Log Template Memory Search and Matching. When a log group requires parsing, we first check whether a matching log template exists within the memory. If some logs within the group find a matching template in the memory, we apply this log template to parse the logs, mitigating the need for LLM queries. However, it is possible that some logs within the same group may match while others may not (e.g., due to limitations in the grouping step or limitation of the log template). Hence, the logs that remain unparsed are then sent to LLM for parsing. The new log template generated from this process is then added to the **log template memory** for future reference. This design significantly reduces the number of LLM queries during the log parsing process.

To efficiently utilize log templates in the memory, there is a need for an efficient **search** mechanism to verify whether or not the given logs match existing log templates in the memory. This is crucial since the memory can be large, consisting of many log templates. For every log, we need potentially at most N searches for N log templates. To improve efficiency, we put forward one key observation: the token length of log templates is always less than or equal to that of the original logs, as multiple tokens may be treated as a single variable during log parsing. For instance, consider the log 'sent 100 bytes data'. After parsing, the corresponding log template is generated as 'sent <*> data'. The original log consists of four tokens, whereas the parsed template has three. This reduction in token count occurs because '100 bytes' is treated as a single variable, thus decreasing the overall length of the template compared to the original log. Consequently, when searching for log templates in the memory, we first sort the templates based on the number of tokens. This sorting allows us to efficiently check new logs by first calculating the token length of the log to be parsed, then using binary search to find all templates with a token count less than or equal to the log length. This design reduces the number of match checks required from $O(N)$ to $O(\log N)$, thereby enhancing the efficiency of the search process.

Our log-template matching process is efficient. Unlike traditional log templates that use placeholders (i.e., "<*>") to abstract dynamic variables within logs, we store log templates in memory as regular expression patterns (i.e., use "(. *?)" instead of placeholders). This adjustment allows us to use regular expressions to efficiently verify whether logs match with log templates in memory and improve matching efficiency.

4 EXPERIMENT SETUP

In this section, we discuss our experiment setup to answer our research questions and LibreLog’s implementation details.

Studied Dataset. We conduct our experiment on the log parsing benchmark LogHub-2.0 provided by He et al. [19, 25]. This benchmark contains logs from 14 open-source systems of different types, such as distributed systems, supercomputer systems, and server-side applications. LogHub is widely used to evaluate and compare the accuracy of log parsers [14, 15, 18, 26, 34, 44]. Compared to LogHub-1.0 [19], the number of logs has increased significantly in LogHub-2.0, increasing from 28K (2K logs per system) to more than 50 million logs with a total of 3,488 different log templates. LogHub-2.0 also provides the groundtruth log template for each log. With this large-scale LogHub-2.0 dataset, researchers can better evaluate the efficiency and effectiveness of log parsers [25, 26].

Environment and Implementation. Our experiments were conducted on an Ubuntu server with an NVIDIA Tesla A100 GPU, AMD EPYC 7763 64-core CPU, and 256GB RAM using Python 3.9. We execute the baselines using their default parameters under the same environment to compare the efficiency. We use Llama3 8B [4] for LibreLog’s underlying LLM because it is a relatively small yet powerful model, balancing performance and efficiency effectively. We set the temperature value to 0 to improve the stability of the model output. Note that it is easy to switch to other LLMs. In RQ4, we evaluate LibreLog by replacing Llama3 with other open-source LLMs.

Evaluation Metrics for Log Parsing. Following prior studies [14, 18, 26, 29, 34, 44], we use four most commonly used metrics to evaluate the effectiveness of log parsers: Group Accuracy and Parsing Accuracy.

Group Accuracy (GA): Grouping Accuracy [76] is a metric used in log parsing to evaluate the extent to which log messages belonging to the same template are correctly grouped together by a parser. GA is defined as the ratio of correctly grouped log messages to the total number of log messages. For a log message to be considered correctly grouped, it must be assigned to the same group as other log messages that share the same underlying template. High GA indicates that the parser can effectively discern patterns within the log data and group similar log messages together. This can be crucial for various downstream log analysis tasks such as anomaly detection [30, 58]. Despite its usefulness, GA has limitations. GA can remain high even if the parsed templates are flawed. Namely, a high GA score might obscure errors in dynamic variable extraction and template identification within the logs, leading to a misleading perception of overall parsing accuracy.

Parsing Accuracy (PA): Parsing Accuracy (PA) [42] complements GA and is calculated as the ratio of accurately parsed log messages to the total number of log messages. For a log message to be deemed correctly parsed, both extracted static text and dynamic variables must match exactly with those specified in the ground truth. PA is a stricter metric because it requires a comprehensive match of all log components, not just their correct grouping. This distinction is crucial, as GA primarily evaluates the correct clustering of logs, while PA ensures precise parsing accuracy at the individual log message level. Precise log parsing of the variables can also significantly impact the effectiveness of downstream log-based analyses [37].

F1 score of Grouping Accuracy (FGA): FGA is a *template-level* metric that evaluates a parser’s clustering quality. Unlike GA, FGA scores accuracy per template rather than per line, preventing frequent templates from hiding errors on rare ones. Let N_g denote the number of ground-truth templates, N_p the number of templates produced by the parser, and N_c the number of predicted templates whose entire log-message set is identical to that of one ground-truth template. The precision and recall of grouping accuracy are

$$P_{GA} = \frac{N_c}{N_p}, \quad R_{GA} = \frac{N_c}{N_g},$$

Table 1. Accuracy comparison for the state-of-the-art parsers and LibreLog.

Dataset	AEL				Drain				UniParser				LUNAR				LILAC				LLMParser _{TSBase}				LibreLog			
	GA	PA	FGA	FTA	GA	PA	FGA	FTA	GA	PA	FGA	FTA	GA	PA	FGA	FTA	GA	PA	FGA	FTA	GA	PA	FGA	FTA	GA	PA	FGA	FTA
HDFS	99.9	62.1	76.4	56.2	99.9	62.1	93.5	60.9	100	94.8	96.8	58.1	100	94.9	92.5	88.2	100	94.8	88.9	77.8	82.9	96.6	41.3	13.0	100	100	92.9	79.5
Hadoop	82.3	53.5	11.7	5.8	92.1	54.1	78.5	38.4	69.1	88.9	62.8	47.6	80.5	79.4	51.7	42.0	92.4	78.5	93.0	56.8	83.8	83.7	74.6	45.1	96.3	87.1	94.0	75.5
Spark	-	-	-	-	88.8	39.4	86.1	41.2	85.4	79.5	2.0	1.2	87.6	91.2	86.9	55.4	87.0	70.8	84.2	51.8	25.9	42.3	38.1	28.6	85.9	88.9	83.2	69.0
Zookeeper	99.6	84.2	78.8	46.5	99.4	84.3	90.4	61.4	98.8	98.8	66.1	51.0	99.3	37.4	88.5	66.7	99.7	37.8	96.0	74.3	71.9	84.1	73.0	12.2	99.3	85.0	97.4	86.3
BGL	91.5	40.6	58.7	16.5	91.9	40.7	62.4	19.3	91.8	94.9	62.0	21.9	83.7	50.3	84.6	59.1	83.3	82.4	84.8	59.3	14.4	24.4	56.9	13.0	90.2	92.9	80.3	71.6
HPC	74.8	74.1	20.1	13.6	79.3	72.1	30.9	15.2	77.7	94.1	66.0	35.1	80.9	69.2	74.9	66.7	84.5	73.5	67.6	57.4	64.2	70.7	35.1	14.2	84.4	97.3	80.5	60.9
Thunderbird	78.6	16.3	11.6	3.5	83.1	21.6	23.7	7.1	57.9	65.4	68.2	29.0	80.4	51.4	82.3	52.9	79.4	38.6	25.4	11.9	57.9	34.7	75.2	13.6	87.0	69.4	84.3	54.0
Linux	91.6	8.2	80.6	21.7	68.6	11.1	77.8	25.9	28.5	16.4	45.1	23.2	70.7	78.5	82.0	59.5	76.4	70.0	76.8	42.6	50.0	88.0	89.9	36.9	91.2	90.2	74.1	60.7
HealthApp	72.5	31.1	0.8	0.3	86.2	31.2	1.0	0.4	46.1	81.7	74.5	46.2	90.6	92.8	94.6	67.7	99.3	67.3	97.1	72.4	83.6	96.7	91.0	38.9	86.2	97.4	95.8	87.7
Apache	100	72.7	100	51.7	100	72.7	100	51.7	94.8	94.2	68.7	26.9	100	99.2	100	69.0	99.7	99.2	94.9	71.2	86.8	99.0	86.2	49.2	100	99.6	100	75.9
Proxifier	97.4	67.7	66.7	41.7	69.2	68.8	20.6	17.6	50.9	63.4	28.6	45.7	98.9	100	87.0	95.7	50.6	77.8	75.0	91.7	82.8	98.2	63.6	42.4	51.0	89.7	35.3	40.0
OpenSSH	70.5	36.4	68.9	33.3	70.7	58.6	87.2	48.7	27.5	28.0	0.9	0.9	73.8	47.5	84.2	44.7	74.8	65.5	88.9	58.3	21.8	78.1	63.2	18.1	86.8	49.6	85.3	47.2
OpenStack	74.3	2.9	68.2	16.5	75.2	2.9	0.7	0.2	100	51.6	96.9	28.9	87.2	80.5	55.8	41.3	52.4	48.6	95.7	83.0	95.2	94.1	85.4	14.6	81.1	83.1	68.9	55.9
Mac	79.7	24.5	79.3	20.5	76.1	35.7	22.9	6.9	73.7	68.8	69.9	28.3	87.7	51.9	85.6	44.2	80.2	42.2	83.8	39.0	67.1	61.3	69.3	15.0	81.4	65.4	80.4	46.9
Average	85.6	44.2	55.5	25.2	84.3	46.8	55.4	28.2	71.6	72.9	57.8	31.7	87.2	73.2	82.2	60.9	82.8	67.6	82.3	60.5	63.5	75.1	67.4	25.4	87.2	85.4	82.3	65.1

Note: The highest values of GA, PA, FGA, and FTA for each system are highlighted in **bold**. The accuracy of AEL on the Spark dataset is excluded because it cannot complete parsing the whole dataset after running for 10 days.

and the final score is their harmonic mean:

$$FGA = \frac{2P_{GA}R_{GA}}{P_{GA} + R_{GA}}.$$

Thus, a template is counted as correct only when its log messages perfectly coincide with those of a single oracle template [25].

F1 score of Template Accuracy (FTA): FTA is a fine-grained *template-level* metric that extends FGA by additionally validating the template string itself. Whereas FGA cares only about which messages belong together, FTA also checks that the template’s static tokens and wildcards exactly match the oracle. It therefore penalizes boundary errors that GA, PA, and even FGA may overlook. A predicted template is considered correct when it satisfies both of the following: (i) its log-message set is identical to that of one ground-truth template; (ii) every static token and every variable placeholder in the template text matches the oracle exactly.

Denote by N_c^{TA} the number of templates meeting these stricter conditions. Then

$$P_{TA} = \frac{N_c^{TA}}{N_p}, \quad R_{TA} = \frac{N_c^{TA}}{N_g}, \quad FTA = \frac{2P_{TA}R_{TA}}{P_{TA} + R_{TA}}.$$

Because FTA scores at the template level and also verifies every static and dynamic token, a handful of high-frequency lines can no longer mask structural errors. Hence, it offers the most balanced measure, subsuming the grouping focus of GA/FGA and the message-level precision of PA [26].

5 EVALUATION

In this section, we evaluate LibreLog by answering four research questions (RQs).

RQ1: What is the effectiveness of LibreLog?

Motivation. Accuracy is the most critical factor for evaluating the effectiveness of log parsers. High accuracy in log parsing aids downstream log analysis tasks [30, 58]. In this RQ, we study the effectiveness of LibreLog.

Approach. We compare LibreLog with other state-of-the-art log parsers, including AEL, Drain, UniParser, LUNAR, LILAC, and LLMParser_{TSBase}. AEL [24] and Drain [18] are two leading traditional syntax-based approaches that are efficient and perform better than most other syntax-based parsers [25, 70, 76]. UniParser [42] is a deep learning-based log parser that employs a long short-term memory (LSTM) model trained on labeled logs,

demonstrating promising parsing accuracy [25]. LUNAR [21], LILAC [26] and LLMParser_{T5Base} [44] are recently proposed LLM-based parsers with high parsing accuracy. Since LUNAR and LILAC use ChatGPT as the underlying LLM, for a fair comparison, we replace ChatGPT with the same open-source LLM (Llama3-8B [4]) that LibreLog uses. We use T5-base [13] (240M parameters) as the LLM for LLMParser by following the prior work. Note that UniParser, LILAC, and LLMParser require manually derived log templates as a few-shot demonstrations. We follow the steps described in the papers to obtain these demonstrations. We evaluate the parsers using the LogHub-2.0 dataset and report GA, PA, FGA and FTA.

Results. Table 1 shows the GA, PA, FGA, and FTA for each log parser across different systems. LibreLog achieved the highest values on all four metrics for most systems, indicating superior performance in both grouping and template-level parsing. Across all systems, LibreLog achieved the highest performance, with an average GA of 87.2, PA of 85.4, FGA of 82.3, and FTA of 65.1, surpassing all other parsers.

LibreLog shows superior GA, PA, FGA, and FTA compared to the unsupervised LLM-based parser – LUNAR and semi-supervised LLM-based parser – LILAC. Compared to LibreLog, LILAC demonstrated lower accuracy with a GA of 82.8, PA of 67.6, and FTA of 60.5. LUNAR showed lower accuracy with a PA of 73.2, FGA of 82.2, and FTA of 60.9. Although LUNAR and LibreLog share some similar designs, LibreLog achieves higher PA and FTA, possibly due to having a self-reflection mechanism to refine variable extraction and template generation. LILAC uses manually labeled logs as demonstrations for in-context learning to enhance parsing accuracy. However, when utilizing less powerful open-source LLMs with smaller parameter sizes (i.e., as opposed to ChatGPT), LILAC’s performance declines significantly due to the limited ability of these models to capture complex log patterns with only a few demonstrations. Consequently, this can lead to inaccurate parsing of variables within the logs (a PA of 67.6 and an FTA of 60.5, while LibreLog’s PA and FTA are 85.4 and 65.1, respectively). Unlike LILAC and LLMParser_{T5Base}, LibreLog is an unsupervised log parser, eliminating the need for labeled logs to enhance the LLM’s log parsing capabilities. The performance of LibreLog is not dependent on the number of labeled logs, thus avoiding the limitations faced by semi-supervised approaches that require labeled logs for fine-tuning or in-context learning.

LibreLog’s group-specific prompting supplies richer context than LILAC’s fixed demonstrations, leading to higher PA and FTA. LILAC feeds the LLM a static set of 32 demonstration logs for every query; these examples remain unchanged even though 12 / 14 systems contain far more than 32 distinct templates. Consequently, many low-frequency templates are never illustrated, and the model tends to over-generalize, depressing PA and FTA. LibreLog first performs unsupervised grouping and then selects examples from within each group. The average and median group sizes are 25k and 36 from LibreLog grouping stage, providing sufficient diversity within groups for prompt selection. This adaptive, diverse context lets LibreLog infer variable boundaries more precisely, explaining its consistent PA and FTA advantages over LILAC without requiring any manually curated demonstrations.

Among all four LLM-based log parsers, LLMParser_{T5Base} shows the lowest GA and FGA, likely because the limited number of fine-tuning samples makes it hard to generalize to large-scale datasets. Among the four LLM-based log parsers (LLMParser_{T5Base}, LUNAR, LILAC, and LibreLog), LLMParser_{T5Base} exhibits the lowest GA of 63.5, the lowest FGA of 67.4, and the second-highest PA of 75.1. When parsing large-scale datasets, logs may display many variations even when they share the same template. Given that LLMParser_{T5Base} is fine-tuned using a small, labeled sample set from the target system, the limited number of log samples likely contributes to its inability to robustly identify logs with the same template across all instances and, thus, lower GA and FGA. This limitation is particularly evident in systems with more logs, such as BGL and Spark, where LLMParser_{T5Base} struggles to achieve high GA (14.4 and 25.9, respectively). Nevertheless, it still identifies dynamic variables with the second-highest PA among all five parsers, underscoring the potential of LLM-based approaches.

Table 2. Number of logs and parsing time, in seconds, for the state-of-the-art (first six columns) and LibreLog.

	Log count	AEL	Drain	UniParser	LUNAR	LILAC	LLMParser _{T5Base}	LibreLog			
		Total time	Total time	Total time	Total time	Total time	Total time	Total time	LLM query	Grouping	Mem. search
HDFS	11,167,740	5,711.52	1,343.56	4,953.80	4,065.48	1,162.20	148,097.72	1,252.62	273.67	867.36	111.59
Hadoop	179,993	361.54	19.54	83.30	2,078.04	4,747.52	4,034.32	285.76	268.81	11.95	5.01
Spark	16,075,117	10 days+	1,539.88	8,941.60	6,956.66	3,346.08	225,046.88	1,752.40	631.66	764.12	356.62
Zookeeper	74,273	3.22	7.12	50.60	300.14	1,702.46	1,585.52	52.23	47.06	4.75	0.42
BGL	4,631,261	29,917.35	501.09	1,820.00	3,597.55	8,624.70	90,526.27	1,244.64	857.82	298.23	88.59
HPC	429,987	18.00	39.02	109.00	923.51	388.88	4,634.87	539.76	510.97	27.45	1.33
Thunderbird	16,601,745	25,199.44	2,132.20	15,503.20	9,245.80	16,316.03	421,864.78	8,659.29	5,343.56	1,466.77	1,848.96
Linux	23,921	4.53	2.61	10.10	1,473.88	2,374.83	1,031.82	216.03	213.42	1.78	0.82
HealthApp	212,394	976.74	17.88	71.40	450.69	1,182.27	3,139.20	103.33	85.25	10.38	7.70
Apache	51,977	3.20	5.42	21.00	103.24	122.79	1,056.53	18.92	15.19	3.36	0.37
Proxifier	21,320	1.69	2.96	14.70	277.06	681.65	821.44	871.52	868.99	2.47	0.07
OpenSSH	638,946	1,338.67	74.12	328.30	373.39	1,134.35	15,262.29	89.37	36.94	49.00	3.44
OpenStack	207,632	30.28	60.18	138.70	1,225.25	1,260.64	7,558.24	377.64	330.66	44.88	2.09
Mac	100,314	10.79	16.94	67.00	7,125.99	15,930.81	4,873.72	5,935.77	5,922.19	9.26	4.32
Average	3,601,187	4,890.54	411.61	2,293.76	2,728.33	4,212.52	66,395.26	1,528.52	1,100.44	254.41	173.67
Total	50,416,620	17.66 (h)	1.60 (h)	8.92 (h)	10.61 (h)	16.38 (h)	258.20 (h)	5.94 (h)	4.28 (h)	0.99 (h)	0.68 (h)

Syntax-based log parsers generally have significantly lower PA and FTA compared to LLM-based parsers, showing challenges in accurately identifying variables. While AEL and Drain, as syntax-based parsers, show results similar to each other, they both exhibit lower GA compared to LibreLog (1.6 % and 3.5 % lower, respectively) and significantly lower PA (41.2 % and 45.2 % lower) as well as FTA (39.9 % and 36.9 % lower). This performance disparity is likely linked to their heuristic-based nature, which relies on predefined rules to capture log features. While these rules can effectively classify logs with similar features, achieving reasonable GAs, their generic nature often fails to accurately recognize variables within different log templates, leading to poor PA and FTA. In contrast, LibreLog leverages pre-grouping and uses memory mechanisms to achieve high GA/FGA, and its LLM-based parsing process accurately identifies variables within grouped logs, resulting in superior PA and FTA.

LibreLog achieves superior GA, PA, FGA, and FTA compared to state-of-the-art parsers. Despite not relying on labeled logs, LibreLog outperforms other LLM-based parsers that are semi-supervised. Additionally, LibreLog significantly enhances PA compared to syntax-based approaches.

RQ2: What is the efficiency of LibreLog?

Motivation. Efficiency is crucial in log parsing since it directly impacts the practical usability of the parser in real-world applications. In this RQ, we study the parsers’ efficiency.

Approach. We measure the total parsing time required by LibreLog and its individual components (i.e., LLM queries, grouping, and memory search), and the four baseline parsers to process logs from the LogHub-2.0 dataset.

Results. *LibreLog is 1.7, 2.7, and 40 times faster than LUNAR, LILAC, and LLMParser_{T5Base}, respectively.* Table 2 shows the parsing time for each log parser across different systems. LibreLog spends a total of 5.94 hours to parse logs from all 14 systems (50 million logs), which is significantly faster than other LLM-based parsers: LUNAR (10.6 hours), LILAC (16 hours) and LLMParser_{T5Base} (258 hours). The parsing time for LibreLog is mainly occupied by the LLM query time, which accounts for 72.05% of the total processing time, followed by the grouping time, which constitutes 16.67% of the overall duration. LUNAR partitions logs and queries LLM once per partition, reducing LLM calls and achieving higher parsing efficiency than LLMParser_{T5Base} and LILAC, though it is still slower than LibreLog. LLMParser_{T5Base} is the slowest among all LLM-based parsers because it processes each log individually, and the vast quantity of logs linearly increases the number of model queries required. Even with a relatively lightweight model like T5-base, which has only 240 million parameters, querying

to parse the logs individually is still slow and impractical for real-world applications. LILAC, with its cache design, eliminates the need to parse each log individually through an LLM, significantly speeding up the process compared to `LLMParserT5Base`. However, LILAC still requires frequent model queries to update the templates in the cache, which limits its efficiency. In contrast, LibreLog optimizes parsing times through its grouping and memory features, resulting in superior efficiency.

AEL exhibits significant efficiency issues when parsing logs beyond certain sizes, while Drain maintains high efficiency across all datasets. AEL can parse datasets with fewer than 100K logs within seconds but requires several hours or even days for datasets with over one million logs (e.g., we stopped AEL after running for 10 days when parsing the 16 million logs from Spark). This inefficiency is due to AEL’s reliance on extensive comparisons between logs and identified templates, where the parsing time grows exponentially with respect to the number of logs and log templates. In contrast, Drain, which uses a fixed-depth parsing tree, is the most efficient parser. LibreLog uses a grouping method similar to Drain’s, with a total grouping time amounting to 0.99 hours, which is less than Drain’s total parsing time of 1.6 hours. This highlights the efficiency of LibreLog’s grouping process. While there is a slight slowdown due to the additional processing involved (5.94 hours compared to Drain’s 1.6 hours), LibreLog shows superior parsing effectiveness compared to Drain and is the second fastest log parser among the evaluated parsers.

LibreLog enhances its efficiency by utilizing grouping and memory components, which reduces the number of LLM queries. LibreLog demonstrates the highest efficiency across LLM-based parsers.

RQ3: How do different settings impact the result of LibreLog?

Motivation. LibreLog implements multiple components to achieve effective and efficient log parsing. In this RQ, we explore how various settings and configurations affect the performance of LibreLog.

Approach. There are three general components in LibreLog that can be adjusted or replaced: log selection from each group for prompting, the number of selected logs, and the inclusion or exclusion of self-reflection processes. To select diverse logs from the log group, we use Jaccard similarity to measure the similarity between every log pair. In this RQ, we also try random sampling and cosine similarity. Furthermore, we evaluate how changing the number of selected logs from 1 to 10 impacts the effectiveness. Finally, we compare the effect of removing the self-reflection component on the efficiency and effectiveness of LibreLog.

Results. *Selecting representative logs based on Jaccard similarity outperforms using cosine similarity and random sampling.* Table 3 reports the total time, GA, PA, FGA, and FTA of LibreLog when the log selection step is replaced with cosine similarity or random sampling. When employing cosine similarity, GA, PA, FGA, and FTA decline by 2.5%, 4.8%, 2.6%, and 2.3%, respectively, compared to using Jaccard similarity. This indicates that although cosine similarity is shown to be an effective similarity metric for text data [60], it does not necessarily select logs that are representative enough for LLM to generalize log templates. However, we notice a slight reduction in execution time (3.3%) when using cosine similarity. Similarly, using random sampling further reduces the processing time (by 8.2%), but the lack of diversity lowers GA, PA, FGA, and FTA to 84.9%, 80.6%, 80.7%, and 64.7%, respectively.

Although the self-reflection mechanism requires additional processing time, it significantly enhances the parsing results of LibreLog. Table 3 compares full version LibreLog and LibreLog without self-reflection in the total execution time, GA, PA, FGA, and FTA. Excluding the self-reflection component from LibreLog results in a 44.6% reduction in parsing time (from around six to three hours). However, removing self-reflection greatly decreases GA, PA, FGA, and FTA by 7.1%, 9.5%, 4.5%, and 3.4%, respectively. This shows that self-reflection significantly enhances the parsing effectiveness of LibreLog, although at the expense of increased overhead due

Table 3. LibreLog performance under different settings. The numbers in the parenthesis indicate the percentage difference compared to the full version of LibreLog.

	Total Time (h)	GA	PA	FGA	FTA
LibreLog	5.944	87.2	85.9	82.3	65.1
w/ cosine similarity	5.745 (↓3.3%)	85.0 (↓2.5%)	81.8 (↓4.8%)	80.2 (↓2.6%)	63.6 (↓2.3%)
w/ random sampling	5.458 (↓8.2%)	84.9 (↓2.6%)	80.6 (↓6.2%)	80.7 (↓1.9%)	64.7 (↓0.6%)
w/o self-reflection	3.292 (↓44.6%)	81.0 (↓7.1%)	77.7 (↓9.5%)	78.6 (↓4.5%)	62.9 (↓3.4%)

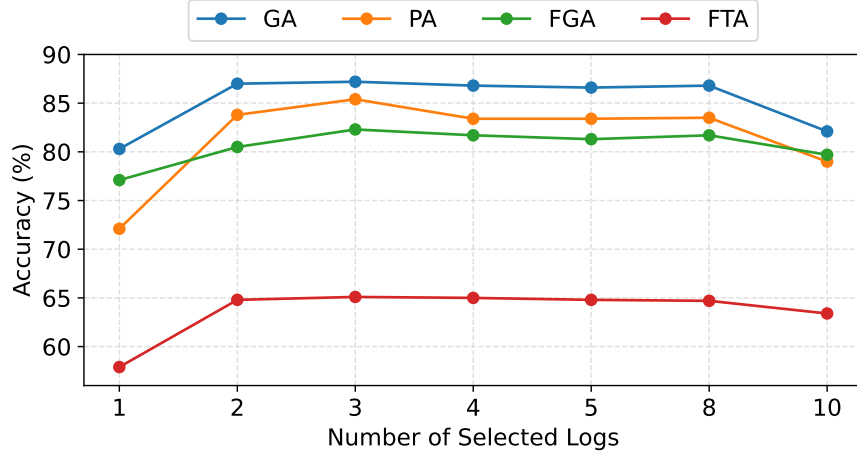


Fig. 4. Accuracy of LibreLog using different numbers of selected logs in the prompt.

to additional LLM queries. Therefore, in practical applications, the inclusion of the self-reflection component in LibreLog can be determined based on the specific needs of effectiveness or efficiency.

During retrieval augmented log parsing, varying the number of selected logs affects the performance of LibreLog. Retrieving three logs into the prompt yields the highest effectiveness. Figure 4 shows the LibreLog performance with variations in the number of logs from a group retrieved into the prompts. LibreLog maintains high GA, PA, FGA, and FTA across the range, with all four metrics peaking when the sample size is three. Notably, when the sample size is reduced to one, GA and PA fall to about 80% and 70%, respectively, and FGA/FTA drop by a comparable margin. This reduction highlights the challenges LLM faces in parsing logs accurately without sufficient comparative data, such as multiple log comparisons or labeled logs. Increasing the sample size from one to two markedly improves every metric, with the best results reached at three examples. However, further increases in sample size from three to eight result in slight decreases: GA and PA stabilize around 86.5% and 83.5%, while FGA and FTA level off as well. This suggests that more log samples may introduce noise, subsequently lowering performance [75]. Importantly, when the sample size reaches 10, all four metrics, GA, PA, FGA, and FTA, decrease compared to a sample size of eight. This decrease is attributed to prompt truncation caused by an overload of retrieved logs that exceeds the LLM’s context window, resulting in incomplete input data.

Using Jaccard similarity for log selection and LLM self-reflection enhances the parsing result, although they come with added overhead. Retrieving more logs within the prompt does not necessarily increase effectiveness; in fact, the optimal number of logs enables LibreLog to reach peak accuracy is three.

RQ4: What is the effectiveness of LibreLog with different LLMs?

Motivation. Unlike previous parsers [26, 33, 66] that are based on commercial LLMs, LibreLog employs an open-source LLM to mitigate privacy concerns and monetary costs. Different LLMs exhibit varying capabilities due to their distinct architectures and pre-training data. In this RQ, we evaluate the accuracy and efficiency of LibreLog across various open-source LLMs.

Approach. We selected three other open-source models (in addition to Llama3-8B) with similar parameter sizes to compare the log parsing accuracy and efficiency, including Mistral-7B [23], CodeGemma-7B [64], and ChatGLM3-6B [16]. These models are commonly used in research and practice. Mistral-7B shows strong text generation capabilities in small model sizes. CodeGemma-7B is pre-trained on code repositories and tailored for code-related tasks. ChatGLM3-6B is known for its bilingual conversational abilities.

Results. Table 4 shows the parsing performance using various LLMs, while Llama3-8B achieved the best overall results.

Compared to Llama3-8B, Mistral-7B requires a slightly longer parsing time and achieves a similar GA and FGA, but shows a noticeable decline in PA and FTA. Mistral-7B is a general model with training objectives and parameter sizes similar to Llama3-8B. However, it exhibits a lower PA (-15.1%) and a lower FTA (-10.3%), while showing comparable results in parsing time, GA (+0.5%), and FGA (+3.4%). This discrepancy in PA and FTA may be attributed to Llama3-8B’s enhanced pre-training data, which includes more code [4], and its slightly larger parameter size. These factors likely contribute to Llama3-8B’s superior ability to abstract variables within logs.

CodeGemma-7B has a better parsing speed, but GA and PA decline relative to Llama3-8B, while FGA and FTA rise slightly. CodeGemma-7B completes parsing of all logs faster by 28.5% than Llama3-8B (total time 4.25 h vs. 5.94 h). Its GA and PA drop by 6.7% and 12.5%, respectively, indicating that the model struggles to abstract variables as precisely as Llama3-8B. Interestingly, FGA and FTA increase by 0.5% and 3.4%, suggesting a marginal improvement in template-level grouping and fidelity. Nevertheless, using CodeGemma-7B as the base LLM for LibreLog still achieves higher GA, PA, FGA, and FTA than the other LLM-based parsers, LILAC and LLMParser_{T5Base}.

As a conversational model, ChatGLM3-6B shows the worst result in parsing effectiveness and efficiency. ChatGLM3-6B, pre-trained on a bilingual corpus in Chinese and English and optimized for conversations, does not include code in its pre-training data, which may have caused its bad parsing ability. This prevents ChatGLM3-6B from generating accurate log templates that can match the logs, necessitating increased model queries for self-reflection. Consequently, the parsing time for ChatGLM3-6B significantly increases by 145.1% compared to Llama3. Despite undergoing extensive self-reflection, ChatGLM3-6B still fails to generate correct log templates. This leads to inferior results in both effectiveness and efficiency compared to other models, illustrating a clear disparity in performance when the pre-training background of the model does not match the specific task requirements. Future studies should consider using LLMs trained or fine-tuned using code or log-related data.

Replacing the LLM leads to variations in effectiveness and performance. Among the four open-source models of similar sizes, Llama3-8B shows the best overall results.

Table 4. Parsing performance of LibreLog using different LLMs.

	Tota Time (h)	GA	PA	FGA	FTA
Llama3-8B	5.94	87.2	85.9	82.3	65.1
Mistral-7B	6.78 (↑14.1%)	87.6 (↑0.5%)	72.9 (↓15.1%)	85.1 (↑3.4%)	58.4 (↓10.3%)
CodeGemma-7B	4.25 (↓28.5%)	81.4 (↓6.7%)	75.2 (↓12.5%)	82.7 (↑0.5%)	67.3 (↑3.4%)
ChatGLM3-6B	14.56 (↑145.1%)	83.7 (↓4.0%)	60.0 (↓30.2%)	79.7 (↓3.2%)	45.3 (↓30.4%)

RQ5: How effective are different techniques for grouping logs?

Motivation. As we mentioned in Section 3.1, LibreLog currently groups log lines with a *fixed-depth tree* that sequentially applies length, shared-prefix, and string-similarity checks. This strategy has proven both fast and effective, yet there are still some other grouping techniques that can be applied to logs. Specifically, density-based clustering algorithms such as DBSCAN [56] and HDBSCAN [46] are good alternative candidates since they can discover clusters of arbitrary shape without requiring hyperparameters such as the number of clusters. To assess whether such data-driven grouping can further enhance overall parsing quality, or at least validate the robustness of our current design, we replace the tree-based grouping with these alternatives and compare their impact on accuracy and efficiency.

Approach. To examine how alternative grouping strategies influence LibreLog, we keep every other component unchanged and replace the fixed-depth tree with clustering-based methods. The implementation of clustering includes three steps: *vectorization*, *dimensionality reduction*, and *density-based clustering*. This design decouples representation learning from clustering, permits the use of model-agnostic density algorithms, and keeps the computation tractable for large datasets.

- **Vectorization.** We first convert every log into a numerical vector, which is suitable for downstream clustering. Because tokens inside logs vary in importance [72], we adopt term-frequency / inverse-document-frequency (TF-IDF) [53] to vectorize the logs. For each token in a log, its *Term Frequency* (TF) measures how important it is within that log. $TF(\text{token}) = \frac{\# \text{token}}{\# \text{total}}$, where $\# \text{token}$ is the count of the target token and $\# \text{total}$ is the total number of tokens in the log. Tokens that appear in many different logs are less informative and too common to distinguish distinct log lines. Therefore, we down-weight them using the Inverse Document Frequency (IDF), where $IDF(\text{token}) = \log(\frac{\#L}{\#L_{\text{token}}})$. $\#L$ is the total number of logs and $\#L_{\text{token}}$ is the number of logs containing the token. The TF-IDF weight of a token is calculated by $w = TF \times IDF$.

For each log line, we can obtain the vector representation \mathbf{V}_L by summing up every token vector with their corresponding TF-IDF weights within \mathbf{V}_L , according to Equation 1:

$$\mathbf{V}_L = \frac{1}{N} \sum_{i=1}^N w_i \mathbf{v}_i. \quad (1)$$

- **Dimensionality reduction.** The TF-IDF vectors of logs are high-dimensional and sparse, which slows clustering and increases memory use. We therefore explore two dimensionality-reduction techniques plus one vector-to-scalar summarization technique to speed up the following clustering step:
 - *Gaussian Random Projection* (GRP) [49]. Random projection is a dimensionality-reduction technique that maps high-dimensional points into a much smaller Euclidean space using a fixed, randomly generated matrix. We create the matrix from a zero-mean Gaussian distribution and multiply each log’s TF-IDF vector by it, collapsing the vector to two dimensions. Given the Johnson–Lindenstrauss lemma [27], pairwise distances are approximately preserved after GRP. Hence, the resulting low-dimensional vectors still reflect the original geometry and therefore serve as compact, clustering-friendly representations.

Table 5. Parsing efficiency of LibreLog using different grouping methods.

	Parsing time (seconds) per 10K logs	GA	PA	FGA	FTA
LibreLog	4.24	87.2	85.4	82.3	65.1
GRP+DBSCAN	20.24 (↑377.4 %)	50.9 (↓41.6 %)	64.0 (↓25.1 %)	47.4 (↓42.4 %)	49.1 (↓24.6 %)
GRP+HDBSCAN	14.04 (↑231.1 %)	74.7 (↓14.3 %)	81.1 (↓5.0 %)	71.3 (↓13.4 %)	61.7 (↓5.2 %)
PCA+DBSCAN	11.73 (↑176.7 %)	43.4 (↓50.2 %)	42.2 (↓50.6 %)	6.7 (↓91.9 %)	13.2 (↓79.7 %)
PCA+HDBSCAN	10.86 (↑156.1 %)	82.5 (↓5.4 %)	78.1 (↓8.5 %)	71.6 (↓13.0 %)	62.9 (↓3.4 %)
NORM+DBSCAN	47.53 (↑1021.0 %)	67.9 (↓22.1 %)	79.0 (↓7.5 %)	61.1 (↓25.8 %)	62.6 (↓3.8 %)
NORM+HDBSCAN	7.61 (↑79.5 %)	79.1 (↓9.3 %)	83.6 (↓2.1 %)	70.0 (↓14.9 %)	61.9 (↓4.9 %)

- *Principal Component Analysis* (PCA) [50]. PCA is a classic dimensionality-reduction method that rotates the data to a new orthogonal basis and keeps only the directions that explain the most variance. We form the covariance matrix of all log TF-IDF vectors, extract the top two principal components, and project each log onto this 2-D subspace. The retained components emphasize the dominant lexical patterns and suppress high-frequency noise, resulting in low-dimensional log vectors for clustering.
- The ℓ_2 norm (NORM) [61]. Different from GRP and PCA, NORM does not reduce dimensionality; instead, it collapses the entire log TF-IDF vector into a single scalar. Concretely, we first standardize every dimension of the TF-IDF matrix to zero mean and unit variance, and then compute $\|\mathbf{v}\|_2 = \sqrt{\sum_{i=1}^d v_i^2}$, where d is the TF-IDF vocabulary size and v_i is the standardized weight of the i -th token in the log. Although this loses some lexical detail, the NORM still carries useful structural information: logs that share the same template have almost identical token counts and weight distributions, so their norms show similar values. In contrast, logs of very different length or vocabulary demonstrate large norm differences. Thus, the single-scalar NORM offers the lightweight summarization of log vectors while preserving a coarse notion of template similarity.
- **Clustering.** There are many clustering algorithms that can be applied at this stage (e.g., k -means [43], hierarchical agglomerative [48], and density-based [46, 56]). Among them, we choose density-based clustering with three practical reasons: (i) it does not require the number of clusters to be specified in advance, making it suitable in unsupervised setting; (ii) it is widely used across domains and has demonstrated effective and efficient performance [57]; and (iii) it uses few hyperparameters and is comparatively robust to various settings. Within density-based methods, we adopt the most representative algorithm DBSCAN [56] and its hierarchical extension HDBSCAN [46].
 - DBSCAN [56] is the most commonly used density-based clustering method. It defines clusters as contiguous regions of high sample density separated by sparse regions. It identifies dense cores, connects nearby points that are density-reachable, and labels outliers as *noise*. We use DBSCAN to separate logs with different feature patterns into distinct clusters, and we aggregate all *noise points* (does not belong to any cluster) into a single cluster for downstream processing.
 - HDBSCAN [46] is a hierarchical variant of DBSCAN. HDBSCAN builds a clustering hierarchy across multiple density levels and extracts the most stable clusters. This mitigates sensitivity to a single global density level and better handles mixed-density data. As with DBSCAN, noise points are considered as a noise cluster for use in the subsequent parsing step.

Combining the three dimensionality reduction techniques with the two clustering algorithms yields six configurations: GRP+DBSCAN, GRP+HDBSCAN, PCA+DBSCAN, PCA+HDBSCAN, NORM+DBSCAN, and NORM+HDBSCAN. Each configuration replaces the original grouping tree’s output and is supplied to the downstream parsing stages, enabling a controlled comparison of their impact on accuracy and runtime.

Results. Table 5 reports the end-to-end parsing performance after replacing the fixed-depth tree grouping with six density-based grouping variants.

Density-based clustering consistently reduces accuracy while increasing runtime. The base LibreLog baseline attains GA 87.2%, PA 85.4%, FGA 82.3%, and FTA 65.1% at an parsing speed of 4.24s per 10K logs. All alternative groupers degrade every metric: GA drops by 5.4–50.2%, PA by 2.1–50.6%, and FGA/FTA by up to 42% and 80%, respectively. Even the best configuration, NORM+HDBSCAN, performs worse than the baseline by 2.1% PA and costs significantly more time (180% more) than the base LibreLog. We find that two factors drive the decrease in accuracy: (i) density-based algorithms can over-split a template when a single variable field contains several high-frequency values, causing logs with each value to form their own group and be assigned to different clusters, and (ii) they sometimes merge distinct but rare templates into the same cluster, so the downstream LLM-based template generation inherits both under- and over-grouping errors. Since there were more parsing errors, they trigger LLM self-reflection more frequently, resulting in higher time costs. Because every mismatch introduces extra self-reflection queries, the parsing time per 10K logs rises even though the clustering step itself is fast.

Compared to DBSCAN, HDBSCAN improves overall accuracy and efficiency, but both perform worse than the base LibreLog. Regardless of using GRP, PCA, or NORM, DBSCAN variants suffer the steepest accuracy losses. Switching from DBSCAN to HDBSCAN raises accuracy while shortening parsing time. With GRP, GA/PA/FGA jump from 50.9/64.0/47.4 to 74.7/81.1/71.3, and time drops from 20.24s to 14.04s. PCA shows the largest improvement, GA from 43.4 to 82.5, PA from 42.2 to 78.1, FGA from 6.7 to 71.6, and FTA from 13.2 to 62.9. Although with NORM the switch from DBSCAN to HDBSCAN slightly lowers the FTA (62.6 to 61.9), the other metrics improve significantly, GA rises from 67.9 to 79.1, PA from 79.0 to 83.6, FGA from 61.1 to 70.0, and parsing time drops dramatically from 47.53s to 7.61s. These improvements are because HDBSCAN, as a hierarchical extension of DBSCAN, adapts to mixed-density data and yields cleaner clusters, reducing the mixing of logs from different templates in a group. **Cleaner groups provide more coherent examples to the LLM-based unsupervised log parsing stage and trigger fewer self-reflections, improving both accuracy and efficiency.** Still, every density-based variant perform worse than the base LibreLog (e.g., best GA 82.5 vs 87.2, best PA 83.6 vs 85.4, best FGA 71.6 vs 82.3, and best FTA 62.9 vs 65.1) and are slower (best time 7.61 s vs 4.24 s), confirming the superior accuracy–efficiency balance of the base LibreLog.

GRP, PCA and NORM show different speed–accuracy trade-offs, yet all configurations perform worse than the base LibreLog. Using HDBSCAN as a common clustering backbone, PCA shows the strongest GA, FGA, and FTA, while NORM demonstrates the fastest speed and PA. Even so, the base LibreLog still performs best overall, in both effectiveness and efficiency. Concretely, PCA+HDBSCAN achieves the highest GA (82.5) and the best FTA 62.9, indicating that PCA’s preserving global variance strategy helps log cluster align with true template grouping. NORM+HDBSCAN is the fastest (7.61s) and reaches the best PA (83.6), but its FGA is lower (70.0). This reflects that the lightweight summarizing log vectors into single-scalars significantly improves the speed over other dimensionality reduction techniques and often preserves enough log information for clustering. However, due to the lack of some lexical details, NORM sometimes cannot separate similar logs apart, which leads to lower GA and FGA. GRP+HDBSCAN offers no standout advantage in all metrics compared with PCA and NORM. Its time is higher than NORM+HDBSCAN (14.04 vs 7.61 s) while its accuracy is worse than PCA+HDBSCAN (GA 74.7 vs 82.5; FTA 61.7 vs 62.9). Random projections retain approximate distances but can blur sparse TF–IDF vector geometry, yielding clusters that are neither the most accurate nor the fastest. In short, when density clustering is used, PCA is preferable for accuracy, NORM offers advantages in speed with moderate accuracy, and GRP offers no clear win over either.

Table 6. LibreLog performance compared to LibreLog with Llama3-70B *self-reflection* on mismatched templates.

	LibreLog					LibreLog w/ enhanced self-reflection				
	Total time (s)	GA	PA	FGA	FTA	Total time (s)	GA	PA	FGA	FTA
HDFS	1,252.62	100.0	100.0	92.9	79.5	2,370.38	100.0	100.0	95.7	92.3
Hadoop	285.76	96.3	87.1	94.0	75.5	568.64	97.8	94.4	94.9	81.3
Spark	1,752.40	85.9	88.9	83.2	69.0	4,427.22	88.6	90.8	88.6	70.2
Zookeeper	52.23	99.3	85.0	97.4	86.3	83.60	99.3	84.9	83.1	87.6
BGL	1,244.64	90.2	92.9	80.3	71.6	1,533.29	93.0	96.0	74.1	74.1
HPC	539.76	84.4	97.3	80.5	60.9	538.47	79.8	92.9	87.8	62.2
Thunderbird	8,659.29	87.0	69.4	84.3	54.0	7,945.82	82.5	72.9	86.1	55.7
Linux	216.03	91.2	90.2	74.1	60.7	398.50	89.7	77.8	74.3	58.9
HealthApp	103.33	86.2	97.4	95.8	87.7	2,569.91	86.8	97.8	87.8	87.0
Apache	18.92	100.0	99.6	100.0	75.9	32.12	100.0	99.2	100.0	72.4
Proxifier	871.52	51.0	89.7	35.3	40.0	152.06	51.0	97.0	54.5	54.5
OpenSSH	89.37	86.8	49.6	85.3	47.2	129.19	81.7	57.2	78.9	57.9
OpenStack	377.64	81.1	83.1	68.9	55.9	2,828.00	82.1	80.1	66.7	49.5
Mac	5,935.77	81.4	65.4	80.4	46.9	4,539.34	89.2	67.8	87.4	51.8
Average	1,528.52	87.2	85.4	82.3	65.1	2,008.32	87.3	86.3	82.8	68.3
Total time	5.94 hours					7.81 hours				

Replacing the fixed-depth grouping tree with density-based clustering methods lead to performance reduction, confirming that the fix-depth tree grouping remains the most effective and efficient choice for LibreLog.

RQ6: Does scaling up the LLM enhance *self-reflection* capabilities?

Motivation. Although LibreLog with Llama3-8B has achieved strong performance, some template mismatches cannot be fixed even after the self-reflection stage. Close inspection shows that these mismatches are due to subtle structural gaps (e.g., missing delimiters or reordered tokens). We hypothesize that this arises because smaller models inherently lack sufficient capability to fully comprehend certain complex log structures. To address this, in this RQ, we investigate whether employing a larger LLM for self-reflect can fix these mismatches and further boost the overall parsing quality.

Approach. Whenever a generated template fails to parse a given log line, LibreLog re-prompts the LLM to reflect and regenerate that template. We update the LLM that LibreLog uses for self-reflection, while leaving the other phases unchanged. Specifically, we use Llama3-70B for self-reflection. Llama3-70 is from the same model family as LibreLog’s base LLM Llama3-8B, but with more parameters. The larger capacity provides a stronger understanding of the log structure and semantics during self-reflection while maintaining consistent input and output formats. Additionally, while more parameters can slow down the model, Llama3-70B strikes a good balance between efficiency and performance [4]. To increase the comprehensiveness of the experiment, we added three additional open-source large LLMs for enhanced self-reflection: gpt-oss-20B [6], Qwen3-30B [67] and Qwen3-Coder-30B [67]. We choose these LLMs because they have more parameters than Llama3-8B and have been widely used across code and natural-language tasks, where they demonstrate strong performance.

Results. *Employing Llama-3-70B for self-reflection yields a 0.1% to 5% improvement in four accuracy metrics, with the largest gains appearing at the template level, indicating that larger LLMs can correct malformed templates more precisely.* As shown in Table 6, overall, GA and FGA are mostly stable (change from 87.2 to 87.3 for GA and 82.3 to 82.8 for FGA). The reason is that both GA and GPA are related to grouping accuracy, and LibreLog’s grouping remains unchanged (still relies on the fixed-depth tree). In contrast, we see

Table 7. Parsing performance of LibreLog w/ enhanced self-reflection using different LLMs on LogHub-2.0.

	Total time (h)	GA	PA	FGA	FTA
LibreLog	5.94	87.2	85.4	82.3	65.1
w/ Llama3-70B	7.81 (↑31.5%)	87.3 (↑0.1%)	86.3 (↑1.1%)	82.8 (↑0.6%)	68.3 (↑4.9%)
w/ gpt-oss-20B	6.71 (↑13.0%)	86.5 (↓0.8%)	85.2 (↓0.2%)	82.7 (↑0.5%)	65.9 (↑1.2%)
w/ Qwen3-30B	7.76 (↑30.6%)	87.8 (↑0.7%)	85.9 (↑0.6%)	82.5 (↑0.2%)	65.2 (↑0.2%)
w/ Qwen3-Coder-30B	7.31 (↑23.1%)	87.4 (↑0.2%)	85.8 (↑0.5%)	82.0 (↓0.4%)	66.6 (↑2.3%)

a larger improvement in PA (from 85.4 to 86.3) and FTA (from 65.1 to 68.3). The findings indicate that using a larger LLM for self-reflect can result in fewer token-level extraction errors and better parsing accuracy.

Systems with fewer templates show more improvement from using a larger LLM for self-reflection. The largest PA/FTA improvements occur on systems with fewer log templates: Proxifier (11 templates; +7.3% PA, +14.5% FTA); HDFS (46 templates; +12.8% FTA); and OpenSSH (38 templates; +7.6% PA, +10.7% FTA). When there are fewer templates, fixing a single mismatched template affects a larger share of logs. Hence, self-reflection with a larger model yields larger accuracy gains. Accuracy increases are more modest, but still present, in high-template count systems such as Hadoop (236 templates; +7.3% PA, +5.8% FTA), Thunderbird (1,241 templates; +3.5% PA, +1.7% FTA), and Mac (626 templates; +2.4 pp PA, +4.9% FTA), where improvements are distributed across many templates. Although the grouping part is unchanged, self-reflection rewrites generated templates and can merge or split template groups, so GA for some systems may fluctuate slightly (e.g., Thunderbird and OpenSSH). On average, GA and FGA both increase slightly, which indicates that overall grouping quality is improved.

Integrating a Llama3-70B self-reflection introduces a moderate overhead relative to the performance gain. Adapting a 70B LLM increases the end-to-end processing time from 5.94 h to 7.81h (approximately 31%). The additional time cost is not uniform: systems such as HDFS and Spark finish sooner because the larger model resolves most templates in a single self-reflection round, whereas Mac and Proxifier experience a slowdown because they contain many more mismatched templates. The average parsing time per system is increased from 1.5K seconds (25 minutes) to 2k seconds (33 minutes). In other words, for most systems, using a larger model for self-reflection will take longer. Although integrating a Llama3-70B self-reflection will introduce more time consumption, it still shows a better parsing speed than other state-of-the-art LLM-based parsers. Thus, the overall processing time remains within the practical budget. Future studies and practitioners can consider this trade-off when applying larger LLM for parsing.

Enhanced self-reflection with larger LLMs yields modest accuracy improvement compared to using the 8B model, but consistently increases the running time. As shown in Table 7, among the three evaluated LLMs, gpt-oss-20B shows the fastest parsing speed but slightly lower GA and PA, while improving FGA and FTA compared to using the 7B model. Qwen3-30B offers a uniform improvement across the four metrics at a higher time cost (30.6%). Qwen3-Coder-30B provides gains in GA, PA, and FTA with 23.1% increase in execution time, but a slight decrease on FGA compared to using the 8B model. Overall, using different larger LLMs for enhanced self-reflection shows distinct accuracy–cost profiles and execution time. Smaller LLMs generally deliver faster parsing with limited accuracy gains. Future deployments of LLM-based log parsers should choose the LLM by balancing time cost constraints against the desired accuracy improvements.

Using a larger LLM for self-reflecting improve the overall parsing accuracy. Enhanced self-reflection speed up the parsing time of systems with less log templates, but overall time is increased from 5.94 hours to 7.81 hours.

6 DISCUSSION

6.1 Remaining parsing challenges after using the 70B LLM for self-reflection

Even with the stronger 70B model for self-reflection, a small fraction of log lines still cannot be correctly parsed. To understand the reasons behind the remaining errors and to chart the next research steps, we analyzed where our unsupervised parser still struggled. The findings can guide future work toward smarter grouping methods and richer, better-annotated log-parsing evaluation datasets that avoid the gaps uncovered here. A closer inspection exposes three recurring causes.

Across the 14 systems (about 50 million logs), our parser leaves about 6.58 million log lines incorrectly parsed. To make the analysis comprehensive yet feasible, we follow prior studies [20, 37] and use random sampling to select 384 incorrectly parsed logs for manual inspection, which achieves a 95% confidence level with a 5% confidence interval. From the 384 cases, we identify three main reasons that affect accuracy: boundary disagreements on variables (26.8%), ground-truth granularity mismatches (11.5%), and over-generalization after coarse grouping (61.7%).

Boundary disagreement on variables (26.8%). Some generated log templates have different variable boundaries from the ground truth annotation. Tokens such as variables with underscores can be treated either as single variables or as multiple variables. Minor format drift across components or versions (e.g., underscores, delimiters, or token reorderings) further blurs the boundaries between static and dynamic variables.

Log: Exception in receiveBlock for block `blk_4241467193520768` java.io.EOFException

Parsed template: Exception in receiveBlock for block `blk_<*>` java.io.EOFException

Ground truth template: Exception in receiveBlock for `block <*>` java.io.EOFException

For the instance above, variable ‘`blk_4241467193520768`’ is parsed to ‘`blk_<*>`’, but the ground truth is ‘`<*>`’. Because PA and FTA require identical static–dynamic boundaries between generated templates and ground truth templates, PA/FTA decrease due to any minor variable boundary mismatch.

Ground-truth granularity mismatches (11.5%). Public log datasets are annotated at varying levels of granularity. Occasionally, the ground truth template labels an optional or empty field as a variable; the generated template will be considered as an error if it fails to correctly include the variable, leading to false negatives.

Log: Connection broken for id 2, my id = 3, `error =`

Parsed template: Connection broken for id `<*>`, my id = `<*>`, `error =`

Ground truth template: Connection broken for id `<*>`, my id = `<*>`, `error =<*>`

For example, the log above contains no variable after ‘`error =`’. The LLM therefore generates a template without a variable after ‘`error =`’, whereas the ground truth template has one variable placeholder ‘`error =<*>`’. Here, the model-generated template is semantically correct: the last variable field is simply empty, but evaluation marks it incorrect because it differs from the ground-truth template. Such cases indicate that a portion of the residual errors originates from coarse or inconsistent ground truth annotation rather than log parser behavior.

Over-generalization after coarse grouping (61.7%). The differences between the logs of some different templates are small, making it hard to tell whether the variation is due to variable values or distinct templates. Logs that share length and prefix tokens but differ in the suffix may be grouped together. Their mixed examples then enter the same retrieval prompt for template generation. Although they have different ground truth templates, when sent to the LLM together, the LLM generalizes their differences as a variable and treats them as the same template.

Template 1: PacketResponder `<*>` `<*>` Exception java.io.IOException: `The stream is closed`

Template 2: PacketResponder `<*>` `<*>` Exception java.io.IOException: `Connection reset by peer`

Parsed template: PacketResponder `<*>` `<*>` Exception java.io.IOException: `<*>`

Given logs from two different templates above, the model abstracts the different parts and generates a single template for these logs. Grouping logs with different templates together inevitably results in over-generalized log templates, thus affecting accuracy.

The residual mismatches highlighted above largely reflect limitations that are difficult to eliminate in fully automatic parsers. Addressing the above causes of failure will likely require progress along two dimensions. First, more detailed evaluation datasets: public datasets would benefit from clearer, machine-checkable annotation guidelines on (i) how to label optional or empty fields, and (ii) consistent variable annotation boundaries. Providing additional semantically equivalent templates would also help log parser evaluation. Second, a promising direction is to develop smarter grouping methods that remain efficient, e.g., suffix-aware or semantics-aware clustering, to better separate similar logs with different templates, which can better assist unsupervised LLM-based parsers.

6.2 Threats to validity

External validity. Data leakage is a potential risk of LLM-based log parsers [26, 44]. Although LibreLog does not involve using labeled logs for fine-tuning or in-context learning, there is a possibility that the LLM might have been pre-trained on publicly available log data. Our evaluation dataset with ground-truth templates was released on August 2023 [25] and Llama3-8B training knowledge cutoff from March 2023 [4], so the leakage risk should be minimal. The log format may also affect our result, but the datasets used are large and cover logs from various systems in different formats. Future studies are needed to evaluate LibreLog on logs from other systems.

Internal validity. For the ChatGPT-based baselines, LUNAR [21] and LILAC [26], we replaced their original LLM backbone with the same small open-source LLM used by LibreLog (Llama-3-8B [4]) for fair comparison. This substitution may reduce their performance because these methods were originally designed and optimized for the ChatGPT backbone. However, we post-processed the model output phase of LUNAR and LILAC to ensure that the output results would not be affected by different model output formats. LibreLog employs Llama3-8B as its base model due to its promising results in many tasks and the relatively small size [4]. We also compared the results across various open-source LLMs and found differences. Future research is needed to evaluate LLM-based parsers’ performance when more advanced LLMs are released in the future. The effectiveness of LibreLog could be influenced by specific parameter settings (e.g., the number of logs selected for prompting). Our evaluations showed that these settings have an impact on the parsing results and discussed the optimal settings. Future studies are needed to evaluate the settings on other datasets.

Construct validity. To mitigate the effects of randomness in evaluating LibreLog, the generation temperature of the model is set to zero. This adjustment ensures that experiments conducted under the same conditions are repeatable and that the results are stable.

7 CONCLUSION

In this paper, we introduced LibreLog, an unsupervised log parsing technique utilizing open-source LLMs to effectively address the limitations of existing LLM-based and syntax-based parsers. LibreLog first groups logs that share a syntactic similarity in the static text but vary in the dynamic variable, using a fixed-depth grouping tree. It then parses logs in these groups with three components: i) retrieval augmented generation using similarity scoring: identifies diverse logs within each group based on Jaccard similarity, aiding the LLM in differentiating static text from dynamic variables; ii) self-reflection: iteratively queries LLMs to refine log templates and enhance parsing accuracy; and iii) log template memory: store parsed templates to minimize LLM queries, thereby boosting parsing efficiency. Our comprehensive evaluations on LogHub-2.0, a public large-scale log dataset, demonstrate that LibreLog achieves an average GA of 87.2, PA of 85.4, FGA of 82.3, and FTA of 65.1, outperforming state-of-the-art LLM-based parsers (i.e., LUNAR [21], LILAC [26] and LLMParse [44]). LibreLog parses logs from all 14 systems (50 million logs) in a total of 5.94 hours, which is 1.79 and 40 times faster than other LLM-based parsers. We also

show that invoking a larger LLM only in the self-reflection loop for mismatched templates further lifts PA (from 85.4 to 86.3) and FTA (from 85.1 to 68.3), while adding a moderate 31% runtime overhead. This marks a substantial advancement over traditional semantic-based and LLM-based parsers in an unsupervised way, confirming the robustness and effectiveness of our approach. Additionally, LibreLog addresses the privacy and cost concerns associated with commercial LLMs, making it a highly efficient and secure solution for practical log parsing needs.

REFERENCES

- [1] Samsung bans chatgpt among employees after sensitive code leak. <https://www.forbes.com/sites/siladityaray/2023/05/02/samsung-bans-chatgpt-and-other-chatbots-for-employees-after-sensitive-code-leak/>, 2023. (Accessed on 07/18/2024).
- [2] Security and privacy: Closed source vs open source battle. <https://medium.com/blue-orange-digital/security-and-privacy-closed-source-vs-open-source-battle-a8757487040e>, 05 2024. (Accessed on 05/17/2024).
- [3] Wall street banks are cracking down on ai-powered chatgpt - bloomberg. <https://www.bloomberg.com/news/articles/2023-02-24/citigroup-goldman-sachs-join-chatgpt-crackdown-fn-reports>, 2024. (Accessed on 07/18/2024).
- [4] Introducing meta llama 3: The most capable openly available llm to date. <https://ai.meta.com/blog/meta-llama-3/>, 2024. (Accessed on 07/22/2024).
- [5] Samuel Abedu, Ahmad Abdellatif, and Emad Shihab. Llm-based chatbots for mining software repositories: Challenges and opportunities. 2024.
- [6] Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.
- [7] C Aicardi, L Bitsch, and S Datta Burton. Trust and transparency in artificial intelligence. ethics & society opinion. european commission. 2020.
- [8] Keivan Alizadeh, Seyed Iman Mirzadeh, Dmitry Belenko, S Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. Llm in a flash: Efficient large language model inference with limited memory. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12562–12584, 2024.
- [9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.
- [10] Jinfu Chen, Weiyi Shang, Ahmed E Hassan, Yong Wang, and Jiangbin Lin. An experience report of generating load tests using log-recovered workloads at varying granularities of user behaviour. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 669–681. IEEE, 2019.
- [11] Xiaolei Chen, Jie Shi, Jia Chen, Peng Wang, and Wei Wang. High-precision online log parsing with large language models. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 354–355, 2024.
- [12] Krishna Teja Chitty-Venkata, Siddhisanket Raskar, Bharat Kale, Farah Ferdaus, Aditya Tanikanti, Ken Raffanetti, Valerie Taylor, Murali Emani, and Venkatram Vishwanath. Llm-inference-bench: Inference benchmarking of large language models on ai accelerators. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1362–1379. IEEE, 2024.
- [13] Hyung Won Chung et al. Scaling instruction-finetuned language models, 2022.
- [14] Hetong Dai, Heng Li, Che-Shao Chen, Weiyi Shang, and Tse-Hsun Chen. Logram: Efficient log parsing using n n-gram dictionaries. *IEEE Transactions on Software Engineering*, 48(3):879–892, 2020.
- [15] Min Du and Feifei Li. Spell: Online streaming parsing of large unstructured system logs. *IEEE Transactions on Knowledge and Data Engineering*, 31(11):2213–2227, 2019. doi: 10.1109/TKDE.2018.2875442.
- [16] Team GLM et al. Chatglm: A family of large language models from glm-130b to glm-4 all tools, 2024.
- [17] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.
- [18] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40, 2017. doi: 10.1109/ICWS.2017.13.
- [19] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. Loghub: a large collection of system log datasets towards automated log analytics. *arXiv preprint arXiv:2008.06448*, 2020.
- [20] Yi Wen Heng, Zeyang Ma, Haoxiang Zhang, Zhenhao Li, et al. Discovery of timeline and crowd reaction of software vulnerability disclosures. *arXiv preprint arXiv:2411.07480*, 2024.
- [21] Junjie Huang, Zhihan Jiang, Zhuangbin Chen, and Michael Lyu. No more labelled examples? an unsupervised log parser with llms. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025. doi: 10.1145/3729377. URL <https://doi.org/10.1145/3729377>.

- [22] Yizhan Huang, Yichen Li, Weibin Wu, Jianping Zhang, and Michael R. Lyu. Your code secret belongs to me: Neural code completion tools can memorize hard-coded credentials. *Proc. ACM Softw. Eng.*, 1(FSE), jul 2024. doi: 10.1145/3660818.
- [23] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, L  lio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timoth  e Lacroix, and William El Sayed. Mistral 7b, 2023.
- [24] Zhen Ming Jiang, Ahmed E. Hassan, Parminder Flora, and Gilbert Hamann. Abstracting execution logs to execution events for enterprise applications (short paper). In *2008 The Eighth International Conference on Quality Software*, pages 181–186, 2008.
- [25] Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jiazhen Gu, Zhuangbin Chen, Jieming Zhu, and Michael R Lyu. A large-scale benchmark for log parsing. *arXiv preprint arXiv:2308.10828*, 2023.
- [26] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R. Lyu. Lilac: Log parsing using llms with adaptive parsing cache. 1(FSE), jul 2024.
- [27] William B Johnson, Joram Lindenstrauss, et al. Extensions of lipschitz mappings into a hilbert space. *Contemporary mathematics*, 26 (189-206):1, 1984.
- [28] Hao Kang, Qingru Zhang, Han Cai, Weiyuan Xu, Tushar Krishna, Yilun Du, and Tsachy Weissman. Win fast or lose slow: Balancing speed and accuracy in latency-sensitive decisions of llms. *arXiv preprint arXiv:2505.19481*, 2025.
- [29] Zanis Ali Khan, Donghwan Shin, Domenico Bianculli, and Lionel Briand. Guidelines for assessing the accuracy of log message template identification techniques. In *Proceedings of the 44th International Conference on Software Engineering, ICSE ’22*, 2022.
- [30] Zanis Ali Khan, Donghwan Shin, Domenico Bianculli, and Lionel Briand. Impact of log parsing on log-based anomaly detection. *arXiv:2305.15897*, 2023.
- [31] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W Mahoney, and Kurt Keutzer. Squeezellm: Dense-and-sparse quantization. *arXiv preprint arXiv:2306.07629*, 2023.
- [32] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626, 2023.
- [33] Van-Hoang Le and Hongyu Zhang. Log parsing: How far can chatgpt go? In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1699–1704, 2023.
- [34] Van-Hoang Le and Hongyu Zhang. Log parsing with prompt-based few-shot learning. In *45th International Conference on Software Engineering: Software Engineering in Practice (ICSE)*, 2023.
- [35] Van-Hoang Le, Yi Xiao, and Hongyu Zhang. 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)*, pages 975–987, Los Alamitos, CA, USA, May 2025. IEEE Computer Society. doi: 10.1109/ICSE55347.2025.00174. URL <https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00174>.
- [36] Yookyung Lee, Jina Kim, and Pilsung Kang. Lanobert: System log anomaly detection based on bert masked language model. *arXiv preprint arXiv:2111.09564*, 2021.
- [37] Zhenhao Li, Chuan Luo, Tse-Hsun (Peter) Chen, Weiye Shang, Shilin He, Qingwei Lin, and Dongmei Zhang. Did we miss something important? studying and exploring variable-aware log abstraction. In *Proceedings of the 45th International Conference on Software Engineering, ICSE ’23*, page 830–842, 2023.
- [38] Zongqian Li, Yinhong Liu, Yixuan Su, and Nigel Collier. Prompt compression for large language models: A survey. *arXiv preprint arXiv:2410.12388*, 2024.
- [39] Feng Lin, Dong Jae Kim, et al. When llm-based code generation meets the software development process. *arXiv preprint arXiv:2403.15852*, 2024.
- [40] Jinyang Liu, Junjie Huang, Yintong Huo, Zhihan Jiang, Jiazhen Gu, Zhuangbin Chen, Cong Feng, Minzhi Yan, and Michael R Lyu. Scalable and adaptive log-based anomaly detection with expert in the loop. *arXiv preprint arXiv:2306.05032*, 2023.
- [41] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, 2019.
- [42] Yudong Liu, Xu Zhang, Shilin He, Hongyu Zhang, Liqun Li, Yu Kang, Yong Xu, Minghua Ma, Qingwei Lin, Yingnong Dang, et al. Uniparser: A unified log parser for heterogeneous log data. In *Proceedings of the ACM Web Conference 2022*, pages 1893–1901, 2022.
- [43] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.
- [44] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. Llmpraser: An exploratory study on using large language models for log parsing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE ’24*, 2024. ISBN 9798400702174. doi: 10.1145/3597503.3639150.
- [45] Zeyang Ma, Dong Jae Kim, and Tse-Hsun Peter Chen. LibreLog: Accurate and Efficient Unsupervised Log Parsing Using Open-Source Large Language Models . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 924–936, Los Alamitos, CA, USA, May 2025. IEEE Computer Society. doi: 10.1109/ICSE55347.2025.00103. URL <https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00103>.

- [46] Claudia Malzer and Marcus Baum. A hybrid approach to hierarchical density-based cluster selection. In *2020 IEEE international conference on multisensor fusion and integration for intelligent systems (MFI)*, pages 223–228. IEEE, 2020.
- [47] Lingbo Mo, Boshi Wang, Muhao Chen, and Huan Sun. How trustworthy are open-source LLMs? an assessment under malicious demonstrations shows their vulnerabilities. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 2775–2792, June 2024.
- [48] Daniel Müllner. Modern hierarchical, agglomerative clustering algorithms. *arXiv preprint arXiv:1109.2378*, 2011.
- [49] Mahmoud Nabil. Random projection and its applications. *arXiv preprint arXiv:1710.03163*, 2017.
- [50] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2(11):559–572, 1901.
- [51] Devjeet Roy, Xuchao Zhang, Rashi Bhav, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. Exploring llm-based agents for root cause analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 208–219, 2024.
- [52] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.
- [53] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.
- [54] Komal Sarda. Leveraging large language models for auto-remediation in microservices architecture. In *2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, pages 16–18. IEEE, 2023.
- [55] Komal Sarda, Zakeya Namrud, Raphael Rouf, Harit Ahuja, Mohammadreza Rasolroveicy, Marin Litoiu, Larisa Shwartz, and Ian Watts. Adarma auto-detection and auto-remediation of microservice anomalies by leveraging large language models. In *Proceedings of the 33rd Annual International Conference on Computer Science and Software Engineering*, pages 200–205, 2023.
- [56] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. Dbscan revisited, revisited: why and how you should (still) use dbscan. *ACM Transactions on Database Systems (TODS)*, 42(3):1–21, 2017.
- [57] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. Dbscan revisited, revisited: why and how you should (still) use dbscan. *ACM Transactions on Database Systems (TODS)*, 42(3):1–21, 2017.
- [58] Donghwan Shin, Zanis Ali Khan, Domenico Bianculli, and Lionel Briand. A theoretical framework for understanding the relationship between log parsing and anomaly detection. In *Runtime Verification: 21st International Conference, RV 2021, Virtual Event, October 11–14, 2021, Proceedings*, page 277–287, 2021.
- [59] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.
- [60] Amit Singhal et al. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.
- [61] Gilbert Strang. *Introduction to linear algebra*. SIAM, 2022.
- [62] Jing Su, Chufeng Jiang, Xin Jin, Yuxin Qiao, Tingsong Xiao, Hongda Ma, Rong Wei, Zhi Jing, Jiajun Xu, and Junhong Lin. Large language models for forecasting and anomaly detection: A systematic literature review. *arXiv preprint arXiv:2402.10350*, 2024.
- [63] P.N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. 2016. ISBN 9789332586055.
- [64] CodeGemma Team et al. Codegemma: Open code models based on gemma, 2024.
- [65] Zehao Wang, Haoxiang Zhang, Tse-Hsun Chen, and Shaowei Wang. Would you like a quick peek? providing logging support to monitor data processing in big data applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 516–526, 2021.
- [66] Junjielong Xu, Ruichun Yang, Yintong Huo, Chengyu Zhang, and Pinjia He. Divlog: Log parsing with prompt enhanced in-context learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE ’24*, 2024. ISBN 9798400702174.
- [67] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.
- [68] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Eric Sun, and Yue Zhang. A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *ArXiv*, abs/2312.02003, 2023.
- [69] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th International Conference on Architectural support for programming languages and operating systems*, pages 143–154, 2010.
- [70] Tianzhu Zhang, Han Qiu, Gabriele Castellano, Myriana Rifai, Chung Shue Chen, and Fabio Pianese. System Log Parsing: A Survey. *IEEE Transactions on Knowledge & Data Engineering*, 35(08):8596–8614, August 2023. ISSN 1558-2191. doi: 10.1109/TKDE.2022.3222417. URL <https://doi.ieeecomputersociety.org/10.1109/TKDE.2022.3222417>.
- [71] Wei Zhang, Xiangyuan Guan, Lu Yunhong, Jie Zhang, Shuangyong Song, Xianfu Cheng, Zhenhe Wu, and Zhoujun Li. Lemur: Log parsing with entropy sampling and chain-of-thought merging. *arXiv preprint arXiv:2402.18205*, 2024.
- [72] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. 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*, pages 807–817, 2019.
- [73] Qingfei Zhao, Ruobing Wang, Yukuo Cen, Daren Zha, Shicheng Tan, Yuxiao Dong, and Jie Tang. Longrag: A dual-perspective retrieval-augmented generation paradigm for long-context question answering. *arXiv preprint arXiv:2410.18050*, 2024.
 - [74] Aoxiao Zhong, Dengyao Mo, Guiyang Liu, Jinbu Liu, Qingda Lu, Qi Zhou, Jiesheng Wu, Quanzheng Li, and Qingsong Wen. Logparser-llm: Advancing efficient log parsing with large language models. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 4559–4570, 2024.
 - [75] Zhanke Zhou, Rong Tao, Jianing Zhu, Yiwen Luo, Zengmao Wang, and Bo Han. Can large language models reason robustly with noisy rationales? In *ICLR 2024 Workshop on Reliable and Responsible Foundation Models*, 2024.
 - [76] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice*, pages 121–130, 2019.