LibreLog: Accurate and Efficient Unsupervised Log Parsing Using Open-Source Large Language Models

Zeyang Ma
Software PErformance, Analysis and Reliability (SPEAR) Lab
Concordia University
Montreal, Quebec, Canada
m_zeyang@encs.concordia.ca

Dong Jae Kim
DePaul University
Chicago, Illinois, USA
djaekim086@gmail.com

Tse-Hsun (Peter) Chen
Software PErformance, Analysis and Reliability (SPEAR) Lab
Concordia University
Montreal, Quebec, Canada
peterc@encs.concordia.ca

Abstract*—Log parsing is a critical step that transforms unstructured log data into structured formats, facilitating subsequent log-based analysis. Traditional syntax-based log parsers are efficient and effective, but they often experience decreased accuracy when processing logs that deviate from the predefined rules. Recently, large language models (LLM) based log parsers have shown superior parsing accuracy. However, existing LLM-based parsers face three main challenges: 1) time-consuming and labor-intensive manual labeling for fine-tuning or in-context learning, 2) increased parsing costs due to the vast volume of log data and limited context size of LLMs, and 3) privacy risks from using commercial models like ChatGPT with sensitive log information. To overcome these limitations, this paper introduces LibreLog, an unsupervised log parsing approach that leverages open-source LLMs (i.e., Llama3-8B) to enhance privacy and reduce operational costs while achieving state-of-the-art parsing accuracy. LibreLog first groups logs with similar static text but varying dynamic variables using a fixed-depth grouping tree. It then parses logs within these groups using three components: i) similarity scoring-based retrieval augmented generation: selects diverse logs within each group based on Jaccard similarity, helping the LLM distinguish between static text and dynamic variables; ii) self-reflection: iteratively query LLMs to refine log templates to improve parsing accuracy; and iii) log template memory: stores parsed templates to reduce LLM queries for improved parsing efficiency. Our evaluation on LogHub-2.0 shows that LibreLog achieves 25% higher parsing accuracy and processes logs 2.7 times faster compared to state-of-the-art LLM-based parsers. In short, LibreLog addresses privacy and cost concerns of using commercial LLMs while achieving state-of-the-arts parsing efficiency and accuracy.

I. Introduction

Real-world software systems generate large amounts of logs, often hundreds of gigabytes or even terabytes per day [11, 19, 48]. 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 [8, 43], anomaly detection [25, 40], and root cause analysis [33, 46]. 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 [22, 26, 37]. 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 [14] and AEL [18], 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 [19, 21, 48].

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

Log Parsing

Parsed templates

Date	Time	Level	Process	Component	Log Template
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.

Recent advances in large language models (LLMs) have enabled researchers to leverage these models for log parsing [9, 20, 23, 24, 31, 44]. 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 [20, 24, 31]. 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 commer-

cial LLMs like ChatGPT by many LLM-based log parsers [20, 23, 44]. 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 [14], 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*** [38] 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 [19], LibreLog demonstrated a 25% or higher parsing accuracy compared to state-of-the-art LLM-based log parsers (i.e., LILAC [20] and LLM-Parser [31]). Moreover, it is 2.75 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.
- Our experiment using four small-size LLMs shows that Llama3-8B achieves the best overall result, highlighting its potential in log analysis.

In short, the paper provides a novel unsupervised log parsing approach that is both efficient and effective while ensuring data privacy and reducing operational costs.

Paper Organization. Section II discusses background and related work. Section III provides the design details of LibreLog. Section IV outlines evaluation setup. Section V presents evaluation results. Section VI discusses threats to validity. Section VII concludes the paper.

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

II. Background and Related Work

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

A. Background

Large Language Models. Large Language Models (LLMs), primarily built on the transformer architecture [4, 7, 34], 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 [7], 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, 27]. 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 [25, 28, 40], root cause analysis [33, 35, 36], and log parsing [9, 20, 23, 24, 31, 44]. 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 [22, 37].

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, 7]. 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 [6]. More importantly, sensitive data uploaded to the LLM providers could potentially be exposed through inadvertent data leaks or malicious attacks [16], 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, 34], 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[32, 45].

B. 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 [11, 12, 14, 18] typically

employ heuristic rules or conduct comparisons among logs to identify common components that serve as templates. Semantic-based log parsers [20, 24, 30, 31] 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 [11, 12, 14, 18] 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 [18] uses heuristics to remove potential dynamic variables and extract log templates. Drain [14] 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 [11, 12] to parse the logs by identifying common textual content within logs. For instance, Spell [12] uses the Longest Common Subsequence to parse logs, and Logram [11] 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 [9, 20, 23, 24, 31, 44] 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 [23] can analyze the context of log messages and dynamically generate log templates without prior knowledge, enhancing accuracy and adaptability across different log formats. DivLog [44] enhances log parsing by extracting similar logs from a candidate set of labeled logs for in-context learning using GPT-3 [7]. Due to the high cost of commercial LLMs such as ChatGPT, LILAC [20] 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.

Some parsers also aim to use open-source LLMs for log parsing. Hooglle [9] adopted an LLM pre-trained on labeled logs for log parsing. LogPPT [24] utilizes a masked language model (RoBERTa [29]) 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. LLMParser [31] 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.

Although the results are promising, recent works in semantic-based log parsers have two main limitations: 1) privacy and monetary costs of using commercial LLMs and 2) requiring manually derived log templates for LLMs to learn. 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 [20]) or the ones that use smaller open-source models (e.g., LogPPT [24] and LLMParser [31]) 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.

In this paper, we propose LibreLog that addresses the two 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. 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.

III. 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-size open-source LLM to enhance privacy and reduce operation 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.

A. 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,

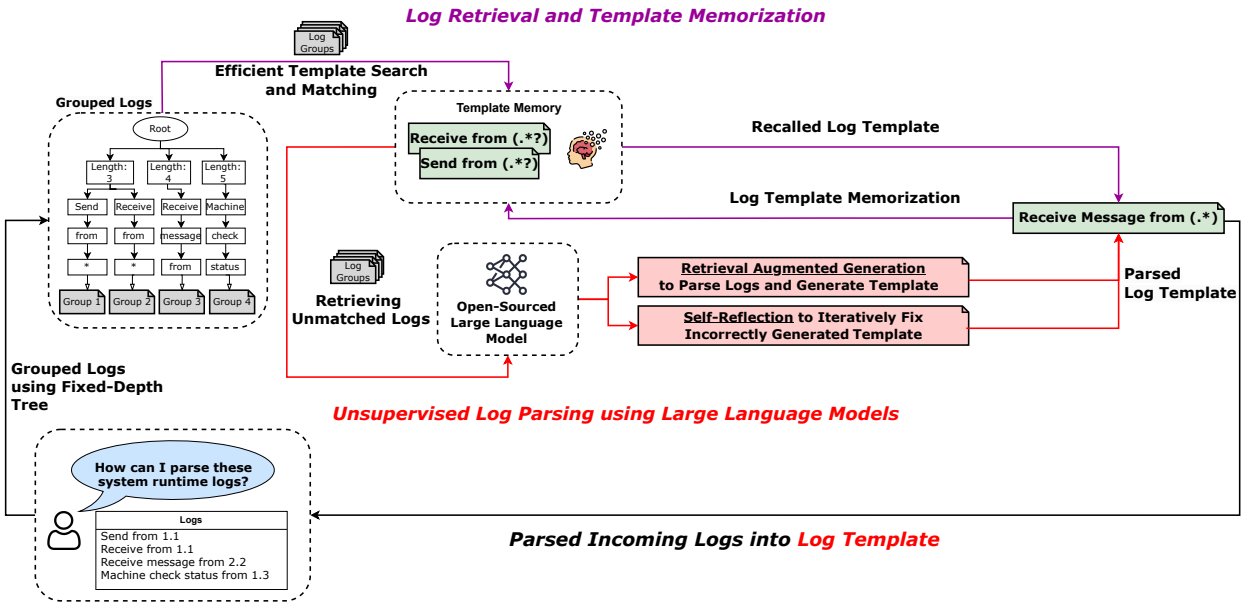

Fig. 2. An overview of LibreLog.

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 [14], 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 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. Prior to 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.

B. LLM-based Unsupervised Log Parsing

Our prompts to LLMs contain representative logs (based on variability) retrieved from each log group (from Section III-A) 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 [31], 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

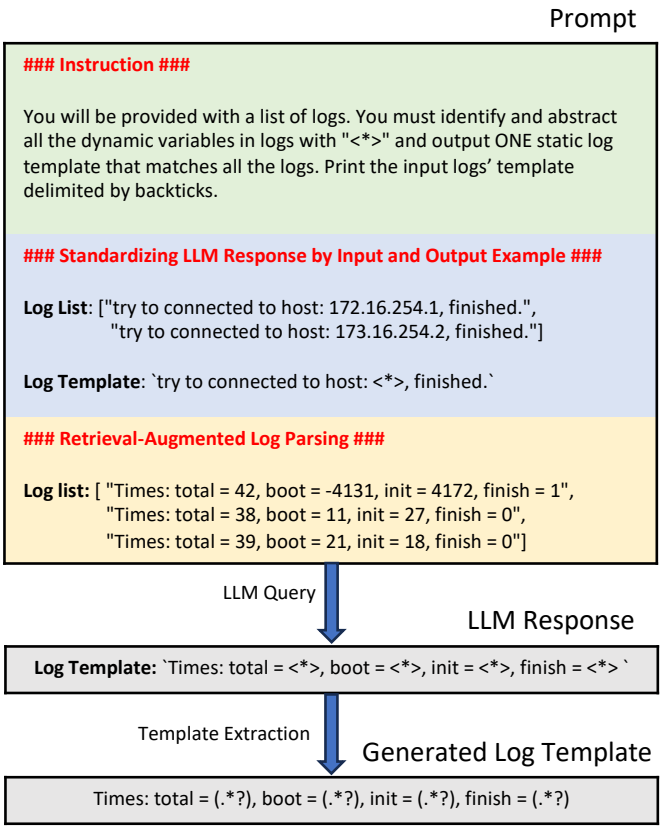

Fig. 3. Example of the prompt template used for LibreLog. The green block illustrates the task instruction provided to the LLM. The blue block highlights the input and output examples used to standardize the log response format. The yellow block depicts the retrieval-augmented selection process that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

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 augmented generation (RAG) approach based on Jaccard Similarity [41]. 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 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 [38], 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.

C. 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 $O(LogN)$, 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.

IV. 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. [15, 19]. 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 [11, 12, 14, 20, 24, 31]. Compared to LogHub-1.0 [15], 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 [19, 20].

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 [11, 14, 20, 21, 24, 31], we use two most commonly used metrics to evaluate the effectiveness of log parsers: Group Accuracy and Parsing Accuracy.

Group Accuracy (GA): Grouping Accuracy [48] 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 [22, 37]. 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) [30] 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 [26].

V. 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 [22, 37]. In this RQ, we study the effectiveness of LibreLog.

Approach. We compare LibreLog with other state-of-the-art log parsers, including AEL, Drain, LILAC, and

TABLE I
A COMPARISON OF THE GROUPING ACCURACY (GA) AND PARSING ACCURACY (PA) FOR THE STATE-OF-THE-ART PARSERS AND LIBRELOG.

	AEL		**Drain**		**LILAC**		**LLMParser$_{T5Base}$**		**LibreLog**	
	GA	PA	GA	PA	GA	PA	GA	PA	GA	PA
HDFS	0.9994	0.6213	0.9990	0.6210	**1.0000**	0.9480	0.8295	0.9663	**1.0000**	**1.0000**
Hadoop	0.8230	0.5350	0.9210	0.5410	0.9240	0.7850	0.8376	0.8370	**0.9625**	**0.8706**
Spark	–	–	**0.8880**	0.3940	0.8700	0.7080	0.2589	0.4231	0.8586	**0.8887**
Zookeeper	**0.9960**	0.8420	0.9940	0.8430	0.9970	0.3780	0.7192	0.8408	0.9932	**0.8499**
BGL	0.9146	0.4062	**0.9190**	0.4070	0.8335	0.8239	0.1439	0.2440	0.9024	**0.9293**
HPC	0.7480	0.7410	0.7930	0.7210	**0.8450**	0.7350	0.6423	0.7070	0.8440	**0.9730**
Thunderbird	0.7859	0.1635	0.8310	0.2160	0.7940	0.3860	0.5790	0.3472	**0.8699**	**0.6940**
Linux	**0.9160**	0.0820	0.6860	0.1110	0.7636	0.7001	0.4999	0.8802	0.9120	**0.9017**
HealthApp	0.7250	0.3110	0.8620	0.3120	**0.9930**	0.6730	0.8364	0.9674	0.8617	**0.9735**
Apache	**1.0000**	0.7270	**1.0000**	0.7270	0.9970	0.9920	0.8680	0.9900	**1.0000**	**0.9960**
Proxifier	0.9740	0.6770	0.6920	0.6880	0.5060	0.7780	0.8276	**0.9817**	0.5101	0.8970
OpenSSH	0.7050	0.3640	0.7070	0.5860	0.7480	0.6550	0.2183	0.7812	**0.8678**	0.4955
OpenStack	0.7430	0.0290	0.7520	0.0290	0.5240	0.4860	**0.9517**	**0.9412**	0.8114	0.8308
Mac	0.7970	0.2450	0.7610	0.3570	0.8090	0.4480	0.6710	0.6131	**0.8141**	**0.6538**
Average	0.8559	0.4418	0.8432	0.4681	0.8289	0.6783	0.6345	0.7514	**0.8720**	**0.8538**

Note: The highest values of GA and PA 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.

LLMParser$_{T5Base}$. Among the non-LLM-based parsers, heuristic-based and tree-based approaches have superior performance [19]. Hence, we selected representative parsers, AEL [18] and Drain [14], for comparison. Drain is a tree-based method that achieves the best accuracy and efficiency among all traditional parsers [19]. AEL is a representative heuristic-based log parser with the second-highest accuracy [19]. These two parsers are also commonly used in previous research for comparison [20, 30, 31]. LILAC [20] and LLMParser$_{T5Base}$ [31] are recently proposed LLM-based parsers with high parsing accuracy. Since LILAC uses 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 [10] (240M parameters) as the LLM for LLMParser by following the prior work. Note that both 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 both Grouping Accuracy (GA) and Parsing Accuracy (PA).

Results. Table I shows the GA and PA for each log parser across different systems. LibreLog achieved the highest GA and PA values for most systems, indicating superior performance in both grouping and parsing logs. Across all systems, LibreLog achieved an average GA of 0.8720 and an average PA of 0.8538, outperforming all other parsers.

LibreLog shows superior GA and PA compared to the semi-supervised LLM-based parser – LILAC. Compared to LibreLog, LILAC demonstrated lower performance with a GA of 0.8289 and PA of 0.6783. 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 0.6783, while LibreLog's PA is 0.8538). 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.

Among all three LLM-based log parsers, LLMParser$_{T5Base}$ shows the lowest GA, and the reason may be the limited number of fine-tuning samples that makes it hard to generalize to large-scale datasets. Among the three LLM-based log parsers (LLMParser$_{T5Base}$, LILAC, and LibreLog), LLMParser$_{T5Base}$ exhibited the lowest GA of 0.6345 and the second-highest PA of 0.7514. When parsing large-scale datasets, logs may exhibit many different variations, even if they share the same log 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. This limitation becomes particularly evident in systems with more logs, such as BGL and Spark, where LLMParser$_{T5Base}$ struggles to achieve high GA (0.1439 and 0.2589, respectively). Nevertheless, it is still able to identify all dynamic variables in a log with the second-highest PA among all five parsers, which shows the potentials of LLM-based parsers.

Syntax-based log parsers generally have significantly lower PAs 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 (1.84% and 3.3% lower, respectively) and significantly lower PA (48.25% and 45.17% lower, respectively). This performance disparity is likely linked to their heuristic-based nature, which relies on predefined rules to identify 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 PAs. In contrast, LibreLog leverages pre-grouping and uses memory mechanisms to achieve high GA, and its LLM-based parsing process accurately identifies variables within grouped logs, resulting in superior PA.

> LibreLog achieves superior GA and PA 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 2.7 and 40 times faster than Lilac and LLMParser$_{T5Base}$, respectively.*** Table II 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: 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. 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 LLMParser$_{T5Base}$. 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 evaluates 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 does 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 III shows the total time, GA, and PA of LibreLog compared to replacing the log selection process with cosine similarity and random sampling. When employing cosine similarity to select representative logs, both GA and PA experienced declines of 2.5% and 4.8%, respectively, compared to using Jaccard similarity. This indicates that although cosine similarity is shown to be an effective similarity metric for text data [39], 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 due to the lack of diversity in the sampled logs, both GA and PA are even lower, at 0.849 and 0.806, respectively.

Although the self-reflection mechanism requires additional processing time, it significantly enhances the parsing results of LibreLog. Table III compares full version LibreLog and LibreLog without self-reflection in the total execution time, GA, and PA. 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 both GA and PA by 7.1% and

TABLE II
NUMBER OF LOGS AND PARSING TIME, IN SECONDS, FOR THE STATE-OF-THE-ART (FIRST FOUR COLUMNS) AND LIBRELOG.

		AEL	**Drain**	**LILAC**	**LLMParser$_{T5Base}$**	**LibreLog**			
	Log count	Total time	Total time	Total time	Total time	Total time	LLM query time	Grouping time	Memory search time
HDFS	11,167,740	5,711.52	1,343.56	1,162.20	148,097.72	1,252.62	273.67	867.36	111.59
Hadoop	179,993	361.54	19.54	4,747.52	4,034.32	285.76	268.81	11.95	5.01
Spark	16,075,117	10 days+	1,539.88	3,346.08	225,046.88	1,752.40	631.66	764.12	356.62
Zookeeper	74,273	3.22	7.12	1,702.46	1,585.52	52.23	47.06	4.75	0.42
BGL	4,631,261	29,917.35	501.09	8,624.70	90,526.27	1,244.64	857.82	298.23	88.59
HPC	429,987	18.00	39.02	388.88	4,634.87	539.76	510.97	27.45	1.33
Thunderbird	16,601,745	25,199.44	2,132.20	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	2,374.83	1,031.82	216.03	213.42	1.78	0.82
HealthApp	212,394	976.74	17.88	1,182.27	3,139.20	103.33	85.25	10.38	7.70
Apache	51,977	3.20	5.42	122.79	1,056.53	18.92	15.19	3.36	0.37
Proxifier	21,320	1.69	2.96	681.65	821.44	871.52	868.99	2.47	0.07
OpenSSH	638,946	1,338.67	74.12	1,134.35	15,262.29	89.37	36.94	49.00	3.44
OpenStack	207,632	30.28	60.18	1,260.64	7,558.24	377.64	330.66	44.88	2.09
Mac	100,314	10.79	16.94	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	4,212.52	66,395.26	1,528.52	1,100.44	254.41	173.67
Total	50,416,620	17.66 hours	1.60 hours	16.38 hours	258.20 hours	5.94 hours	4.28 hours	0.99 hours	0.68 hours

TABLE III
LIBRELOG PERFORMANCE UNDER DIFFERENT SETTINGS. THE NUMBERS IN THE PARENTHESIS INDICATE THE PERCENTAGE DIFFERENCE COMPARED TO THE FULL VERSION OF LIBRELOG.

	Total Time	**GA**	**PA**
LibreLog	5.944 hours	0.872	0.859
w/ cosine similarity	5.745 (↓3.3%)	0.85 (↓2.5%)	0.818 (↓4.8%)
w/ random sampling	5.458 (↓8.2%)	0.849 (↓2.6%)	0.806 (↓6.2%)
w/o self-reflection	3.292 (↓44.6%)	0.81 (↓7.1%)	0.777 (↓9.5%)

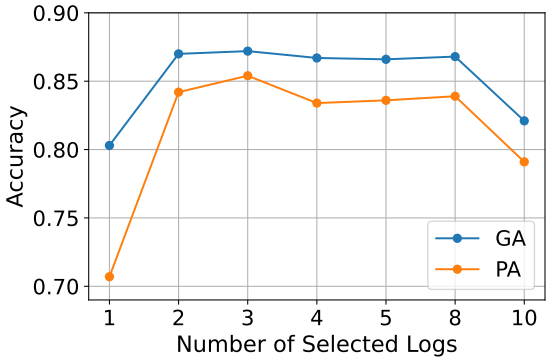

Fig. 4. GA and PA of LibreLog using different numbers of selected logs in the prompt.

9.5%, respectively. This shows that self-reflection significantly enhances the parsing effectiveness of LibreLog, although at the expense of increased overhead due 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. Fig 4 shows the LibreLog performance with variations in the number of logs from a group retrieved into the prompts. LibreLog maintains high accuracy across various sample sizes, with optimal performance achieved when the sample size is set to three, reaching peak values in both GA and PA. Notably, when the sample size is reduced to one, GA and PA drop to 0.80 and 0.70, respectively, representing a decline of 8.26% and 18% compared to a sample size of three. This reduction highlights the challenges LLM faces in parsing logs accurately without sufficient comparative data, such as multiple log comparisons or labeled logs. As the sample size increases from one to two, both GA and PA show significant improvements, peaking when the sample size reaches three. However, further increases in sample size from three to eight result in slight decreases in GA and PA, stabilizing around 0.865 and 0.835, respectively. This suggests that an excess of log samples may introduce noise, subsequently lowering performance [47]. Importantly, when the sample size reaches 10, both GA and PA decrease compared to a sample size of eight. This decrease is attributed to prompt truncation caused by an overload of retrieved logs, which exceeds the context size of the LLM, 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 [20, 23, 44] that are based on commercial LLMs, LibreLog employs 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 performance 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 performance with different LLMs, including Mistral-7B [17], CodeGemma-7B [42], and ChatGLM3-6B [13]. 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 IV shows the parsing performance using various LLMs. Among all, Llama3-8B achieved the best overall results.

Compared to Llama3-8B, Mistral-7B requires a slightly longer parsing time, yet achieves a similar GA and a noticeable decline in PA. Mistral-7B is a general model with training objectives and parameter sizes similar to Llama3-8B. However, it exhibits a lower PA, decreased by 14.6%, with comparable results in parsing time and GA. This discrepancy in PA may be attributed to Llama3-8B's enhanced pre-training data, which includes more code [4], and its 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 both GA and PA face a decline compared to Llama3-8B. CodeGemma-7B completes the parsing of all logs in only 71.5% of the time required by Llama3-8B. However, CodeGemma-7B does not achieve comparably high accuracy, indicating that while it is capable of generating log templates that match the logs, it struggles to consistently and accurately abstract variables within these logs. Nevertheless, using CodeGemma-7B still achieves higher GA and PA than 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.

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.

VI. Threats to Validity

External validity. Data leakage is a potential risk of LLM-based log parsers [20, 31]. 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 [19] 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.

TABLE IV
Parsing performance of LibreLog using different LLMs.

	Total Time	GA	PA
Llama3-8B	5.94 hours	0.872	0.854
Mistral-7B	6.78 (↑14.1%)	0.876 (↑0.5%)	0.729 (↓14.6%)
CodeGemma-7B	4.25 (↓28.5%)	0.814 (↓6.7%)	0.752 (↓11.9%)
ChatGLM3-6B	14.56 (↑145.1%)	0.837 (↓4%)	0.600 (↓29.7%)

Internal validity. 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.

VII. 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 0.8720 and an average PA of 0.8538, outperforming state-of-the-art parsers (i.e., ILIAC [20] and LLMParser [31]) by 5% and 25%, respectively. LibreLog parses logs from all 14 systems (50 million logs) in a total of 5.94 hours, which is 2.75 and 40 times faster than other

LLM-based parsers 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] C Aicardi, L Bitsch, and S Datta Burton. Trust and transparency in artificial intelligence. ethics & society opinion. european commission. 2020.

[7] 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.

[8] 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.

[9] 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.

[10] Hyung Won Chung et al. Scaling instruction-finetuned language models, 2022.

[11] 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.

[12] 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.

[13] Team GLM et al. Chatglm: A family of large language models from glm-130b to glm-4 all tools, 2024.

[14] 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.

[15] 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.

[16] 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.

[17] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] Zanis Ali Khan, Donghwan Shin, Domenico Bianculli, and Lionel Briand. Impact of log parsing on log-based anomaly detection. *arXiv:2305.15897*, 2023.

[23] 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.

[24] 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.

[25] Yukyung Lee, Jina Kim, and Pilsung Kang. Lanobert: System log anomaly detection based on bert masked language model. *arXiv preprint arXiv:2111.09564*, 2021.

[26] Zhenhao Li, Chuan Luo, Tse-Hsun (Peter) Chen, Weiyi 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.

[27] Feng Lin, Dong Jae Kim, et al. When llm-based code generation meets the software development process. *arXiv preprint arXiv:2403.15852*, 2024.

[28] 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.

[29] 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.

[30] 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.

[31] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. Llmparser: 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.

[32] 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.

[33] Devjeet Roy, Xuchao Zhang, Rashi Bhave, 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] Amit Singhal et al. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

[40] 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.

[41] P.N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. 2016. ISBN 9789332586055.

[42] CodeGemma Team et al. Codegemma: Open code models based on gemma, 2024.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.