Self-Evolutionary Group-wise Log Parsing Based on Large Language Model

Anonymous Author(s)

***Abstract*—Log parsing involves extracting appropriate templates from semi-structured logs, providing foundational information for downstream log analysis tasks such as anomaly detection and log comprehension. Initially, the task of log parsing was approached by domain experts who manually designed heuristic rules to extract templates. However, the effectiveness of these manual rules deteriorates when certain characteristics of a new log dataset do not conform to the pre-designed rules. To address these issues, introducing large language models (LLM) into log parsing has yielded promising results. Nevertheless, there are two limitations: one is the reliance on manually annotated templates within the prompt, and the other is the low efficiency of log processing. To address these challenges, we propose a self-evolving method called *SelfLog*, which, on one hand, uses similar <group, template> pairs extracted by LLM itself in the historical data to act as the prompt of a new log, allowing the model to learn in a self-evolution and labeling-free way. On the other hand, we propose an N-Gram-based grouper and log hitter. This approach not only improves the parsing performance of LLM by extracting the templates in a group-wise way instead of a log-wise way but also significantly reduces the unnecessary calling to LLMs for those logs whose group template is already extracted in history. We evaluate the performance and efficiency of *SelfLog* on 16 public datasets, involving tens of millions of logs, and the experiments demonstrate that *SelfLog* has achieved state-of-the-art (SOTA) levels in 0.975's GA, and 0.942's PA. More importantly, without sacrificing accuracy, the processing speed has reached a remarkable 45,000 logs per second.**

***Keywords*—large language model, log parsing, self-evolution**

I. Introduction

Logs [1] and time series [2], along with trace [3], jointly constitute the three vital types of data for monitoring and analyzing the reliability of software systems. Log data is the most easily acquired and the most widely encompassing. However, due to its semi-structured nature, it poses a greater challenge for analysis. Modern software systems such as operating systems like Windows and Android, or file systems like HDFS [1], generate a substantial volume of logs daily. Analyzing and detecting anomalies in such a vast number of logs manually is impractical. To achieve automatic log processing, **log parsing**, which involves transforming semi-structured logs into a structured format, making it the most crucial preliminary step for downstream tasks such as log anomaly detection [4], [5], log compression [6], [7], and log summarizing [8].

As shown in Fig. 1, log parsing primarily involves distinguishing between the constant and variable parts (named parameters in Fig. 1) of the log content through a series of methods. By replacing the variable parts with wildcards, we create a log template. It can be observed from Fig. 1 that

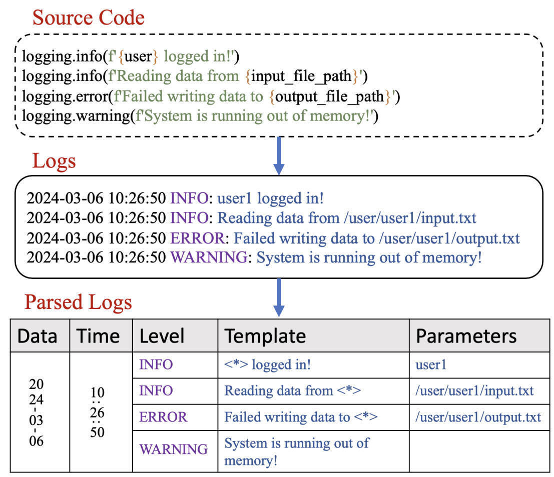


Fig. 1. An illustration example of log parsing.

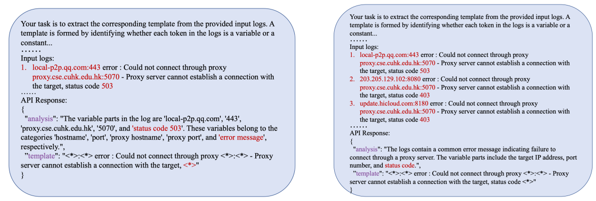
if the source code is available, it becomes much easier to extract the corresponding template. However, in many systems, the original software code is not accessible, leading to the emergence of many data-driven log parsing methods [9]–[11]. These data-driven methods are primarily categorized into unsupervised and supervised approaches. Unsupervised methods typically employ heuristic rules [12]–[14] or statistical features [9]–[11] to extract templates. However, the effectiveness of these methods can be greatly impacted if the log datasets to be analyzed do not align well with the pre-designed rules or features. For example, in Drain [12], it is assumed that the first token of each log is constant. Yet, in real-world systems, we have found that this is not always the case, such as with "`proxy.cse.cuhk.edu.hk:5070 close, 451 bytes sent, 353 bytes received, lifetime <1 sec`", where the first token is variable. Supervised log parsing methods [15], [16], on the other hand, train or fine-tune models using manually annotated <log, template> pairs or token types. However, these methods are sensitive on the distribution of the training data and may perform poorly on logs unseen during training.

To address the issues with the aforementioned methods, some current research has begun to employ Large Language Models (LLMs) for log parsing [17]. Logs are inherently statements printed by programmers, naturally containing semantic information, and LLMs are inherently capable of extracting semantic information. Furthermore, LLMs, due to their powerful zero-shot capabilities, can better transfer to a new set of logs without the need for additional hyperparameter adjustments. Thus, using LLMs for log parsing is a promising direction. However, existing LLM-based log parsing methods have the following two drawbacks:

- Current methods, such as DivLog [17], provide the LLM with prompts containing similar logs and corresponding templates, enabling the LLM to extract new log templates through in-context learning (ICL). However, this approach heavily relies on the quality of the examples in the prompt. When software systems undergo upgrades and iterations, a new round of manual annotation of new log templates is required.
- Most importantly, existing LLM-based methods do not explicitly evaluate and discuss the log processing speed and token cost after deployment, which is key to determining whether LLMs can be widely applied in log parsing. We find through actual measurement that existing LLM-based methods can only process no more than 10 logs per second (as detailed in Section IV-F), whereas in real software systems, it is very common for tens of thousands of logs to be generated per second.

To address these challenges, we propose a self-evolutionary group-wise LLM-based log parsing system called ***SelfLog***. Although the templates extracted by the LLM from historical logs are not as accurate as those of domain experts, they can provide useful information for LLM to revise its responses. Inspired by this, we build a *Prompt Database* to store LLM history extracted <log, template> pairs. When a new log arrives, the most similar logs can be retrieved through an approximate nearest neighbor (ANN) search. Then, by incorporating similar logs and the templates previously extracted by the LLM itself into the prompt, it can help the LLM reflect on the correctness of the previously extracted templates, thereby improving the extraction performance for new logs and helping *SelfLog* no longer rely on manual annotation by human experts.

In the meantime, we observe that existing LLM methods process logs one at a time for template extraction, as depicted in Fig. 2(a), a method we refer to as point-wise parsing. However, this approach overlooks certain information. Typically, domain experts determine the variables in a log by comparing it with several similar logs and identifying the differences between them. As shown in Fig. 2(a), if a single log is given to an LLM, it might incorrectly identify `status code 503` as a variable. Yet, if we present a group of similar logs to the LLM, as shown in Fig. 2(b), the model accurately identifies `503` as the variable and `status code` as a constant, resulting in the correct template: `status code <*>`. Therefore, we design an *N-Gram-based Grouper* that first groups the logs and then invokes the LLM to extract templates for each group. This method not only enhances accuracy but also has the advantage of reducing the need to invoke the LLM for each log. We only need to call the LLM to extract templates for each group. For groups whose templates have already been extracted, we store the templates in our designed *Log Hitter*. Groups that hit *Log Hitter* return directly without invoking the LLM, which significantly increases the parsing speed of *SelfLog* and substantially reduces the number of tokens required for model calls.



(a) Point-wise parsing. (b) Group-wise parsing.

Fig. 2. An example from the Proxifier dataset [1], demonstrating the template extraction effectiveness of LLM when each log is given to the model, versus when several similar logs are grouped together and then given to the model.

We evaluated *SelfLog* on 16 public datasets, and its performance exceeded the current state-of-the-art (SOTA). In Group Accuracy, its precision was 10% higher than SOTA, and in Parsing Accuracy, it improved by 16%. At the same time, we tested it on a dataset of tens of millions of logs, and our processing rate reached 45,000 logs per second, meeting the log generation rate of current online systems.

In summary, our main contributions are as follows:

- We propose ***SelfLog*** , a self-evolutionary group-wise log parser that achieves excellent log parsing results without the need for manual annotation of new templates, relying solely on LLM's historical parsing results through continuous reflection and correction.
- For the first time, we focus on the efficiency issue in LLM-based log parsing. By combining a specially designed N-Gram-based Grouper, Log Hitter, *SelfLog* can parse over 45,000 logs per second with higher parsing accuracy. At the same time, *SelfLog* only consumes 1% tokens quota compared with the SOTA LLM-based log parsing method, making it affordable for real-world systems.

The following sections of the paper are organized as follows: In Section II, we introduce the motivation. In Section III, we detail the implementation of our *SelfLog*. In Section IV, we describe the experimental setup and evaluate the algorithm. Section V discusses threats to validity. Section VI reviews related work, and Section VII summarizes the paper.

II. MOTIVATION AND BACKGROUND

In this section, we present a broad definition of system logs and explain the basic steps involved in log parsing. We also introduce the large language models (LLM), as well as the background knowledge related to in-context learning (ICL) and prompts.

A. Log Parsing

The system generates a large amount of logs every day [18], [19]. It is unrealistic to rely on operation and maintenance person to manually check the logs to detect system abnormalities in time. Therefore, fully automatic log processing is necessary [1], [20]. The first step in log processing is log parsing, which processes semi-structured and unstructured logs and converts them into structured data for other downstream tasks such as log anomaly detection [4], [5], log compression [6], [7], and root cause analysis [21], [22]. Log parsing includes preprocessing and log template extraction, as shown in Fig. 1. The original log includes the timestamp automatically stamped by the system, verbosity level, and log content written in the program code. The formats of the first two parts are bound to the system settings and only require fixed regular expressions to be aligned and extracted. The log content consists of a constant string written in the code and a part that changes dynamically according to the system status. The log parser extracts the constant parts from the log content and replaces the variable parts with wildcards, such as the asterisk (*). This text string, composed of constants and wildcards, is commonly referred to as a log template. A single log template corresponds to multiple logs where the variables take on different values. The earliest log parser directly parsed the system code and extracted the log template according to the log output statement [23], [24] which is shown in the source code part of Fig. 1. In many scenarios, the original source code of the system itself is not accessible, hence there is a substantial amount of work that relies on extracting templates directly from the output logs themselves, which can be divided into unsupervised method [9], [10], [12] and supervised method [15], [16], [25]. However, there are still various shortcomings and the effectiveness need to be improved.

Unsupervised log parsers, which propose heuristic rules to extract log templates based on the author's observation of logs, often have design flaws that cannot correctly parse all logs. For example, when Drian [12] clusters logs, it first divides them according to the length after word segmentation. However, in the publicly available dataset Proxifier [1], there are two log entries: `...received, lifetime <1 sec` and `...received, lifetime 00:02`. After tokenization by whitespace, these two logs will have different lengths, but they actually belong to the same template. Furthermore, Drain [12] assumes that the first token of each log is a constant, but in Proxifier, there are logs that start with variables. These examples illustrate that existing methods based on heuristic rules or statistical features require special treatment for different datasets. If the dataset is not properly handled, the accuracy of template extraction can be greatly reduced.

Supervised log parsing requires training a template extraction model, or fine-tuning a pre-trained model, based on a dataset that has been manually annotated. However, manual annotation is costly, especially when the system generating the logs is dynamically changing and undergoing updates and upgrades, making manual annotation even more difficult. VALB [25] trains a BiLSTM [26] model by manually annotating variable types to distinguish between the types of constants and variables. They have predefined nine types of variables, such as Object ID (OID) and Location Indicator (LOI). In real-world datasets, such as LogPai [1], some corresponding templates have very few logs associated with them, which is insufficient for model training or fine-tuning.

B. Large Language Model and In-Context Learning

Large Language Models (LLMs) are a subcategory of machine learning models. Initially, they are used in the field of Natural Language Processing (NLP) [27] to understand and generate text that is readable by humans. Subsequently, their application has been extended to include images (PLEASE add reference) and speech (PLEASE add reference). The widespread use of LLMs is not only due to their large number of parameters but, most importantly, their impressive ability to follow instructions, which allows them to be employed for a variety of different downstream tasks and demonstrates their strong zero-shot capabilities. These tasks include translation, text generation, and logical reasoning, among others.

In-context learning (ICL) is a vital aspect of LLMs [28]. In the context of Large Language Models (LLMs) such as GPT-3 [29], a prompt represents the initial user input that kickstarts the process of text generation by the model. The nature and specificity of the prompt broadly determine the direction and content of the generated response [30]. Prompts can range from single words to complex paragraphs. Through their training on diverse and extensive data, LLMs can handle a wide spectrum of prompts, generate appropriate responses, and provide insights or narratives based on them. In essence, the prompt is the steering wheel that guides the text generation journey of the LLM. The ability to absorb, interpret, and utilize the shared contextual information in the conversation to provide more relevant and useful responses [31] of LLM is called ICL.

The emergence of Large Language Models (LLMs) and Inductive Conformal Logic (ICL) has provided new insights for robust and universal log parsing. System logs are output to log files by programmers through code. They are records of events that happen within a software application, system, or network [24]. To facilitate operation and maintenance people to monitor the system based on the logs, the logs are highly readable and very similar to natural language. Besides, there is also log information in the training data of LLM [32]. With the human knowledge and in-context examples in prompts, DivLog [17] found that it can achieve better results than SOTA log parsers in log template extraction without fine-tuning LLM. In real scenarios, the number and content of system log templates change dynamically. Traditional log parsers may need to re-train or re-design rules to improve the parsing effect of new logs. With the powerful natural language understanding capabilities of LLM, LLM-based Log Parsers only need to change the in-context examples in prompt to achieve accurate parsing of new log.

However, even though LLMs can effectively understand logs, their calling costs and the speed at which they generate templates are quite limited. In industrial systems, it is very common to produce tens of thousands of logs in a single minute, which presents a challenge for large models to process within such a short time frame. In Section III, we design

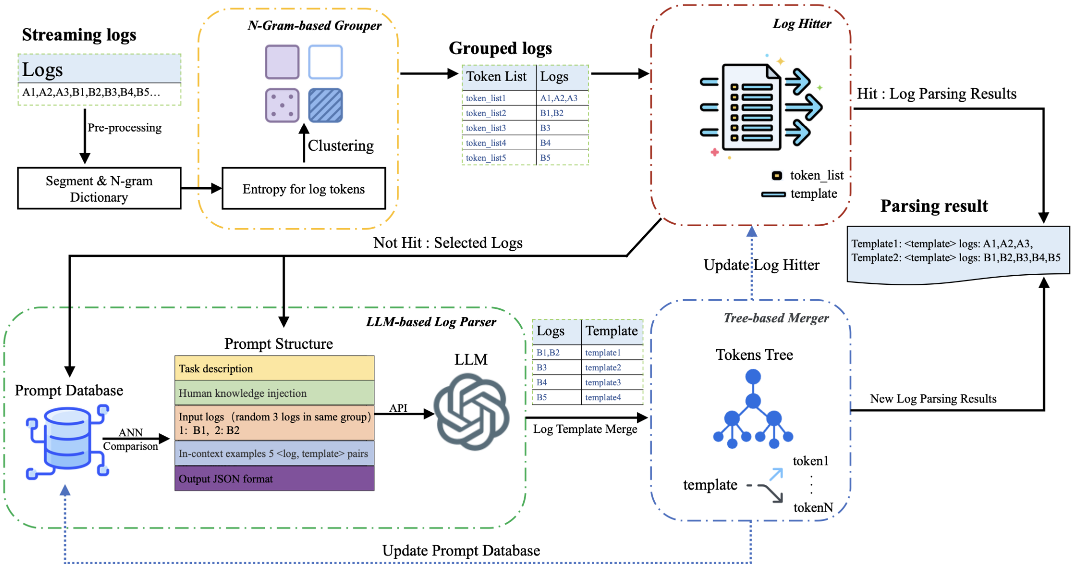


Fig. 3. The workflow of *SelfLog* framework. Here we demonstrate how to process streaming logs, but this system is also capable of seamlessly handling offline logs. In the case of offline log processing, only a forward pass (represented by the solid black line in the diagram) is necessary, without the need for iterative updates to the online database operations (represented by the dashed blue lines in the diagram).

algorithm that utilizes a self-evolution approach to generate higher-quality prompts without the need for manual expert annotation, thus enhancing the effectiveness of log parsing. Concurrently, we have also designed a framework to increase the efficiency of log processing using LLMs, aiming to increase the log processing rate of existing LLM-based log parsing systems(ADD CITATION) from one log per second to tens of thousands of logs per second.

III. OUR APPROACH

In this section, we present *SelfLog*, a self-evolutionary log parsing system that focuses on how, through a self-evolutionary approach, Large Language Models (LLMs) can achieve good results without relying on manually annotated <log, template> data. Additionally, it addresses the issue where the log processing efficiency is influenced by the generation rate of the LLM. We first provide an overview of *SelfLog*, followed by a detailed introduction to each of its key modules. It is noteworthy that we will introduce the system with a focus on the most common scenario in industrial systems: streaming logs. In a streaming setup, logs are continuously generated, and downstream log parsing algorithms need to extract templates in real-time. At the end of this section, we will discuss how the scenario of offline analysis is actually a special case of online streaming analysis.

A. Overview of SelfLog

As shown in Fig. 3, the *SelfLog* system only needs to call the LLM API, through In-Context Learning (ICL), by providing a specially designed prompt to the LLM, to perform the task of log parsing without the need to train the LLM. The entire system is divided into four major modules: **N-Gram-based Grouper, Log Hitter, LLM-based Log Parser, and Tree-based Merger**.

The N-Gram-based Grouper is primarily used to cluster and group the preprocessed logs, which has two main goals. One is that it can greatly save on the financial cost of calling the large model, as we no longer need to call the model for each log entry. Instead, we only need to call the model once for a new group as a whole. Additionally, it can greatly enhance accuracy because if there is only one log, the model can only determine which token is a variable based on semantic information. However, if there are multiple logs within the same group, the model can determine which token is a variable by comparing which parts of the logs differ, which is also the core idea behind methods like Drain [12]. We are the first to apply this to LLMs.

With the Grouper in place, the design of the Log Hitter naturally follows, as the same group is likely to correspond to a same template. If this group already has an existing template, there is no need to make further calls to the large model, which is particularly important in an online environment. This is because repeated calls to the model not only greatly reduce processing efficiency but also, due to the hallucinations and uncertain outputs of large models, can lead to unstable parsing results. The LLM-based Log Parser mainly achieves good detection results through a carefully designed prompt and the method of ICL. The in-context examples in the prompt play an important role for the model to continuously revise its current output based on the previous outputs in a self-evolution way. The Tree-based Merger is primarily responsible for correcting the model's output because neither the Grouper nor the LLM can guarantee a 100% accuracy rate. The Merger will merge some logs that were incorrectly divided into multiple groups and templates, thereby enhancing the model's precision.

B. Pre-processing

The original logs contain the timestamps assigned by the system to the log content, and log types such as `INFO` and `ERROR`, process ID, etc. In the same system, these contents are all in fixed locations in the log, so they can be extracted through simple regular expressions. For example, “`17/06/09 20:10:40 INFO spark.SecurityManager: Changing view acls to: yarn,curi`” is a raw log from Spark [1] system. The regular expression “`[r'(\d+\.){3} \d+', r'\b[KGTM]?B\b', r'([\w-]+\.){2,}[\w-]+']`” can extract each part separately. Thus we only need to focus on the log content, which is printed by the system code through the log print statement (see Fig. 1), which contains fixed constants prewritten in the code and variables dynamically filled in based on system operating information.

Since methods based on statistical features [12] require calculating the characteristics of a segment of text within a log to determine whether that segment is a variable or a constant, how to segment a log text becomes critically important. Existing methods mostly involve segmenting a log by using delimiters, and converting it into separate segments of text, each of which we refer to as a token. However, this method of segmentation necessitates selecting appropriate delimiters for different datasets. For instance, for BGL [1] logs, the delimiters might be “`..()`”, while for Windows datasets [1], the delimiters could be “`=:[]`”. We believe that this approach to segmentation is not robust enough. With the advent of large models, we no longer need to strictly rely on such tokenization rules. In *SelfLog*, the purpose of tokenization during preprocessing is to facilitate subsequent log grouping, rather than directly extracting templates. Therefore, we only need to identify the commonalities across multiple logs and filter out as many of the variable parts as possible. Hence, we have chosen “`[A-Za-z0-9*]+`” as our tokenization rule. It can be seen that anything not composed of letters and numbers is used as a delimiter. For example, for the log “`pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=202.100.179.208 user=root`”, after our tokenization process, it becomes a list of constant tokens “`unix sshd auth authentication failure logname euid ruser rhost user root`”. By comparing the token list with the original log, we can observe that pure numeric sequences have been removed, as we assume that pure numbers are most likely variables and cannot represent a category of logs. Furthermore, to mitigate the potential impact of variables, we query the WordNet [33] lexicon for all tokens that, after tokenization, are three characters or fewer in length. If a token appears infrequently in WordNet, we consider it to be an invalid word and likely a prefix or suffix. In the previous example, tokens such as “pam”, “uid”, “tty”, and “ssh” were eliminated.

Algorithm 1: N-Gram-based Grouper

Input : log X

1 TX = get_token_list(X)
2 `// step1: find 2-gram const token`
3 position = get_2gram_const_index(TX)
4 `// step2: Get variable token list from right part`
5 variable_list_right = PILAR_gram(TX, position)
6 `// step3: Get variable token list from left part`
7 variable_list_left = PILAR_gram(TX, position)
8 `// step4: removing variable from TX`
9 CX = TX - variable_list_right - variable_list_left

Output: CX: constant tokens of log X

C. N-Gram-based Grouper

In the pre-processing phase, each log is represented with a list of tokens, and yet some of these tokens may be variables. We need to further identify these variables, remove them from the token list, and then use the token list for grouping, ensuring that logs within each group belong to the same template. It is worth noting that even if we do not identify all variables here, leading to logs of the same template being divided into two different groups, it is not a problem because later stages involving the Large Language Model (LLM) and the Tree-based Merger can correct them. We have improved upon the entropy-based method from PILAR [10] to determine whether a token is a variable or a constant, with the specific method detailed in Algorithm 1. The constants in the log are written in the code by programmers to facilitate the person to observe the system status and code debugging. Therefore, constant tokens are often words with higher frequency in the corpus. Different tokens are assigned different frequencies according to their frequency of occurrence in WordNet [33]. Function get_2gram_const_index (Line 3) calculates the largest sum of 2 consecutive token weights and returns their position. Then, starting from the position and moving to the right(Line 5), the algorithm employs the method from PILAR, using a 3-gram approach to dynamically determine whether each token is a variable. Each token is based on the ratio of the number of co-occurrences with its neighbors and the number of neighbor occurrences after removing itself, compared with the set threshold. If it is less than the threshold, it is considered a variable. Function PILAR_gram is the algorithm in listing 1 of the PILAR. It returns the variable_list_righ. Similarly, starting from the position and moving to the left, it determines whether each token is a variable (Line 7). Finally, return CX(Line 9).

Compared to PILAR, our N-Gram-based Grouper differs in the following two ways: Firstly, PILAR relies on assuming that the first word of the log is a constant, but this is often inaccurate because some logs start with variables. By checking the log templates in the Proxifier ground truth, we found that 2000 logs all start with variables. Because the algorithm in

PILAR defaults to the first token as a constant, the execution direction of the algorithm is from left to right. We judge whether a token is a variable based on the weight value, the starting point of the algorithm needs to be executed in both directions from right to left and from left to right to calculate the entropy of log tokens. Secondly, Unlike PILAR, which sets thresholds based on expert experience, we directly set the threshold to be automatically adjusted according to the number of different logs to improve the robustness of the group stage.

After obtaining the list of constant tokens for each log through Algorithm 1, we then categorize the logs into different groups based on the token list. Each group is keyed by the token list, with the value being a list of logs that records all logs belonging to that group. Subsequently, the LLM-based Log Parser will extract the corresponding log template for each newly emerged group, and after the template is extracted, it will be updated into the log hitter in the form of a <token_list, template> pair.

D. Log Hitter

After grouping, the logs are divided into multiple groups according to the token list. The Log Hitter maintains a dictionary with the token list as the key and the log template as the value. The grouped logs will first be looked up in the dictionary according to the token list, and if there is a hit, the corresponding template will be directly returned to complete the log parsing. If there is no hit, the token list will be recorded as the key first, and the three logs with large editing distances in the group will be selected as the input of the LLM-based Log Parser, and the logs will be parsed by LLM. Finally, the log template obtained after Tree-based Merger processing is updated to the dictionary. Log Hitter records historical grouping information and continuously updates it. Only logs that have not appeared before are handed over to LLM for processing, which greatly improves the efficiency of log parsing.

E. LLM-based Log Parser

A model prompt is a brief text snippet provided to an LLM model to guide its generation of related content. Unless stated otherwise, we use GPT-3.5 as our LLM model, and we also evaluate the performance of other LLMs in the evaluation section. These prompts are typically crafted as questions, descriptions, or instructions to elicit the model's output on specific topics or styles. By cleverly constructing prompts, it's possible to steer the model towards generating text that aligns with expectations, thereby meeting user needs or accomplishing particular tasks. In this paper, we carefully design prompts to guide LLM in log template extraction. As shown by the different colors in Fig. 4, our prompts mainly consist of the following five parts, which we will introduce one by one.

Your task is to extract the corresponding template from the provided input logs. A template is formed by identifying whether each token in the logs is a variable or a constant. A constant refers to the part that is common to all logs of this category and does not change with different specific logs. A variable, on the other hand, refers to the part that has different values across various logs. By identifying the variables within the logs and substituting them with the wildcard '<*>', a template can be constructed.

Keep in mind that '*' is just a simple character, and it should not be understood as a multiplication sign. For example, (a) *7 is not aaaaaaa…

Input logs belong to the same template, So you can also use the differences to help you judge the variable part in the log :
1. PCI Interrupt Link [LNKA] (IRQs 3 4 5 6 7 10 11 12) *14
2. PCI Interrupt Link [LNKA] (IRQs 3 4 5 6 7 10 11 12) *18
3. PCI Interrupt Link [LNKB] (IRQs 3 4 5 6 7 10 11 13) *14

Here is the examples of the log to template task(This is the information I collected and may not be correct):
Example: log: PCI Interrupt Link [LNKB] (IRQs 3 4 5 6 7 10 11 12)
template: PCI Interrupt Link [<*>] (IRQs <*> <*> <*> <*> <*> <*> <*> <*>)
…

Output JSON format:
{
"analysis": "Provide a short explanation for variable(not more than 40 words)",
"Log Template": "Provide the template extracted from the new log entry."
}

Fig. 4. An example of the complete prompt. The yellow block is the task description and the green block is human knowledge. The apricot block is the selected three input logs from the same group. The blue block is an example dynamically selected based on the Approximate Nearest Neighboring (ANN) from the prompt database, which is one of the core designs of our work. The **purple** part specifies the output format.

Task Description: This part should be placed at the very beginning of the prompt to clearly state the task that the LLM needs to perform, and it is part of the instruction section. Our specific task description is shown in the figure. In addition to providing the model with instructions for log extraction, we also inform the model of the system to which these logs belong, activating the corresponding log training part within the model.

Human Knowledge Injection: This part is optional. If there is explicit knowledge that can be articulated in actual applications, it can be added here to enhance the model's expression. We include knowledge to inform the model that the asterisk (*) is not a multiplication sign but a representative of a wildcard, to prevent conflicts with other parts of the model's knowledge. Examples of log machine correspondence templates based on historical manual confirmation in DivLog can also be placed in this part. Therefore, our algorithm can be combined with DivLog to achieve better improvement.

Input Logs: This part is the main input corresponding to the log template extraction task. Through the design of the N-Gram-based Grouper mentioned earlier, our model no longer extracts templates from individual logs. Instead, we extract new templates for each group. So, when a new group that has not been seen before appears, we randomly select three logs with the greatest edit distance from this new group as the model's input for template extraction. In Fig. 2, we demonstrate the difference in final effect between using the LLM to perform template extraction on each log entry individually and feeding multiple similar logs into the LLM as a group for log parsing. From Fig. 2 (a), it can be seen that when a single log is fed to the LLM, the model will identify the `status code 503` as a variable type of `error message`, thereby recognizing the entire `status code 503` as a variable. In fact, the `status code` is a constant, and `503` is the variable. If similar logs are input into the model as a group, as shown in Fig. 2(a), the model sees both the `status code 503` and the `status code 403`, thus accurately identifying the variable part. We will evaluate in Section 5 the impact of choosing different numbers of logs in a group on the final outcome.

Self-evolution Examples: This part is the main part designed in this paper. DivLog [17] works by adding manually annotated logs and their corresponding templates to the prompt, but this still needs manual annotation for new logs. In this paper, we record the logs and their corresponding templates that the LLM has parsed in history, storing them in the Prompt Database. Each time a new log needs to be parsed, we retrieve the most similar historical logs and their templates from the data through an Approximate Nearest Neighbors (ANN) search, serving as the corpus for In-Context Learning (ICL). This approach not only allows for complete automation without the need for expert annotation but also enables the model to reflect on potential issues in previously extracted templates and make timely corrections.

Output JSON Format: The content provided by the LLM model is usually quite diverse and often includes some analysis and explanations of the problem. These outputs are usually mixed with the extracted templates. If there are no constraints on the model's output, it would be difficult to directly extract the answer from the large model's response. Therefore, we impose explicit constraints to let the LLM fill the analysis process and the final template into the pre-set json fields, which facilitates the subsequent accurate extraction of the log template from LLM's answers.

F. Tree-based Merger

We test existing large models and find that even models like GPT-3 cannot identify all variables. As shown in the example in Fig. 5, since logs are mostly entered in a streaming manner, and the initial logs are all from the user "cyrus", the model will extract a template with `session opened for user cyrus by (uid=<*>`. Following this, when logs from the user "news" are entered, the model will propose a corresponding template, and so on. When there is a period with logs from both "cyrus", "news", "test", and other users, the model can recognize that what lies between "user" and "by" is a variable. To address this issue, we construct a tree as depicted in Fig. 5. This tree updates in real-time based on the parsing results of the streaming data. By utilizing this tree, we can perform a double check of corner cases that the large model cannot accurately recognize, thereby enhancing the parsing performance of the model.

IV. Evaluation

In this section, we design detailed experiments to answer and verify the following six research questions:

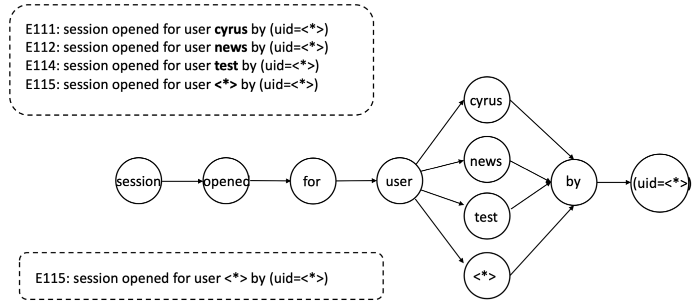


Fig. 5. Illustration of Tree-based Merger.

RQ1: Effectiveness of *SelfLog*. How does SelfLog perform in comparison to other state-of-the-art algorithms across the 16 publicly annotated datasets by LogPai [1]?
RQ2: Efficiency and Cost of *SelfLog*. Compared with the LLM-based log parsing method, how efficient is *SelfLog*?
RQ3: Ablation Study. How do the different constituents in our design contribute to overall performance?
RQ4: Parameter Sensitivity. How do configuration parameters affect the parsing effects?
RQ5: Parsing Speed. What is the maximum parsing speed at which *SelfLog* currently processes streaming logs?
RQ6: LLM Backbone. What is the impact of different LLMs on the *SelfLog* effect?

A. Experimental setup

1) Datasets: The experimental dataset comes from the real log data of 16 different systems open-sourced by LogPai [1]. LogPai manually labeled templates of 2K logs for each dataset.

2) Evaluation metric: Consistent with recent research findings [34], we employ Parsing Accuracy (PA), Precision Template Accuracy (PTA), and Recall Template Accuracy (RTA). Additionally, we have incorporated the Group Accuracy (GA) metric as used in the paper [12], [15], [16], [25], [34].

- GA (Group Accuracy) was initially introduced by the paper [34] and has since been adopted for strictly assessing the accuracy of log template extraction. It considers a template extraction to be correct only if all corresponding logs belonging to the same template are accurately extracted.
- PA (Parsing Accuracy) was first proposed by LogGram [9]. PA focuses on the consistency between the log template extracted by the algorithm and the ground truth. If all tokens in the log are correctly identified as constants and variables, the extraction is considered correct.
- PTA (Precision Template Accuracy) and RTA (Recall Template Accuracy) is proposed by Khan et al. [34]. PTA is measured by the percentage of correctly identified templates to the total number of identified templates, whereas RTA is measured by the percentage of correctly identified templates to the total number of ground truth templates.

TABLE I

ACCURACY COMPARISON WITH DIFFERENT LOG PARSERS ON LOGPAI DATASETS ([1]). THE BEST SCORES FOR EACH METRIC OF EVERY DATASET ARE BOLDED. DUE TO LIMITED TABLE SPACE, WE OMIT PTA AND RTA BECAUSE THEY SHOW CONSISTENT RESULTS WITH PA. IT IS NOTEWORTHY THAT, IN ADDITION TO THIS TABLE, WE INCLUDE GA, PA, PTA, AND RTA IN THE FOLLOWING FIGURES AND TABLES.

Dataset	LenMa		Spell		Drain		Logram		LogPPT		DivLog		*SelfLog*	
	GA	PA	GA	PA	GA	PA	GA	PA	GA	PA	GA	PA	GA	PA
HDFS	0.998	0.01	1.000	0.297	0.998	0.3545	0.940	0.005	0.845	0.389	0.143	0.966	**1.000**	**1.000**
BGL	0.690	0.082	0.787	0.197	0.963	0.342	0.645	0.125	0.478	0.789	0.451	**0.949**	**0.994**	0.934
HPC	0.830	0.632	0.654	0.5295	0.887	0.6355	0.906	0.643	**0.947**	0.927	0.194	**0.936**	0.924	0.909
Apache	1.000	0.000	1.000	0.694	1.000	0.694	0.314	0.0065	1.000	0.994	0.012	0.928	**1.000**	**1.000**
HealthApp	0.174	0.129	0.639	0.152	0.780	0.1085	0.279	0.112	1.000	0.6685	0.548	0.944	**1.000**	**1.000**
Mac	0.698	0.125	0.757	0.0325	0.787	0.218	0.520	0.169	0.778	0.490	0.548	0.771	**0.831**	**0.82**
Proxifier	0.508	0.000	0.527	0.000	0.527	0.000	0.027	0.000	1.000	0.000	0.025	0.895	**1.000**	**0.999**
Zookeeper	0.841	0.452	0.964	0.452	0.967	0.497	0.725	0.474	0.995	0.988	0.154	**0.976**	**0.993**	0.864
Thunderbird	0.943	0.026	0.844	0.027	0.955	0.047	0.189	0.004	0.257	0.473	0.256	**0.971**	**0.991**	0.933
Spark	0.884	0.004	0.905	0.3205	0.920	0.362	0.382	0.2585	0.4915	0.954	0.634	**0.967**	**0.997**	0.943
Android	0.880	0.714	0.919	0.245	0.911	0.709	0.791	0.413	0.885	0.331	0.523	0.842	**0.983**	**0.965**
Linux	0.701	0.122	0.605	0.088	0.690	0.184	0.147	0.124	0.389	0.388	0.185	**0.971**	**0.937**	0.868
Hadoop	0.885	0.0825	0.778	0.1125	0.948	0.269	0.428	0.113	0.787	0.384	0.291	**0.949**	**0.989**	0.902
OpenStack	0.743	0.019	0.764	0.000	0.733	0.019	0.236	0.000	0.503	0.872	0.092	0.744	**0.957**	**0.938**
Windows	0.566	0.1535	0.989	0.0035	0.997	0.159	0.695	0.1405	0.991	0.354	0.401	0.974	**0.996**	**0.994**
OpenSSH	0.925	0.133	0.554	0.1905	0.788	0.508	0.430	0.298	0.2295	0.9335	0.495	0.939	**1.000**	**0.997**
Average	0.766	0.167	0.792	0.208	0.865	0.319	0.478	0.18	0.723	0.62	0.309	0.920	**0.975**	**0.942**

3) Baselines: We compared the most advanced open-source log parsing methods. LenMa [35] clusters logs based on log similarity. Logram [9] distinguishes constant variables based on the frequency of log tokens. Drain [12] clusters logs based on rule trees. Spell [13] clusters logs based on the longest identical subsequence between logs. LogPPT [15] uses 32 logs to fine-tune the language model for log analysis. DivLog [17] uses LLM for log parsing by adding contextual knowledge to prompts.

B. Effectiveness of SelfLog

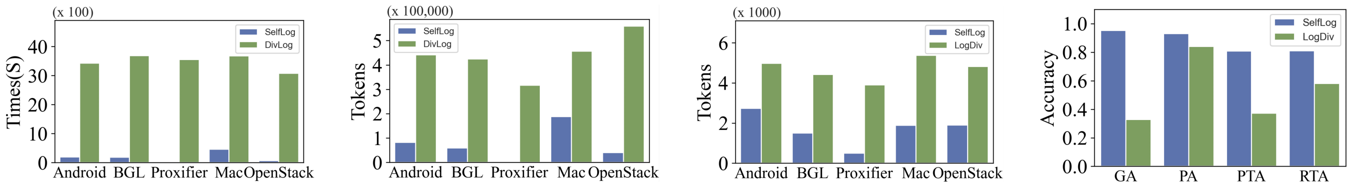
Table I displays the GA and PA of seven log parsing methods across the 16 datasets. *SelfLog* outperformed the other methods, achieving the highest average performance (see the bottom line of Table I) in both GA and PA. *SelfLog* also ranks as the best among existing algorithms in terms of PTA and RTA. Due to space limitations, to compare with more log parsers, we only selected the GA and PA to be displayed in the table. The PTA and RTA of *SelfLog* are shown in Table II below. It showed a 12.7% improvement in GA compared to Drain and a 51.9% improvement in PA compared to LogPPT. LogPPT and Logram methods are the most unstable. The accuracy of Logram on Proxifier dataset is only 0.027. This is because variables appear repeatedly in Proxifier dataset, causing many variables to be incorrectly recognized as constants. Drain has also achieved good results in both stability and average GA, but in Proxifier dataset the GA of Drain is only 0.527. Because all logs start with variables, Drain needs to perform group analysis based on the first few tokens, so the effect will be poor. This is because Drain assumes that the initial tokens in logs are constants, but in the Proxifier dataset, the majority of logs start with variables, leading to misjudgments by Drain.

C. Efficient and Cost of SelfLog

As shown in Table I, DivLog is the best method apart from *SelfLog*. Both our *SelfLog* and DivLog are based on LLMs, and the two most important metrics for using LLMs are processing time and cost. Therefore, we used 2000 logs from five representative datasets to compare the processing time of the two methods, as well as the number of input and output tokens, since LLMs are billed based on the number of tokens. In addition to these, we also detail PTA, RTA, PA, and GA as accuracy criteria. From Fig. 6, it can be seen that *SelfLog* is significantly lower than DivLog in both processing time and token size. Under the circumstances that the log processing accuracy of *SelfLog* is better than that of DivLog (as shown in Fig. 6 (d), with a 190.5% improvement in the GA metric and a 9.1% improvement in the PA metric), the processing time of Proxifier dataset for SelfLog is only 1% of that for DivLog, and the number of tokens is $\frac{1}{10}$ that of DivLog. The main reason behind this is that DivLog requires a call to the LLM for each log entry, whereas *SelfLog*, through N-Gram-based Grouper and Log Hitter, only needs to call the LLM when a new group appears. Since the LLM is currently the bottleneck in log processing, reducing the number of calls to the LLM can greatly improve the efficiency of log processing. Moreover, as shown in Fig. 7(b), giving a group of logs to the LLM for template extraction can better assist the model in finding differences in the logs, thereby preparing to identify constants and variables, and thus achieving better results.

D. Ablation Study

As shown in Table II, we sequentially removed the grouper, parser, and merger of *SelfLog* to observe changes in the model across four evaluation metrics. We did not perform an ablation on Log Hitter because it does not contribute to accuracy. Its main function is akin to a cache, capable of storing previously parsed log groups and their templates



(a) Run Time (b) Input Tokens (c) Output Tokens (d) Parsing Result

Fig. 6. Comparative histogram of log parsing effect between *SelfLog* and DivLog at running time, number of input tokens and output tokens when calling API, and log parsing effect.

for quick retrieval, eliminating the need for additional LLM invocations. The second row of Table II indicates that the component most affected within the entire *SelfLog* is the N-Gram-based Grouper. Upon its removal, GA dropped by 0.632, PTA by 0.658, RTA by 0.285, and PA by 0.19. Concurrently, the number of invocations of the LLM by *SelfLog* increased, leading to a significant rise in overall input and output tokens. The decline in efficiency is mainly due to the absence of grouping, every log entry requires an invocation of the LLM. The reason for the decline in effectiveness is illustrated in Fig. 7(b) with a detailed example, showing that presenting logs to the LLM in a grouped manner, as opposed to one by one, is more beneficial for template extraction, as the model can more accurately determine variables by comparing logs within the group.

Besides the Grouper, the second most impactful module on effectiveness is the LLM-based Log Parser, with declines of at least 0.5 in PA, PTA, and RTA. This is because, compared to statistical rule-based log parsing methods, LLM can make better judgments on whether each token is a variable or a constant leveraging its powerful natural language processing abilities. Without the LLM module, even with the presence of the Grouper, the accuracy of PA could only reach 0.434. Although the effectiveness of log parsing is already relatively high after the N-Gram-based Grouper and LLM-based Log Parser, Table II also shows that the final Tree-based Merger can enhance PA, PTA, and RTA one step further (more than 0.1). This is because logs are generally produced in a streaming manner, and it is possible that within a certain input window, a particular variable's token may appear frequently (as shown in Fig. 5) and be mistakenly identified as a constant. The Merger, through the construction of a token tree, can correct these misidentified variables, thereby improving the model's performance.

E. Parameter Sensitivity

In this section, we explore the impact of hyperparameters of our model on the outcome. There are three hyperparameters for the entire *SelfLog* system: the threshold used when dividing groups with N-Gram, the number of *Input Logs* from the same group fed into the prompt during log parsing with LLM, and the number of *Self-evolution Examples* selected from the prompt database. Their respective results are displayed in Table III, and Fig. 7(a) and Fig. 7(b). Firstly, we evaluate the impact of varying the N-Gram threshold in the Grouper

TABLE II
ABLATION STUDY RESULTS OF *SelfLog*. THE LAST THREE LINES RESPECTIVELY REPRESENT THE PARSING EFFECT AFTER REMOVING DIFFERENT COMPONENTS FROM *SelfLog* .

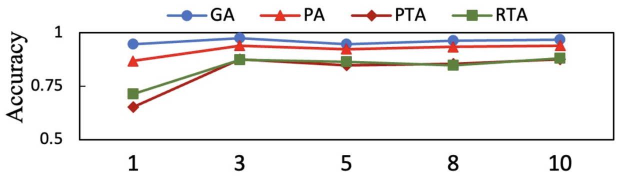
Variants	GA	PA	PTA	RTA
SelfLog	0.975	0.942	0.876	0.873
- N-Gram-based Grouper	0.343	0.752	0.218	0.588
- LLM-based Log Parser	0.943	0.434	0.345	0.346
- Tree-based Merger	0.932	0.837	0.626	0.791

TABLE III
THE AVERAGE GA UNDER DIFFERENT THRESHOLDS OF PILAR AND *SelfLog* ON 16 DATASETS, THE IMPROVED EFFECT IS THE IMPROVEMENT OF *SelfLog* RELATIVE TO DIVLOG. $lines$ REPRESENTS THE TOTAL NUMBER OF LOG ENTRIES.

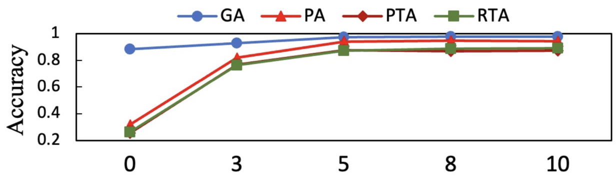
Threshold	GA of PILAR	GA of *SelfLog*	Improved effect
threshold=0.10	0.81	0.876	8.14%
threshold=0.15	0.82	0.876	6.82%
threshold=0.20	0.82	0.877	6.95%
threshold=0.25	0.82	0.874	6.58%
threshold=0.30	0.79	0.870	10.12%
threshold=0.35	0.80	0.891	11.37%
threshold=0.40	0.81	0.891	10.00%
threshold=0.45	0.80	0.889	11.13%
threshold=0.50	0.79	0.889	12.53%
threshold=$1/lines * 5$	-	0.877	6.95%
fluctuation range	$0 \sim 0.03$	$0 \sim 0.019$	-

on GA. We also examine the effects on other metrics such as PA, PTA, and RTA with parameter variation, with similar conclusions. Table III shows that our method maintains a high level of performance across different threshold values, with an improvement of at least 6.82% over DivLog [17], ranging from 0.876 to 0.891. Compared to PILAR [10], a method specifically optimized for parsing robustness, our fluctuation across different parameters is 0.019, which is 63% of PILAR's fluctuation (0.019 *v.s.* 0.3), where a smaller fluctuation indicates better stability. It is noteworthy that the grouping threshold can be removed one step further. We propose a heuristic rule that the threshold for determining whether a token is variable using N-Gram can be dynamically adjusted by the total number of log lines, *i.e.,* $\frac{1}{lines*5}$.

Regarding the number of representation logs in the same group for extracting the template, Fig. 7(a) reflects that the model stabilizes when the number of log entries exceeds 3.



(a) Number of *Input Logs*.



(b) Number of *Self-evolution Examples*.

Fig. 7. The impact of varying quantities of *Input Logs* and *Self-Evolution Examples* on model performance.

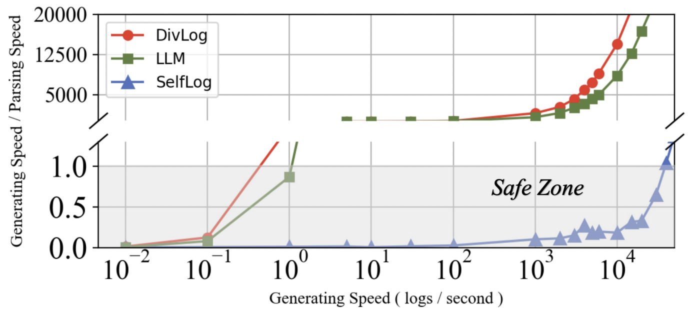


Fig. 8. Parsing speed of different LLM-based log parsing methods.

When the number of log entries for the same group increases from 1 to 3, GA improves by 6.7%, and PA by 8.2%, with the specific reasons introduced in Section III and Fig. 2 of the paper. As shown in Fig. 7(b), it is evident that without self-evolution examples, the model performs poorly. When the number of self-evolution examples increases from 0 to 3, PA improves significantly from 0.3 to 0.82. However, when the number of selected examples exceeds 5, the model's performance tends to converge. This is because we use an Approximate Nearest Neighbors (ANN) method to select self-evolution examples from the prompt database, ensuring that as long as there are relevant logs, they can be retrieved. Thanks to LLM's powerful few-shot learning capabilities, we can achieve good results with few relevant examples. Further adding examples yields marginal improvements.

F. Parsing Speed

While employing LLMs for log parsing offers numerous advantages, such as their strong semantic understanding capabilities and the ability of ICL to enhance the results of log parsing, the reality is that existing logs are typically generated in a streaming fashion and require real-time template extraction for immediate downstream anomaly detection. It is quite common for a large distributed system to generate tens of thousands of logs per second. However, existing algorithms such as DivLog are constrained by the generation

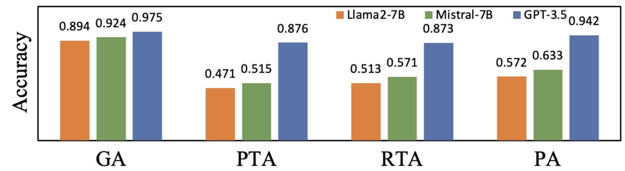


Fig. 9. A comparison of the performance of *SelfLog* when using different models as its backbone.

speed of LLM themselves. The generation speed of current large models is about 100 tokens per second [36], and a single log typically contains between 10 to 100 tokens in LogPAI [1], which means the rate can only reach a few logs per second. In contrast, our *SelfLog* benefits from a group-wise parsing paradigm and the caching mechanism of the log hitter, which significantly reduces the number of calls to the LLMs. As a result, the LLM is no longer a bottleneck. To get the exact parsing speed for existing LLM-based methods, including *SelfLog*, We use logs from HDFS [1] as input data, with 11,175,629 logs available. We replay these logs at different rates to test the log parsing speed of various models. In the experiment, we conduct multiple trials, each with a varying log generation speed, as shown in Fig. 8, where we test log generation speeds from 0.01 logs per second to 50,000 logs per second. We monitor the processing speeds of DivLog [17], vanilla LLM, and *SelfLog*, calculating the ratio of log generation speed to log parsing speed as the Yaxis. A ratio of less than 1.0 indicates that the log parsing speed exceeds the log generation speed, suggesting the model has sufficient capacity to handle more logs. Conversely, a ratio greater than 1 means the log parsing speed is less than the log generation speed, leading to a continuous backlog and, over time, potential Out-Of-Memory (OOM) issues. In Fig. 8, we mark the area where the Y-axis is less than 1 as the "safe zone". When Y equals 1, the corresponding X value represents the peak parsing speed supported by the algorithm. It can be observed from the figure that existing LLM-based log parsing algorithms, which require an LLM call for each log, have processing speeds of fewer than 10 logs per second and are already beyond the "safe zone" when the log generating speed exceeds 10 logs per second, resulting in a backlog. In contrast, *SelfLog* remains within the "safe zone" even when the log rate is 10,000 per second and reaches a remarkable peak parsing speed of 45,000 logs per second.

G. Model backbone

Fig. 9 demonstrates the performance of *SelfLog* when utilizing different LLMs as the backbone. It is evident that as the capabilities of the LLMs improve, the performance of *SelfLog* also continuously enhances. Due to our resource limitations, we have only tested the 7-billion-parameter open-source model. We believe that with the ongoing advancement of the LLM community, *SelfLog* can achieve further improvements in the future.

V. Threats to Validity

External Validity: In this article, we study and compare the effects of *SelfLog* and six state-of-the-art log parsing algorithms on 16 open-source datasets of LogPai [1]. Although these 16 datasets come from different systems, each dataset only has 2,000 manually labeled data, which does not represent logs in real scenarios. In the future, more realistic hand-labeled log datasets can be constructed to optimize the evaluation of various log parsers. We tested the efficiency and effect of *SelfLog* when processing a large number of logs in online work. Only testing the HDFS dataset cannot comprehensively and accurately display the online work efficiency of *SelfLog*. Further testing in real scenarios is needed.

Internal Validity: In the future, with the improvement of model capabilities, N-Gram-based Grouper may become a bottleneck limiting the effect of LLM on log analysis. When there is an error in classifying logs belonging to different templates into the same group, it will directly affect the final parsing results. But currently, *SelfLog* is still a robust, effective, and efficient log parsing algorithm.

Construct Validity: We set the *temperature* parameter of LLM as 0 to reduce the randomness of the results returned by LLM, but the results returned by LLM for the same input are still inconsistent. We record the experimental results through multiple experiments. Though ANN is better than the KNN used by DivLog [17] in terms of efficiency, it is not as good as KNN (K-Nearest Neighbors) in terms of retrieval accuracy.

VI. Related Works

A. Unsupervised log parsers

Unsupervised log parser does not require manual annotation of data for training and can be directly used in different systems for log parsing. Unsupervised log parsers can be further divided into frequent pattern mining-based [9]–[11], clustering-based [37]–[39], heuristic rule-based [12]–[14], and LLM-based methods [17]. Methods based on frequent pattern mining start from the data distribution itself and rely on data features (e.g. token frequent) to propose templates. The advantage is that it doesn't rely on artificially designed hyperparameters based on the data itself, and the method is highly robust (PILAR [10]). The disadvantage is that it is easily affected by the imbalance of data distribution. Logram [9] and LogCluster [11] all perform log analysis by extracting frequent patterns from logs. The clustering-based method adopts grouping first. By default, logs in the same group have the same template. Templates are proposed based on the differences in logs in the same group (different tokens are replaced with <$\star$>). LogMine [37] and LogTree [38] use the hierarchical clustering method to group logs, and LTE [39] use density-based clustering to group logs. LenMa [35] and FLP [40] adopt online grouping strategies to support online parsing. Based on the heuristic rule method, human knowledge is transformed into rules for log analysis by carefully observing the data. Drain [12], Spell [13], and IPLoM [14] have achieved good log parsing results by fine-tuning algorithm hyperparameters for different data. However, due to algorithm design flaws, they cannot correctly parse all log types and have poor robustness. The LLM-based method directly utilizes LLM's powerful natural language understanding capabilities. By providing a few context examples to build prompts, DivLog [17] has achieved the most advanced results in PA.

B. Supervised log parsers

Supervised log parsers usually use deep learning methods to train or fine-tune models by manually annotating data. VALB [25] manually annotate constants and variable categories using a method similar to named entity recognition, using the BiLSTM [26] model to understand and perform template extraction and variable category annotation. SemParser [16] extracts concept-instance (CI) pairs through the designed semantic miner, and then uses the joint parser to combine the context information to identify variables. LogPPT [15] proposes to use a small number of logs and template examples to fine-tune the pre-trained model RoBERTa [41] and then perform log analysis. However, the computation cost of fine-tuning is negligible.

VII. Conclusion

The advent of LLMs has presented a promising alternative for accurate log parsing, yet they come with their own set of challenges, particularly the need for manual annotation and the inefficiency of processing large volumes of logs. To overcome these obstacles, we introduce *SelfLog*, a self-evolving log parsing method that leverages the power of LLMs while mitigating their limitations. Our approach operates in two innovative ways: firstly, by using similar history <group, template> pairs outputted by LLM itself, which serves as prompts for new log entries, thus allowing the model to evolve and learn autonomously without the need for manual labeling. Secondly, we implement an N-Gram-based grouper and log hitter mechanism, which enhances the parsing performance by processing logs in groups rather than individually and significantly reduces redundant calls to LLMs for logs whose group templates have been previously extracted. Our comprehensive evaluation across 16 public datasets, encompassing tens of millions of logs, has demonstrated that *SelfLog*not only achieves state-of-the-art performance with a GA of 0.984 and a PA of 0.743 but also excels in efficiency, processing at a remarkable speed (over 45,000 logs per secon) compared with existing LLM-based log parsing methods. In a nutshell, by integrating N-Gram-based grouping with self-evolutionary in-context learning, *SelfLog* fully harnesses the advantages of LLM in few-shot learning while avoiding inefficiency pitfalls. We will continue to explore the application of this paradigm in log analysis in the future.

References

[1] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 121–130.

[2] Y. Chen, E. Keogh, B. Hu, N. Begum, A. Bagnall, A. Mueen, and G. Batista, “The ucr time series classification archive,” July 2015, www.cs.ucr.edu/~eamonn/time_series_data/.

[3] C. Zhang, X. Peng, C. Sha, K. Zhang, Z. Fu, X. Wu, Q. Lin, and D. Zhang, “Deeptralog: Trace-log combined microservice anomaly detection through graph-based deep learning,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 623–634.

[4] S. He, J. Zhu, P. He, and M. R. Lyu, “Experience report: System log analysis for anomaly detection,” in *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*. IEEE, 2016, pp. 207–218.

[5] M. Du, F. Li, G. Zheng, and V. Srikumar, “Deeplog: Anomaly detection and diagnosis from system logs through deep learning,” in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 1285–1298.

[6] J. Liu, J. Zhu, S. He, P. He, Z. Zheng, and M. R. Lyu, “Logzip: Extracting hidden structures via iterative clustering for log compression,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 863–873.

[7] R. Tian, Z. Diao, H. Jiang, and G. Xie, “Logdac: A universal efficient parser-based log compression approach,” in *ICC 2022-IEEE International Conference on Communications*. IEEE, 2022, pp. 3679–3684.

[8] S. Locke, H. Li, T.-H. P. Chen, W. Shang, and W. Liu, “Logassist: Assisting log analysis through log summarization,” *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3227–3241, 2021.

[9] H. Dai, H. Li, C.-S. Chen, W. Shang, and T.-H. Chen, “Logram: Efficient log parsing using n n-gram dictionaries,” *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 879–892, 2020.

[10] H. Dai, Y. Tang, H. Li, and W. Shang, “Pilar: Studying and mitigating the influence of configurations on log parsing,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 818–829.

[11] R. Vaarandi and M. Pihelgas, “Logcluster-a data clustering and pattern mining algorithm for event logs,” in *2015 11th International conference on network and service management (CNSM)*. IEEE, 2015, pp. 1–7.

[12] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, “Drain: An online log parsing approach with fixed depth tree,” in *2017 IEEE international conference on web services (ICWS)*. IEEE, 2017, pp. 33–40.

[13] M. Du and F. Li, “Spell: Streaming parsing of system event logs,” in *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 2016, pp. 859–864.

[14] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, “A lightweight algorithm for message type extraction in system application logs,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 11, pp. 1921–1936, 2011.

[15] V.-H. Le and H. Zhang, “Log parsing with prompt-based few-shot learning,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2438–2449.

[16] Y. Huo, Y. Su, C. Lee, and M. R. Lyu, “Semparser: A semantic parser for log analytics,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 881–893.

[17] J. Xu, R. Yang, Y. Huo, C. Zhang, and P. He, “Divlog: Log parsing with prompt enhanced in-context learning,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[18] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. 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*, 2019, pp. 807–817.

[19] X. Wang, X. Zhang, L. Li, S. He, H. Zhang, Y. Liu, L. Zheng, Y. Kang, Q. Lin, Y. Dang *et al.*, “Spine: a scalable log parser with feedback guidance,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1198–1208.

[20] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu, “A survey on automated log analysis for reliability engineering,” *ACM computing surveys (CSUR)*, vol. 54, no. 6, pp. 1–37, 2021.

[21] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, “Sage: practical and scalable ml-driven performance debugging in microservices,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 135–151.

[22] H. Wang, Z. Wu, H. Jiang, Y. Huang, J. Wang, S. Kopru, and T. Xie, “Groot: An event-graph-based approach for root cause analysis in industrial settings,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 419–429.

[23] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, “Sherlog: error diagnosis by connecting clues from run-time logs,” in *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, 2010, pp. 143–154.

[24] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, “Detecting large-scale system problems by mining console logs,” in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 117–132.

[25] Z. Li, C. Luo, T.-H. Chen, W. Shang, S. He, Q. Lin, and D. Zhang, “Did we miss something important? studying and exploring variable-aware log abstraction,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 830–842.

[26] A. Graves, A.-r. Mohamed, and G. Hinton, “Speech recognition with deep recurrent neural networks,” in *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee, 2013, pp. 6645–6649.

[27] S. C. Fanni, M. Febi, G. Aghakhanyan, and E. Neri, “Natural language processing,” in *Introduction to Artificial Intelligence*. Springer, 2023, pp. 87–99.

[28] N. Wies, Y. Levine, and A. Shashua, “The learnability of in-context learning,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[29] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[30] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, and Z. Sui, “A survey on in-context learning,” *arXiv preprint arXiv:2301.00234*, 2022.

[31] S. J. Reddi, S. Miryoosefi, S. Karp, S. Krishnan, S. Kale, S. Kim, and S. Kumar, “Efficient training of language models using few-shot learning,” in *International Conference on Machine Learning*. PMLR, 2023, pp. 14 553–14 568.

[32] Z. Wang, W. Zhong, Y. Wang, Q. Zhu, F. Mi, B. Wang, L. Shang, X. Jiang, and Q. Liu, “Data management for large language models: A survey,” *arXiv preprint arXiv:2312.01700*, 2023.

[33] G. A. Miller, “Wordnet: a lexical database for english,” *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[34] Z. A. Khan, D. Shin, D. Bianculli, and L. Briand, “Guidelines for assessing the accuracy of log message template identification techniques,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1095–1106.

[35] K. Shima, “Length matters: Clustering system log messages using length of words,” *arXiv preprint arXiv:1611.03213*, 2016.

[36] S. Kou, L. Hu, Z. He, Z. Deng, and H. Zhang, “Cllms: Consistency large language models,” *arXiv preprint arXiv:2403.00835*, 2024.

[37] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen, “Logmine: Fast pattern recognition for log analytics,” in *Proceedings of the 25th ACM international on conference on information and knowledge management*, 2016, pp. 1573–1582.

[38] L. Tang and T. Li, “Logtree: A framework for generating system events from raw textual logs,” in *2010 IEEE International Conference on Data Mining*. IEEE, 2010, pp. 491–500.

[39] J. Ya, T. Liu, H. Zhang, J. Shi, and L. Guo, “An automatic approach to extract the formats of network and security log messages,” in *MILCOM 2015-2015 IEEE Military Communications Conference*. IEEE, 2015, pp. 1542–1547.

[40] Y. Zhong, Y. Guo, and C. Liu, “Flp: a feature-based method for log parsing,” *Electronics letters*, vol. 54, no. 23, pp. 1334–1336, 2018.

[41] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” *arXiv preprint arXiv:1907.11692*, 2019.