LogBatcher^{+}: Towards Practical Log Parsing with Large Language Models

Yi Xiao, Van-Hoang Le, *Member, IEEE*, Hongyu Zhang, *Senior Member, IEEE*

***Abstract*—Log parsing, the process of converting raw log messages into structured formats, is an important initial step for automated analysis of logs of large-scale software systems. Traditional log parsers often rely on heuristics or handcrafted features, which may not generalize well across diverse log sources or require extensive model tuning. Recently, some log parsers have utilized powerful generative capabilities of large language models (LLMs). However, they heavily rely on demonstration examples, resulting in substantial overhead in LLM invocations. To address these issues, we propose LogBatcher^{+}, a cost-effective LLM-based log parser that requires no training process or labeled data. To leverage latent characteristics of log data and reduce the overhead, we construct a batch of logs and divide them into several partitions through clustering. Then we perform a cache matching process to match logs with previously parsed log templates. Finally, we provide LLMs with better prompt context specialized for log parsing by batching a group of logs from each partition and variables from the cache. We have conducted experiments on 14 public large-scale log datasets, and the results show that LogBatcher^{+} is effective and efficient for log parsing.**

***Index Terms*—Log Parsing, Batch Prompting, Large Language Models**

I. Introduction

Software-intensive systems such as cloud systems and online service systems often record runtime information by printing console logs. Software logs are semi-structured data printed by logging statements (e.g., `printf()`, `logInfo()`) in source code. The primary purpose of system logs is to record system states and important events at various critical points to help engineers better understand system behaviours and diagnose problems. The rich information included in log data enables a variety of software reliability management tasks, such as detecting system anomalies [1], [2], [3], ensuring application security [4], [5], [6], and diagnosing errors [7], [8], [9].

To facilitate various downstream analytics tasks, log parsing, which parses free-text into a structured format [10], is the first and foremost step. An accurate log parser is always in high demand for intelligent log analytics because it could simplify the process of downstream analytics tasks and allow more methods (e.g., Machine Learning and Deep Learning) to be applied [11]. Log parsing is the task of converting a raw log message into a specific log template associated with the corresponding parameters. As shown in Figure 1, each log message is printed by a logging statement in the source code and records a specific system event with its header and body. The header is determined by the logging framework and includes information such as component and verbosity level. The log message body (log message for short) typically consists of two parts: 1) *Template* - constant strings (or keywords) describing the system event; 2) *Parameters* - dynamic variables, which vary during runtime and reflect system runtime information. For example, in the log message in Figure 1, the header (i.e., "`17/08/22 15:50:46`", "`INFO`", and "`BlockManager`") can be easily distinguished through regular expressions. The log message consists of a template "`Failed to report <*> to master; giving up`" and a parameter "`rdd_5_1`". The log template typically contains constant strings, referring to commonalities across log data. The log parameters are dynamic variables, referring to variabilities that vary across log messages.

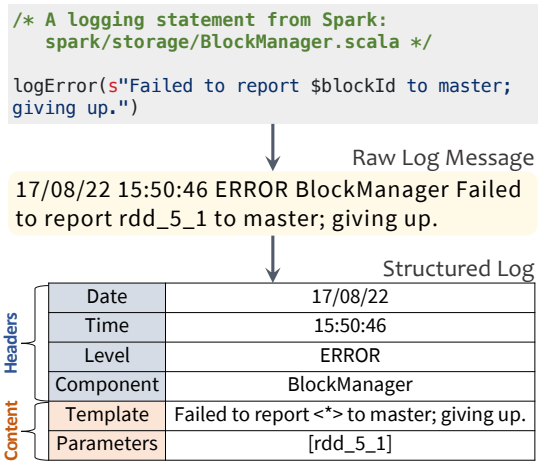

Fig. 1. An Illustration of Log Parsing

In recent years, there have been tremendous efforts towards achieving the goal of automated log parsing. Since the source code is generally inaccessible during system maintenance, existing log parsing methods propose to leverage *syntax* and *semantic* patterns of logs to identify and separate static text and dynamic variables. Syntax-based log parsers [12], [13], [14], [15] utilize specific features or heuristics (e.g., token count, frequency, and position) to extract the constant parts of log messages as templates. In contrast, semantic-based log parsers propose to recognize dynamic variables based on their semantic differences from constant keywords. Unfortunately, the performance of these log parsers in practice remains

Hongyu Zhang is the corresponding author.

Yi Xiao and Hongyu Zhang are with the School of Big Data and Software Engineering, Chongqing University, China (email: xiaoyi@stu.cqu.edu.cn, hyzhang@cqu.edu.cn).

Van-Hoang Le is with the School of Information and Physical Sciences, The University of Newcastle, Australia (email: hoang.le@newcastle.edu.au).

unsatisfactory [16], [17]. On the one hand, syntax-based log parsers heavily rely on crafted rules and domain knowledge, thus being ineffective when encountering previously unseen log patterns [18], [19]. On the other hand, semantic-based log parsers still require certain training overheads, such as training models from scratch or fine-tuning pre-trained language models with labeled data, which is scarce and costly to obtain [20].

To address these limitations, recent studies [20], [19], [21] propose to leverage the text understanding capacity of large language models (LLMs) for automated log parsing. Specifically, these studies adopt the in-context learning (ICL) prompting technique to adapt LLMs to the log parsing task. In ICL, a prompt consists of an instruction and associated demonstration examples. Despite the effectiveness, these LLM-based log parsers still fail to meet practical usage of log parsing due to the following reasons:

1) **Over reliance on demonstrations:** As LLMs are not explicitly specialized for log parsing, existing LLM-based log parsers require labeled demonstration examples (i.e., demonstrations) to construct in-context prompts. The performance of LLM-based log parsing has been shown to be sensitive to the quality and quantity of demonstrations [19], [20]. Furthermore, demonstrations can be quickly outdated as the volume and format of logs rapidly change [22], [1]. Hence, selecting demonstrations in in-context learning can be a delicate art and might require significant trial-and-errors.
2) **LLM invocation cost:** Log data is typically generated in a massive volume. Naively querying LLMs for each log message is impractical due to the substantial cost of invoking LLMs' service API. Furthermore, the cost incurred by the instruction and demonstrations in the prompts is not neglectable.
3) **Inefficient parsing performance:** While existing LLM-based log parsers have demonstrated effectiveness and been evaluated on large-scale datasets, their efficiency remains generally inferior to traditional parsing approaches. Although components such as adaptive random caching [19] and template memory [23] have been proposed to improve parsing efficiency, they still cannot achieve satisfactory results.

To address the aforementioned challenges, in this paper, we propose LogBatcher$^+$, a novel *training-free*, *demonstration-free*, and *cost-effective* LLM-based log parser. LogBatcher$^+$ leverages latent commonalities and variabilities of log data [24] to provide LLMs with better prompt context specialized for log parsing. Specifically, LogBatcher$^+$ first groups log data into several partitions using a versatile clustering algorithm. Then, for each partition, LogBatcher$^+$ samples log messages with high diversity to construct a batch of logs as the prompt to query LLMs to parse logs. By doing so, we can introduce variabilities within the prompt context to better guide LLMs to perform the log parsing task without the need for demonstrations. To further reduce the number of LLM invocations, LogBatcher$^+$ adopts a simple yet effective caching mechanism to store the intermediate results of LLMs and avoid redundant queries.

We have conducted a comprehensive evaluation on 14 public large-scale datasets [16]. The results show that LogBatcher$^+$ outperforms state-of-the-art baselines in terms of accuracy, efficiency, and LLM inference cost. It can achieve an average Group Accuracy [10] of 0.945 and Parsing Accuracy of 0.917, which are significantly higher than the best-performing supervised LLM-based log parser (i.e., LILAC [19]). Moreover, LogBatcher$^+$ is robust across diverse log datasets without the need for demonstrations, and can achieve an average speedup of 3.2$\times$ to 5.0$\times$ compared to other LLM-based log parsers.

The main contributions of this paper are as follows:

1) We propose LogBatcher$^+$, the first *demonstration-free* LLM-based log parsing framework to the best of our knowledge. Besides, LogBatcher$^+$ does not require any training overhead and is cost-effective for parsing large-scale log data.
2) We introduce a log-specific prompting strategy to provide LLMs with a batch of logs and historical variables, which allows LLMs to better incorporate the latent commonalities and variabilities among log messages. Furthermore, the token consumption of LLMs is reduced.
3) We propose a novel parsing cache mechanism that combines hash table with prefix tree, significantly enhancing caching efficiency. Experimental results on large-scale datasets demonstrate that LogBatcher$^+$ surpasses other parsers in terms of efficiency.
4) We conduct a comprehensive evaluation on the public Loghub-2.0 dataset [16]. Experimental results show that LogBatcher$^+$ outperforms state-of-the-art baselines in terms of both accuracy and efficiency.

This work is an extension of our ASE 2024 paper [25]. Compared to the preliminary version, we make several major enhancements: achieving online parsing by dynamically building a chunk of unmatched logs for parsing, improving the quality of the prompts by introducing variable-aware prompts, and improving the efficiency of the approach by introducing a more efficient adaptive cache. We perform a more comprehensive evaluation of LogBatcher$^+$'s effectiveness, robustness, and efficiency on 14 large-scale datasets and across a broader set of baselines.

II. Background and Related Work

A. Log Parsing

Log parsing is one of the first steps for log analysis tasks [10]. It is a process to extract the static log template parts and the corresponding dynamic parameters (or variables) from free-text raw log messages. A straightforward method of log parsing involves matching raw log messages with logging statements within the source code [26], [27] or designing handcrafted regular expressions to extract log templates and parameters [10]. However, these approaches are impractical due to the inaccessibility of the source code (especially for third-party libraries [10]) and the huge volume of logs. To achieve the goal of automated log parsing, many *syntax*-based and *semantic*-based approaches have been proposed to identify log templates as the frequent part of log messages.

Syntax-based log parsers [12], [28], [14], [29] assume that log templates inherit some common patterns which emerge constantly across the entire log dataset. Some parsers [14], [30], [31] extract log templates by identifying the constant parts of log messages through the mining of frequent patterns, for example, Logram [14] finds frequent n-gram patterns which emerge constantly across the entire log dataset as templates. Logs that belong to the same template exhibit similarities. Consequently, some methods [32], [33], [34] employ clustering techniques to group logs and extract the constant portions of log messages for log parsing. Heuristics-based log parsers [29], [12], [28] leverage unique characteristics from log messages to extract common templates efficiently. For example, AEL [29] employs a list of heuristic rules to extract common templates. Drain [12] employs a fixed-depth tree structure to assist in dividing logs into different groups, assuming that all log parameters within specific templates possess an identical number of tokens, while Brain [28] updated Drain by using a bidirectional parallel tree.

Semantic-based log parsers leverage semantic differences between keywords and parameters to formulate log parsing as a token classification task. For example, UniParser [35] unifies log parsing for heterogeneous log data by training with labeled data from multiple log sources to capture common patterns of templates and parameters. LogPPT [18] introduces a novel paradigm for log parsing, employing template-free prompt-tuning to fine-tune the pre-trained language model, RoBERTa. Although effective, existing semantic-based log parsers require certain training overheads, such as training models from scratch or fine-tuning pre-trained language models with labeled data, which is scarce and costly to obtain [20].

Recently, some studies have proposed to utilize large language models (LLMs) owing to their extensive pre-trained knowledge. These studies have achieved promising results in log parsing [20], [19], [21], [23] thanks to the strong in-context learning capability of LLMs. In the following sections, we will introduce some recent LLM-based log parsers and discuss their limitations.

B. Log Parsing with Large Language Models

Large language models (LLMs) have achieved remarkable success in various natural language processing [36], [37] and computer vision tasks [38], [39]. In-context learning is a promising prompt engineering method for adopting LLMs without fine-tuning them [40]. In-context learning typically requires an *instruction* that describes the task and *demonstrations* that provide several examples of how to solve the task. Recent studies have demonstrated that in-context learning can aid LLMs in achieving remarkable performance in a variety of tasks [41], [42], [43].

Le and Zhang [20] validated the potential of LLMs in log parsing and obtained promising results. DivLog [21], LILAC [19] and LogParser-LLM [44] enhance the performance of large models by selecting demonstrations from labeled log data and utilizing the in-context learning capabilities of LLMs. They employ different methods to sample a labeled candidate log set. These methods are sensitive to the quantity and coverage of labeled logs and incur LLM inference overhead. Lemur [45] invokes LLM to merge generated similar templates, improving the accuracy of log parsing groupings. However, it requires extensive hyperparameter tuning for specific datasets. LibreLog [23] improves parsing accuracy by incorporating LLM on the basis of the grouping results obtained from a syntax-based Log parser(i.e., Drain [12]). However, it cannot achieve online parsing and faces challenges with lower efficiency. It has been found that these LLM-based log parsers have outperformed semantic-based log parsers (e.g., LogPPT [18] and UniParser [35]) in terms of parsing accuracy [19], [21].

Despite promising results, LLM-based log parsing can be costly in terms of token usage, especially when large volumes of LLM calls are needed. The costs of one LLM invocation scale linearly with the number of tokens, including both the input prompt tokens (instruction and demonstrations). Consequently, managing LLM invocation cost is vital for practical applications. Since LLM infrastructure/services can change over time, recent studies [46], [47] measure and reduce token consumption as the primary metric for LLM cost management. Similarly, in this paper, we focus on accomplishing more data processing with fewer tokens and LLMs calls to achieve cost-effective log parsing.

III. A Motivating Example

Recently, several studies [20], [19], [21] have proposed to utilize LLMs for log parsing and achieved promising results. Still, these studies fail to achieve satisfactory performance in practice. We have identified two major limitations of existing LLM-based log parsing approaches, which prevent their practical usage.

[Instruction] I want you to act like an expert in log parsing. I will give you a log message wrapped by backticks. Your task is to identify all the dynamic variables in logs, replace them with {variables}, and output a static log template. Please print the input log's template wrapped by backticks.
[Query] Log message: `Created local directory at /opt/hdfs/nodemanager/usercache/curi/appcache/application_1485248649253_0147/blockmgr-70293f72-844a-4b39-9ad6-fb0ad7e364e4`
[Demo 1]
Log message: `Starting executor ID 5 on host mesos-slave-07`
Log template: `Starting executor ID {variables} on host {variables}`
[Demo 2]
Log message: `Connecting to driver: spark://CoarseGrainedScheduler@10.10.34.11:48636`
Log template: `Connecting to driver: spark://{variables}`

Input	Output
[Instruction] [Query]	Created local directory at {directory_path} ✓
[Instruction] [Demo 1] [Query]	Created local directory at {variables}/blockmgr-{variables} ✗
[Instruction] [Demo 2] [Query]	Created local directory at {variables} ✓
[Instruction] [Demo 1] [Demo 2] [Query]	Created local directory at {variables}/blockmgr-{variables} ✗

Fig. 2. Selecting in-context demonstrations for log parsing on Spark (*Results are produced using gpt-3.5-turbo [48] with instruction and demonstrations adopted from [19]*)

Over reliance on demonstrations. Although LLMs are equipped with a huge amount of pre-trained knowledge, they

are not specialized in the log parsing task. Directly querying LLMs for log parsing could result in unsatisfactory performance [19], [20]. Hence, to overcome this problem, recent studies [21], [19] straightforwardly leverage the in-context learning prompting technique to impart log-specific knowledge to LLMs via labeled demonstrations. However, selecting even a few useful demonstrations can quickly become more laborious as the volume and format of logs rapidly change [22], [1]. More importantly, selecting in-context demonstrations can be challenging as the quality of these demonstrations directly affects LLM-based log parsing. Figure 2 illustrates the impact of four different demonstrations on the parsing performance. In this example, we set *temperature* to 0 to avoid bias from LLM randomness. Sample inputs and outputs shown from top to bottom (Spark log) are: (1) zero-shot without demonstration: correct answer; (2) a correct but noisy demonstration (Demo 1), which leads to a wrong answer; (3) a correct demonstration (Demo 2), which leads to a correct answer; and (4) combining Demo 1 and Demo 2 again leads to an incorrect answer. This issue highlights the sensitivity of demonstrations to the performance of LLM-based log parsing. To quantitatively understand the impact of labeled demonstrations on parsing performance of LLMs, we vary the number of demonstrations from 32 to 0 and evaluate the parsing accuracy of the state-of-the-art LLM-based log parser, LILAC [19]. As shown in Figure 3, regarding widely-used group accuracy (GA) [10] and message-level accuracy (MLA) metrics [35], LILAC witnesses a significant drop in performance when the number of demonstrations decreases. Specifically, its performance declines 15% and 20% in GA and MLA, respectively, when the number of demonstrations decreases from 32 to 0.

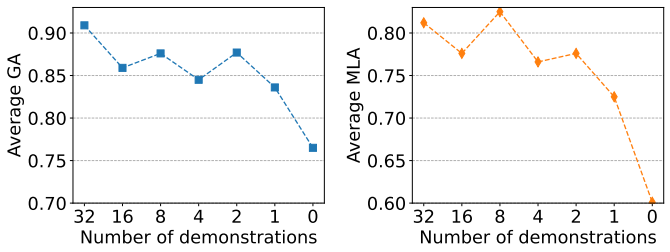

Fig. 3. Impact of different numbers of demonstrations

LLM invocation cost. Log data can be generated in a massive volume in production. For example, Mi et al. [49] reported that the Alibaba cloud system produces about 30-50 gigabytes (around 100-200 million lines) of tracing logs per hour. Naively querying LLMs for each log message is impractical due to the substantial cost of inference. As illustrated in Figure 2, querying GPT-3.5-Turbo [48] with a prompt consisting of one instruction (66 tokens in total[1]) and one log message (55 tokens in total) will cost $(55 + 66) \times 100{,}000{,}000 \times (0.50/1{,}000{,}000) = \$6{,}050$ because the price of GPT-3.5-Turbo API services is \$0.5 per 1M tokens [50]. Due to the large amount of log messages, the cost for LLM-based log parsing could pose a significant financial burden in practice.

[1]We compute the number of tokens using the OpenAI's *tiktoken* package: https://github.com/openai/tiktoken

Note that the cost incurred by the instruction and demonstrations in the prompt is not neglectable. For example, the token count of the prompt instruction in Figure 2 is 66, which is more than the token count of the log message (55). Considering adding more demonstrations (36 tokens per demonstration in Figure 2) to the prompt to improve the parsing performance, the token count of the prompt will increase linearly with the number of demonstrations. This will further increase the cost of LLM invocation, making it even more expensive to query LLMs for log parsing.

Inefficient parsing performance. Some LLM-based log parsers (such as DivLog) query an LLM for each and every log message, which is impractical for large-scale datasets. Some studies [45], [23] suggest clustering the entire dataset for further parsing, which also face efficiency challenges. Recent LLM-based log parsers (like LogBatcher, LILAC, and LogParser-LLM) have started to employ various data structures (e.g., list in LogBatcher and prefix tree in LILAC) for caching to enhance the efficiency of LLM-based parsers. Although promising parsing efficiency has been achieved, the average parsing time remains 2.5× to 3.8× slower compared to syntax-based log parsers (e.g., Drain) [19], [23].

IV. Methodology

Drawing upon the observations described in Section III, we propose LogBatcher$^+$, a novel *demonstration-free*, *training-free*, and *cost-effective* LLM-based log parser. The main idea behind LogBatcher$^+$ is that log data possesses latent characteristics, i.e., commonality and variability, which allows LLMs to perform log parsing without demonstrations. Specifically, as the goal of log parsing is to recognize the dynamic variables (i.e., variability) from static patterns (i.e., commonality), we use a batch of log messages as the input to LLMs instead of using a single log message. In this way, we can incorporate commonalities and variabilities among log messages into the input of LLM, thus allowing LLMs to better correlate the log parsing with the log data itself without the need of labeled demonstration examples.

An overview of LogBatcher$^+$framework is illustrated in Figure 4. Since raw log data are massively generated in the production environment [49], [51] and current systems highly demand online log parsing [13], [52], We perform log parsing on log chunks instead of individual log message. Specifically, for each incoming log message, we utilize the caching mechanism described below for matching. Simultaneously, we aggregate a certain amount of unmatched logs into a chunk, which then enters the parsing phase collectively. Each log chunk goes through three main components: ① *Partitioning*: separating each log chunk into several partitions using a versatile clustering algorithm. ② *Caching*: performing a cache matching process for logs in each partition to match them with previously parsed log templates to avoid duplicate LLM queries and improve parsing efficiency. ③ *Batching – Querying*: sampling a diverse set of logs from each partition to form a batch, which is then sent to the LLM for parsing. Finally, we refine the identified templates and match the logs with the templates to mitigate the impact of clustering errors.

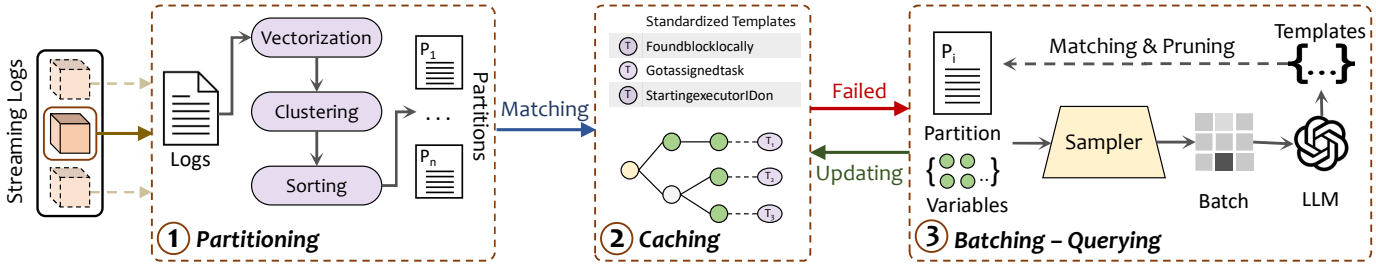

Fig. 4. An overview of LogBatcher$^+$

A. Partitioning

The aim of this phase is to ensure that logs allocated to the same partitions share some commonalities. This is crucial for the subsequent in-context learning process, as it allows LLMs to learn the commonalities within log data and associate them with the log parsing task. We employ a versatile clustering algorithm based on DBSCAN [53] to partition logs. Figure 5 illustrates the log partitioning process.

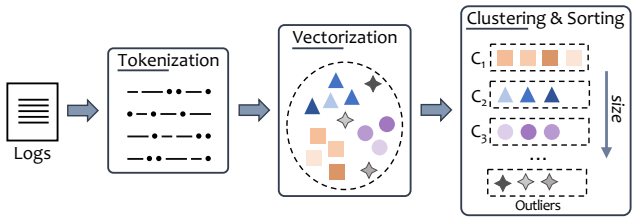

Fig. 5. Log partitioning through clustering

1) Tokenization: The initial step of *partitioning* involves log tokenization and cleaning, which are crucial for accurate clustering. First, we use general delimiters (i.e., *white space*) to perform initial tokenization of the logs. Considering that logs have some unique delimiters due to their relevance to the code, we define specific rules to further refine the tokenization for each token. Finally, we clean the logs by masking potential variable tokens. We utilize some basic regular expressions to refine the tokenization. For example, the symbol "=" can serve as a delimiter in logs such as "`START: tftp pid=16563 from=10.100.4.251`". However, if "=" appears within a URL, as in "`after trim url = https://www.google.com/search?q=test`", it disrupts the integrity of the variable, leading to clustering errors. After that, we improve the clustering performance by masking tokens that resemble parameters such as numbers, IP addresses, and URLs. Given a batch of logs $\mathrm{L} = \{L_1, L_2, \ldots, L_n\}$, each log L_i is tokenized into a set of tokens $\{t_{i1}, t_{i2}, \ldots, t_{im}\}$.

2) Vectorization: Vectorization is a prerequisite for clustering as it transforms log data into a numerical format, which is suitable for clustering algorithms. Since different tokens in logs are of varying importance [1], we adopt the Frequency-Inverse Document Frequency (TF-IDF) [54] to vectorize the log. Specifically, we first calculate the *Term Frequency* (TF) to describe the importance of a token in a log message, where $TF(token) = \frac{\#token}{\#total}$, $\#token$ is the number of target token in a log message, $\#total$ is the number of all tokens in a log message. On the other hand, if a token appears in many logs, it is less informative and becomes too common to be able to distinguish distinct log messages. Therefore, we calculate the *Inverse Document Frequency* (IDF) to reduce the weight of overly common tokens, where $IDF(token) = \log(\frac{\#L}{\#L_{token}})$, $\#L$ is the total number of logs, $\#L_{token}$ is the number of logs containing the target token. For each word, its TF-IDF weight w is calculated by $TF \times IDF$.

Finally, we can obtain the vector representation $\mathbf{V}_L \in \mathbb{R}^d$ of each log message by summing up the token vectors L with their corresponding TF-IDF weights, according to Equation 1:

$$\mathbf{V}_L = \frac{1}{N} \sum_{i=1}^{N} w_i \ . \ v_i \tag{1}$$

3) Clustering & Sorting: LogBatcher$^+$ adopts the DBSCAN algorithm (Density-Based Spatial Clustering of Applications with Noise) [53] to cluster log messages in a chunk into different groups, each of which is more likely to contain the log messages with similar semantics. DBSCAN groups together data points that are closely packed, marking as outliers points that lie alone in low-density regions. The reasons we choose DBSCAN are threefold: (1) it does not require specifying the number of clusters in advance, which is more practical in the log parsing task; (2) it has been demonstrated to be more effective and efficient and has been widely used in many domains [55]; (3) it has a small number of hyperparameters and is less sensitive to hyperparameter selection, thus being easy-to-use in practice. After clustering, we sort the clusters by size in descending order and consider all outliers as a separate cluster to process at last. The reason is that smaller clusters are more likely to contain logs with unique characteristics (e.g., noises) that are difficult to be parsed. By processing them at last, we can leverage previously parsed templates stored in the cache to filter out the noise and improve parsing performance.

B. Caching

Parsing all arriving logs with LLMs is impractical due to the high API cost and latency, especially when logs are generated in large quantities. To address this issue and improve parsing efficiency, we adopt a combination of a prefix tree and a hash table to implement an adaptive caching mechanism, which stores previously parsed templates and matches them with unseen logs. Specifically, before parsing, we filter out logs that can be matched with the cache. For those unmatched logs, we

process them with LLMs, adding the newly generated template to the cache.

1) Adaptive Caching: We detail the usage of the two data structures in our approach as follows:

- *Prefix tree* Following previous work [12], [19], [44], we first use a prefix tree structure to dynamically store tokenized log templates generated by the LLM, serving as a parsing cache. As shown in figure 6, each intermediate node represents a specific token (including "<*>" as a wildcard for variable content) and leaf nodes correspond to unique log templates formed by concatenating tokens along the root-to-leaf path. The tree enables single-pass template retrieval through path traversal, eliminating the need for sequential comparisons. Templates sharing common prefixes are clustered in subtrees, reflecting their structural similarity and facilitating targeted cache updates.
- *Hash table* As the number of log templates grows, the prefix tree suffers from efficiency degradation in cache matching. To maintain the efficiency of the caching, we employ a hash table as an integral component of our hybrid caching architecture. When new log templates are generated, each template undergoes a standardization process (e.g., removing all "<*>" wildcards, numbers, and whitespaces). The processed template string is then hashed to generate fingerprints as keys. The hash table establishes direct key-value mappings between template fingerprints and their complete structures.

2) Caching Matching: Since the insertion and lookup operations in the hash table exhibit constant-time complexity (O(1)), given a new log message, LogBatcher$^+$ first attempts to check whether the target template has already been parsed and stored in the parsing cache. Specifically, the log message is standardized and hashed to generate a fingerprint as a key. This key is then queried in the hash table for immediate template matching. If no match is found in the hash table, the search continues in the prefix tree. The log message is tokenized and traverses the prefix tree through depth-first search: fixed tokens follow exact node matches while "<*>" wildcards recursively consume tokens until subsequent matches align with tree paths. The successful traversal to a leaf node retrieves the cached template, while unmatched logs will trigger the parsing process of LogBatcher$^+$ to generate new templates for them.

3) Caching Updating: During the cache matching process, three cases may happen: (1) The log directly finds its corresponding template in the hash table. (2) The log fails to match in both structures. (3) The log fails to find a matching template in the hash table but succeeds through a prefix tree search. Case 1 indicates the log can quickly match a template without cache updating.

For Case 2, where no matching template exists in the cache, we query the LLM to obtain the correct template. The template is then updated into the cache through dual pathways: Standardized templates are first inserted into the hash table via hashing. Tokenized versions of these templates are subsequently added to the prefix tree. During the insertion of newly generated templates into the prefix tree, their associated variable slots (marked by "<*>" wildcards) are structurally

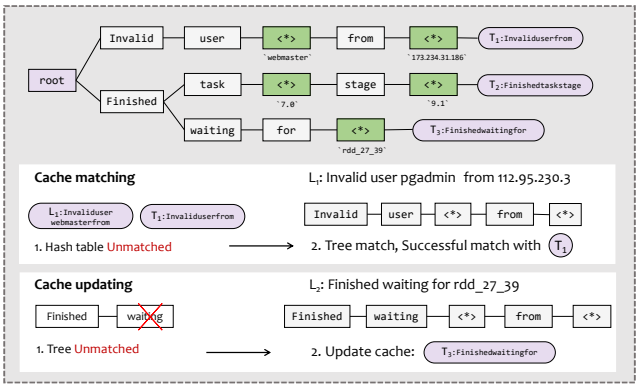

Fig. 6. Demonstration of caching mechanism

preserved within corresponding tree nodes. Specifically, this involves replacing "<*>" in the log templates with the generic matching symbol "`(.?)`", allowing regular expressions to return all the corresponding variables. As shown in IV-C3, these identified variables will be used to enhance the parsing performance of LLM.

For Case 3, we additionally add the log itself to the hash table. If the standardized form of a log is inconsistent with that of its corresponding template, the log might match only in the prefix tree. For example, the log "`Reopen Block blk_-7008279672769077211`" is standardized to "`ReopenBlockblk`" while its template "`Reopen Block <*>`" is standardized to "`ReopenBlock`". The system-specific identifier "`blk`" in the variable "`blk_-7008279672769077211`" prevents direct matching. This mitigates the over-masking of variables. By updating logs under such a case into the hash tables, the probability of directly matching logs within the hash table will be significantly enhanced. For instance, if the standardized form "`ReopenBlockblk`" from log "`blk_-7008279672769077211`" is stored in the cache, logs sharing the same template will directly find matches in the hase table, enabling adaptive caching for system-specific variables and resolving mismatches caused by excessive standardization.

C. Batching – Querying

Logs that belong to the same template not only share frequently occurring tokens but also exhibit rich variability in their dynamic parts. These characteristics of log data are widely observed in practice, and are adopted by many data-driven log parsing methods [12], [34]. Recent LLM-based log parsers, however, overlook these characteristics, leading to the overly sensitive nature of LLMs to demonstrations. To address this issue, we propose a batching - querying approach to provide LLMs with commonalities and variabilities within input logs for demonstration-free log parsing.

1) Batching: After partitioning, logs in each partition already exhibit commonalities in their semantics and syntax. We sample a set of logs from each partition to form an input batch for LLMs. To this end, we adopt a diversity-based sampling method to select logs that maximize the sample diversity.

Specifically, we calculate the cosine similarity between every two logs based on their TF-IDF vectors, forming a similarity matrix. We then use the Determinantal Point Process (DPP) algorithm [56] to select logs that maximize the sample diversity. By doing so, we can ensure that the input batch contains both commonalities (introducing by clustering-based partitioning) and variabilities (introducing by diversity-based sampling) within the input logs, which can help LLMs better associate the task description with the input logs and improve parsing accuracy.

2) Variables Selection: After constructing the log batch, we select diverse variable samples from the cached historical variables as in-context references. Specifically, we first collect all variables from the cache (e.g., parsed variables such as "`rdd_27_39`"). We then adopt the same diversity-based sampling strategy as used in the Batching phase. Variables from the same system can enhance LLMs' ability to accurately infer variable types of the queried log batch.

3) Prompting Design: Following previous work [20], [19], we design and use the prompt format, as shown in Figure 7. A common in-context learning paradigm consists of three parts: instruction, demonstration and query. However, different from them, we provide LLMs with historical variables and the input in the form of a batch as follows:

- *Instruction*: To provide the LLM with task-specific information, we briefly describe the goal of log parsing, and the formats of input and output. Moreover, we emphasize the main objective of log parsing as abstracting the variables as well as indicate that constant text should not be identified as variables to avoid over-parsing, where the LLM tries to find variables in every log. Existing research [57], [58] highlighted the benefits of classifying specific variable types. By specifying possible variable types in our prompts, LLM will be more stable when categorizing variables.
- *Historical variables*: To enhance the ability of LLM to identify variable parts, we provide the LLM with a batch of historical variables as a reference. As shown in Figure 6, the "`rdd_27_39`" in cached template "`Finished waiting for <*>`" is part of the parsing cache. Variables from the same system exhibit certain commonalities, which serve as a reference for LLM and help identify potential variables.
- *Queried Log Batch*: We provide the LLM with a batch of logs as input, separated by the newline character. This batch is sampled from the partitioned logs, which contain both commonalities and variabilities within the input logs. Hence, the input is well-related to the instruction, which can help LLMs better understand the log parsing task.

4) Post-Processing: The output from an LLM may contain redundant information beyond the desired template. With the locator "`` ` ``" and placeholder "`{placeholder}`", we can easily filter the raw output from LLM and get the identified template. To make the style of labeling the same for every system, Khan et al. [59] customized some heuristic rules, to correct the identified template. Some related works [19], [18] also adopt these rules to refine the generated templates and minimize the impact of inconsistent labels. We adopt and optimize this post-process. For example, [59] only considers

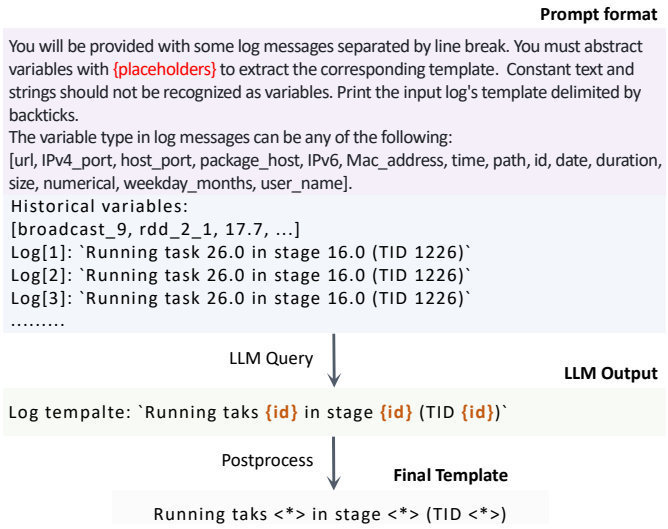

Fig. 7. An illustration of our prompt design

decimal numbers in the logs as variables, but in reality, hexadecimal numbers appear just as frequently.

5) Matching & Pruning: For the results of clustering, two common issues usually arise: logs that belong to the same template are grouped into different clusters, and logs that belong to different clusters are mistakenly grouped into the same cluster. The methods mentioned earlier effectively solve the first problem. For the second problem, we use the matching & pruning method. Pruning is essentially a re-group process using the identified template. As mentioned in Section IV-B, the identified template can be matched with logs through transformation with regular expressions. In most cases, the resulting template can match all logs in the cluster. When not all logs can be matched, indicating the second issue mentioned earlier in clustering, we consider the template as valid for the logs it can match. The unmatched logs are then pruned and sent to a new cluster, which will reenter the queue sequence for further parsing. Even though logs may be misclassified into the same cluster and trigger an invocation, we still make good use of this invocation, avoiding additional overhead. Algorithm 1 shows this process.

Algorithm 1: Matching & Pruning

Input: $\mathcal{C}$: Parsed Cluster
T: Identified Template
Output: $\mathcal{C}_{new}$: New Cluster

```
regex ← convertToRegex(T)
C_new ← ∅
foreach log ∈ C do
    if match(log, regex) then
        C ← C \ {log}
        C_new ← C_new ∪ {log}
    end
end
return C_new
```

After this batching-querying process, we obtain a log template that can match all or partial (if pruning is needed) logs in each partition. We repeat this process for every partition

that cannot find corresponding template, until all partitions are successfully parsed.

V. Experimental Design

A. Research Questions

We evaluate our approach by answering the following research questions:

RQ1. How does LogBatcher$^+$ perform compared to the baselines? In this RQ, we aim to comprehensively evaluate the performance of our proposed method. Specially, we compare our method with two state-of-the-art unsupervised data-driven log parsers (i.e., Drain [12], AEL [29]), two semantic-based log parsers (i.e., LogPPT [18], UniParser [35]) and two LLM-based log parsers (i.e., LILAC [19] and LibreLog [23]). We adopt the implementation of these methods from their public replication packages [10], [23], [19]. LILAC and LibreLog are recently proposed to leverage the in-context learning capacity of LLMs for log parsing. For a fair comparison, we select the same LLM (i.e., GPT-4o-mini) to reproduce the results of LILAC as used in LogBatcher$^+$. Since LibreLog is only designed for local deployment of open-source LLMs, we select its default LLM (i.e., Llama3-8B) to reproduce the results of LibreLog.

RQ2. How do different modules contribute to LogBatcher$^+$? LogBatcher$^+$ consists of three main components: Partitioning, Caching, and Batching. We evaluate the importance of each component by removing each of them from the framework and evaluating the performance. It is worth noting that performing online parsing on large-scale logs with LLMs would be impractical without caching. Therefore, we perform the ablation study with the following settings: (1) w/o$_{\text{partitioning}}$: we divide logs into several partitions using time windows; (2) w/o$_{\text{batching}}$: we only use one log entry as the LLM input.

RQ3. How does LogBatcher$^+$ perform with different prompting designs? While a common in-context learning paradigm consists of instruction, demonstration, and query, LogBatcher$^+$ employs a more log-specific prompting strategy. The prompts incorporate variable types, historical variables, and a batch of logs to help LLMs identify variables in the log more precisely. In this RQ, we will evaluate LogBatcher$^+$ with two primary dimensions of variation: (1) the size of log batch in the prompt, and (2) the inclusion of variable-aware components (including variable types and historical variables). We further analyze how our prompt design affects LLM invocation cost.

RQ4. How do different settings affect LogBatcher$^+$? To delve deeper into the effectiveness and robustness of LogBatcher$^+$, we explore how different settings of its major components affect the overall performance. Specifically, we use different clustering methods for partitioning, different sampling methods for batching, different chunk sizes, and different LLMs to evaluate the performance of LogBatcher$^+$.

B. Datasets

We conduct experiments on 16 public log datasets (Loghub-2k [60]) originated from the LogPai project [10]. These datasets cover logs from distributed systems, standalone software, supercomputers, PC operating systems, mobile systems, microservices, etc. Zhu et al. [10] sampled 2,000 log entries from each system in the dataset and manually labeled them. However, it has been observed that the original labels have some errors due to inconsistent labeling styles [59], [60], [16]. Therefore, following existing work [18], [21], [61], we use the version of the datasets corrected by Khan et al. [59]. Furthermore, as the Proxifier dataset has many different versions, we calibrated some labels according to the guidelines proposed by [16]. The datasets we used in our experiments are publicly available at our webpage [62].

C. Evaluation Metrics

Following recent studies [12], [16], [35], [59], we use four main metrics for evaluation, including:

Group Accuracy (GA): Group Accuracy [12] is the most commonly used metric for log parsing. The GA metric is defined as the ratio of "correctly parsed" log messages over the total number of log messages, where a log message is considered "correctly parsed" if and only if it is grouped with other log messages consistent with the ground truth.

Parsing Accuracy (PA): Parsing Accuracy [35] is defined as the ratio of "correctly parsed" log messages over the total number of log messages, where a log message is considered to be "correctly parsed" if and only if every token of the log message is correctly identified as template or parameter.

F1 score of Group Accuracy (FGA): FGA [16] metric is a template-level metric which measures the ratio of "correctly grouped" templates. It is computed as the harmonic mean of precision and recall of grouping accuracy, where the template is considered as correct if log messages of the predicted template have the same group of log messages as the oracle template.

F1 score of Template Accuracy (FTA): FTA [59] metric is also a template-level metric measures the ratio of "correctly identified" templates. It is computed as the harmonic mean of precision and recall of template accuracy, where the template is considered as correct if and only if it satisfies two requirements: log messages of the predicted template have the same group of log messages as the oracle, and all tokens of the template are the same as the oracle template.

In addition, to assess the LLM invocation cost of our proposed approach, we follow recent studies [46], [47] to measure the token consumption because the price structure of commercial LLM providers is related to the number of input tokens. Specifically, we use two metrics to measure token consumption, including (1) $\mathbf{T}_{\text{total}}$: the total number of tokens consumed for all invocations when parsing a dataset and (2) $\mathbf{T}_{\text{invoc}}$: the average number of tokens consumed per invocation.

D. Baselines

Based on the recent benchmark studies [16], [59], we select two state-of-the-art unsupervised data-driven log parsers (i.e., Drain [12], AEL [29]), two state-of-the-art semantic-based supervised log parsers (i.e., LogPPT [18] and UniParser [35]) and two recent LLM-based log parsers (i.e., LILAC [19] and

Librelog [23]). We adopt the latest version of these methods from their publicly available replication packages [10], [21], [19]. To ensure a fair comparison, we replicate the results of LILAC using the same LLM (i.e., GPT-4o-mini) as in LogBatcher$^+$. Furthermore, we replicate the results of LibreLog using its default LLM (e.g., Llama3-8B) since it is designed for Open-Srouce LLMs. We did not include other LLM-based log parsers (such as LogParser-LLM [44] and AdaParser [63]) in our comparison because their code is not publicly available.

E. Implementation and Settings

Implementation. In our experiments, we set the default LLM to *GPT-4o-mini* (version 0718[2]), which is the most cost-efficient model from OpenAI [64], [65]. We conduct our experiments on an Ubuntu 20.04 LTS server with an NVIDIA Tesla A800 GPU and Intel Xeon Gold 6342 CPU, using Python 3.9. We invoke the LLM through the Python library provided by OpenAI.

Settings. We adopt the implementation of DBSCAN provided by sklearn [66] for the Partitioning component. We set the hyperparameters of DBSCAN as follows: *epsilon* = 0.5 and *min_samples* = 5. For the Batching component, we set the batch size to 10. To avoid the randomness of the LLM, we set the temperature to 0. Furthermore, following previous work [16], [59], [19], we repeat all experiments 5 times and report the average results.

VI. Results and Analysis

A. ***RQ1:*** *How does LogBatcher$^+$ perform compared to the baselines?*

This RQ evaluates the performance of LogBatcher$^+$ from three aspects: effectiveness, robustness, and efficiency.

1) Effectiveness: Table I provides a comparative analysis of various log parsing methods across multiple datasets in terms of Group Accuracy (GA), Parsing Accuracy (PA), F1 score of Group Accuracy (FGA) and F1 score of Template Accuracy (FTA). For each dataset, the highest accuracy of each metric is highlighted in **bold**.

Experimental results show that LogBatcher$^+$ significantly outperforms the unsupervised data-driven log parsers and supervised semantic-based log parsers, including Drain [12], AEL [29], LogPPT [18], and UniParser [35]. Specifically, LogBatcher$^+$ exceeds the highest GA, FGA, PA and FTA of these methods by 10.2%, 34.8%, 19.1%, and 33.3% on average, respectively. Compared to LLM-based log parsers, i.e., supervised LILAC [19] and unsupervised LibreLog [23], LogBatcher$^+$ also achieves superior performance. Specifically, LogBatcher$^+$ outperforms unsupervised LibreLog in terms of all four metrics. For example, it achieves better FGA on 12 datasets and better FTA on 13 datasets. Compared to Supervised LILAC, the top-performing LLM-based log parser, LogBatcher$^+$ still outperforms by 5.4%, 1.5%, 8.0%, and 4.9% in terms of average GA, FGA, PA and FTA. It is worth noting that without any labeled demonstrations, LogBatcher$^+$ can still achieve the best average results in terms of all metrics. Overall, the experimental results confirm that LogBatcher$^+$ is effective for the log parsing task.

2) Robustness: LogBatcher aims to support a wide range of log data from various systems, as a universal log parser in a production environment demands strong performance and generalization capabilities [10]. Hence, we analyze and compare the robustness against different types of logs of LogBatcher$^+$ with that of the baselines by drawing a box plot to illustrate the accuracy distribution of each log parser's metrics across all datasets. Figure 8 shows the results. LogBatcher$^+$ consistently achieves the lowest standard deviation across all four metrics, indicating the stable performance of LogBatcher$^+$. Meanwhile, it achieves the highest median in terms of GA, PA, and FTA. Specifically, it yields a median of 0.97 for GA robustness, 0.96 for PA robustness and 0.79 for FTA robustness, which are better to other baselines. In terms of FGA, it yields a median of 0.91, only 0.7% lower than LILAC. Additionally, LogBatcher$^+$'s performance exhibits significantly fewer outliers compared to other baseline methods. This indicates that even in less typical scenarios, LogBatcher$^+$ can still achieve stable and reliable results. Overall, the experimental results demonstrate that LogBatcher$^+$ is robust and can be applied to various log datasets effectively.

3) Efficiency: Logs are persistently and frequently generated [49], which demand high parsing efficiency of log parsers. However, for semantic-based parsers and LLM-based parsers, efficiency has always been a weakness, limiting their practicality. LogBatcher$^+$ does not require any training process or labeled data, thus eliminating the time needed before parsing, whereas other parsers require time to train models (e.g., LogPT and UniParser), sample and label data (e.g., LILAC), or pre-group the entire dataset (e.g., LibreLog). In this experiment, we evaluate the efficiency of LogBatcher$^+$ and the baselines in terms of parsing time. Figure 9 shows the average parsing time across 14 datasets, with an average of 3.6 million log messages per dataset. The results show that LogBatcher$^+$ takes an average of 329.1 seconds to process 3.6 million logs, significantly outperforming semantic-based parsers. Additionally, LogBatcher$^+$ is 3.2$\times$ to 5.0$\times$ times faster than current LLM-based methods, LILAC and LibreLog. Due to its unique caching design, which shifts more matching workload to the hash table,LogBatcher$^+$ significantly reduces the overhead of cache matching (its cache matching process is 13.6$\times$ faster than LILAC). At the same time, LogBatcher$^+$ even surpasses syntax-based parsers (e.g., Drain), requiring only 75% of the time taken by Drain to process all logs. The improvement in parsing time demonstrates that LogBatcher$^+$ is efficient for the log parsing task.

Overall, our experimental results show that, being an unsupervised log parser, LogBatcher$^+$ can achieve better or comparable results to the current SOTA supervised method, LILAC. Also, it outperforms all unsupervised baselines, in terms of effectiveness, robustness, and efficiency.

[2]https://platform.openai.com/docs/models/gpt-4o-mini

TABLE I
COMPARISON WITH THE STATE-OF-THE-ART LOG PARSERS

	AEL				Drain				LogPPT				UniParser				LILAC				LibreLog				LogBatcher$^+$			
	GA	FGA	PA	FTA	GA	FGA	PA	FTA	GA	FGA	PA	FTA	GA	FGA	PA	FTA	GA	FGA	PA	FTA	GA	FGA	PA	FTA	GA	FGA	PA	FTA
Proxifier	0.974	0.667	0.677	0.417	0.692	0.206	0.688	0.176	0.509	0.286	0.634	0.457	0.989	0.870	**1.000**	0.957	0.520	0.857	0.520	0.857	0.510	0.353	0.897	0.400	**1.000**	**1.000**	**1.000**	**1.000**
Linux	0.916	0.806	0.082	0.217	0.686	0.778	0.111	0.259	0.285	0.451	0.164	0.232	0.205	0.712	0.168	0.428	0.844	**0.916**	0.773	0.636	0.912	0.741	0.902	0.607	**0.953**	0.871	**0.921**	**0.722**
Apache	**1.000**	**1.000**	0.727	0.517	**1.000**	**1.000**	0.727	0.517	0.948	0.687	0.942	0.269	0.786	0.605	0.948	0.368	**1.000**	**1.000**	**0.998**	**0.867**	**1.000**	**1.000**	0.996	0.759	**1.000**	**1.000**	0.994	0.767
Zookeeper	0.996	0.788	0.842	0.465	0.994	0.904	0.843	0.614	0.988	0.661	**0.988**	0.510	0.967	0.918	0.845	0.809	0.997	0.960	0.792	0.823	0.993	**0.974**	0.850	0.863	**0.998**	0.966	0.972	**0.909**
Hadoop	0.823	0.117	0.535	0.058	0.921	0.785	0.541	0.384	0.691	0.628	**0.889**	0.476	0.483	0.526	0.666	0.434	0.926	**0.951**	0.864	0.750	**0.963**	0.940	0.871	**0.755**	0.927	0.891	0.829	0.733
HealthApp	0.725	0.008	0.311	0.003	0.862	0.010	0.312	0.004	0.461	0.745	0.817	0.462	**0.998**	0.947	**0.997**	0.822	0.993	**0.971**	0.722	0.851	0.862	0.958	0.974	0.877	**0.998**	**0.971**	0.984	**0.902**
OpenStack	0.743	0.682	0.029	0.165	0.752	0.007	0.029	0.002	**1.000**	0.969	0.516	0.289	0.534	0.874	0.406	0.738	**1.000**	**1.000**	**1.000**	**0.958**	0.811	0.689	0.831	0.559	**1.000**	**1.000**	0.972	0.896
HPC	0.748	0.201	0.741	0.136	0.793	0.309	0.721	0.152	0.777	0.660	0.941	0.351	0.782	0.780	**0.997**	0.768	0.871	0.890	0.964	0.726	0.844	0.805	0.973	0.609	**0.986**	**0.897**	0.990	**0.897**
Mac	0.797	0.793	0.245	0.205	0.761	0.229	0.357	0.069	0.737	0.699	0.688	0.283	0.544	0.493	0.390	0.274	0.866	0.864	0.613	0.506	0.814	0.804	0.654	0.469	**0.893**	**0.885**	**0.698**	**0.611**
OpenSSH	0.705	0.689	0.364	0.333	0.707	0.872	0.586	0.487	0.275	0.009	0.289	0.005	0.277	0.081	0.654	0.105	0.747	0.800	0.966	0.743	**0.868**	0.853	0.496	0.472	0.726	**0.909**	**0.968**	**0.779**
Spark	0.000	0.000	0.000	0.000	0.888	0.861	0.394	0.412	0.854	0.020	0.795	0.012	0.476	0.374	0.952	0.299	**0.999**	**0.875**	**0.997**	0.690	0.859	0.832	0.889	0.690	0.974	0.853	0.955	**0.730**
Thunderbird	0.786	0.116	0.163	0.035	0.831	0.237	0.216	0.071	0.579	0.682	0.654	0.290	0.564	0.216	0.401	0.117	0.808	0.796	0.628	0.515	0.870	0.843	0.694	0.540	**0.889**	**0.873**	**0.712**	**0.660**
BGL	0.915	0.587	0.406	0.165	**0.919**	0.624	0.407	0.193	0.918	0.624	**0.949**	0.219	0.245	0.253	0.938	0.261	0.897	**0.917**	0.932	**0.801**	0.902	0.803	0.929	0.716	0.883	0.843	0.899	0.791
HDFS	0.999	0.764	0.621	0.562	0.999	0.935	0.621	0.609	**1.000**	0.968	0.948	0.581	0.721	0.391	0.943	0.312	**1.000**	0.968	0.949	0.946	**1.000**	0.929	**1.000**	0.795	**1.000**	**1.000**	0.949	**0.957**
Average	0.795	0.516	0.410	0.234	0.843	0.554	0.468	0.282	0.716	0.578	0.730	0.317	0.612	0.574	0.736	0.478	0.891	0.912	0.837	0.762	0.872	0.823	0.854	0.651	**0.945**	**0.926**	**0.917**	**0.811**

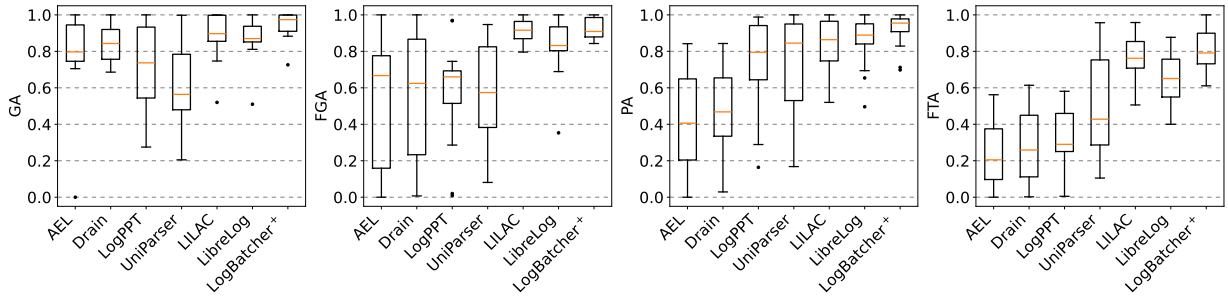

Fig. 8. Robustness comparison between baselines and LogBatcher$^+$

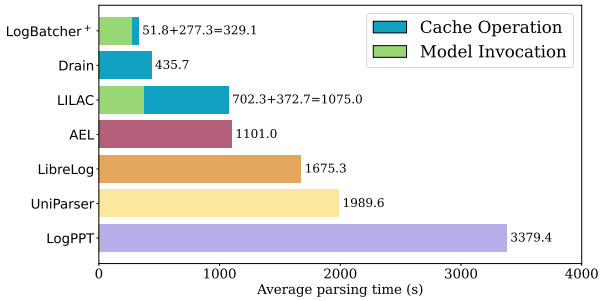

Fig. 9. Average parsing time of baselines and LogBatcher$^+$.

B. *RQ2: How do different modules contribute to LogBatcher$^+$?*

This RQ gives a comprehensive explanation of each module's contribution. Table II shows the results. It is clear that removing any of the three modules will affect performance to some extent.

TABLE II
ABLATION STUDY RESULTS

	GA	FGA	PA	FTA
Full LogBatcher$^+$	0.945	0.926	0.917	0.811
w/o partitioning	0.823($\downarrow$14.8%)	0.801($\downarrow$15.6%)	0.893($\downarrow$2.7%)	0.768($\downarrow$5.6%)
w/o batching	0.846($\downarrow$11.7%)	0.810($\downarrow$14.3%)	0.887($\downarrow$3.4%)	0.768($\downarrow$5.6%)

1) Partitioning: Intuitively, partitioning the logs is beneficial for the group accuracy, and this is indeed the case. Partitioning is the component that most significantly impacts grouping accuracy among the three components. Without it, the GA and FGA drop by 14.8% and 15.6%. Additionally, it also affects the message-level accuracy because the partitioning phase allows us to provide LLMs with commonalities within the input log data and correlate them with the log parsing task description. Without partitioning, the LLM input contains less commonalities, resulting in PA and FTA decreased by 2.7% and 5.6%.

2) Batching: The proposed batching module is designed primarily to provide the LLM with diverse logs and variables, expecting that the LLM can learn from the variability existing among the data. The results demonstrate the importance of batching for the entire parsing process. Without batching, the PA and FTA achieved by LogBatcher$^+$ significantly drops by 3.4% and 5.6%, indicating that the LLM is less effective when the data provided has no diversity. This indicates that the batching module is crucial for LogBatcher$^+$ to achieve high parsing accuracy in terms of exact matching.

In summary, the ablation study confirms the usefulness of the major components of LogBatcher$^+$.

C. *RQ3: How does LogBatcher$^+$ perform with different prompt designs?*

1) Effect of Batch Size: We explore the batch sizes of 1, 5, 10, 15, and 20 (setting the batch size to 1 means removing the batching component). The results appear in Table III. It can be seen that when the batch size approaches 1, the performance drops significantly. The optimal batch size is found to be around 10. When the batch size exceeds 10, the performance of LogBatcher$^+$ lightly decreases. Considering larger batch

sizes can lead to higher LLM invocation overhead, in our experiments we set the default batch size to 10.

2) Variable-Aware Prompt: The variable-aware prompt is designed by incorporating both variable types and historical variables. In this experiment, we evaluate the importance of these two components by removing them from the framework and evaluating the performance. As shown in Table III. Removing either component leads to a performance decrease. For instance, removing variable types results in a 3.5% decrease in GA, while removing historical variables causes a 9.5% drop in PA. While these ablations can reduce token consumption to some extent, we choose to retain them to achieve a good performance.

3) LLM invocation cost: In recent work, LLM-based parsers have concentrated on selecting suitable demonstrations for the query [21], [19], which has led to demonstrations being significantly longer than the query itself. In contrast, our approach improves the efficiency of using LLMs by adding more diverse logs and variables to the query through *batch prompting*. Note that LibreLog is designed for open-source LLMs and cannot be quantified using LLM invocation cost. We select LILAC and LogBatcher$^+$ to calculate the token consumption. The results are illustrated in Table III. It is obvious that, in terms of total token consumption and average token consumption per invocation, our method achieves better results on most datasets. Both LogBatcher$^+$ and LILAC adopt a caching mechanism to reduce the number of queries to LLMs, which significantly reduces the total token consumption. Moreover, LogBatcher$^+$ employs a batching – querying mechanism to provide LLMs with logs and variables instead of demonstration per invocation, which further reduces the token consumption. The low token consumption rate also indicates that LogBatcher$^+$ is more efficient in terms of LLM invocation cost and energy consumption, which is crucial for real-world applications. In particular, we follow a recent study [67] to estimate the carbon footprint of using LLMs as $CO2 = \#Tokens \times Energy\ per\ Token \times gCO2e/kWh$, where $Energy\ per\ Token \approx 0.001$ kWh per 1k tokens and $gCO2e/kWh \approx 240.6$ gCO2e/kWh at Azure US West data centers. Based on this estimation, LogBatcher$^+$ can reduce 15.8% carbon footprint compared to LILAC when parsing all logs in the subject datasets. Besides, using the method for calculating the cost of invoking GPT-4o-mini API described in Section III, LogBatcher$^+$ only costs \$0.00004 per invocation, which is 13.3% lower than LILAC.

TABLE III
AVERAGE ACCURACY AND LLM INVOCATION COST OF LOGBATCHER$^+$ AND LILAC WITH DIFFERENT PROMPT DESIGNS.

	GA	FGA	PA	FTA	$\mathbf{T}_{\text{total}}$	$\mathbf{T}_{\text{invoc}}$
LogBatcher$^+$	**0.945**	**0.926**	**0.917**	0.811	71614	264
LILAC	0.891	0.912	0.837	0.762	82905	299
W/ batch size 1	0.846	0.810	0.887	0.768	**58409**	**210**
W/ batch size 5	0.928	0.910	0.890	0.800	61987	232
W/ batch size 15	0.915	0.921	0.900	**0.813**	76542	282
W/ batch size 20	0.940	0.914	0.899	0.803	80796	298
W/o variable types	0.910	0.903	0.879	0.798	52263	259
W/o historical variables	0.871	0.887	0.822	0.794	63588	262

D. ***RQ4.*** *How do different settings affect LogBatcher$^+$?*

1) Clustering Method: We evaluate the impact of different clustering methods on the performance of LogBatcher$^+$. Specifically, we compare the results of LogBatcher$^+$ using hierarchical clustering [19] and meanshift clustering [61]. The results are shown in Table IV. It is clear that LogBatcher$^+$ achieves the best performance when using the default clustering method (i.e., DBSCAN). It achieves the highest GA, FGA, PA, and FTA (i.e., 0.945, 0.926, 0.917, 0.811 respectively), which are 20.4%, 15.7%, 10.6%, and 13.4% higher than hierarchical clustering, and 9.2%, 5.7%, 8.0%, and 4.8% higher than meanshift clustering. The main reason is that DBSCAN can effectively identify and exclude noisy data points, which is important for datasets with imbalanced log distributions [16].

TABLE IV
RESULTS WITH DIFFERENT CLUSTERING METHODS

	GA	**FGA**	**PA**	**FTA**
LogBatcher$^+$	0.945	0.926	0.917	0.811
w/ Hierarchical clustering	$0.785_{(\downarrow 20.4\%)}$	$0.801_{(\downarrow 15.7\%)}$	$0.829_{(\downarrow 10.6\%)}$	$0.715_{(\downarrow 13.4\%)}$
w/ Meanshift clustering	$0.865_{(\downarrow 9.2\%)}$	$0.876_{(\downarrow 5.7\%)}$	$0.849_{(\downarrow 8.0\%)}$	$0.774_{(\downarrow 4.8\%)}$

2) Sampling Method: Selecting a sampling method involves determining which logs are grouped together into a batch. We examine three widely adopted sampling methods: similarity-based sampling, diversity-based sampling, and random sampling. To obtain a batch of similar logs, we use the approach from [46], employing k-means clustering to identify and batch the most similar logs. For grouping diverse logs, we apply the Determinantal Point Process (DPP, a probabilistic model that favors diverse subsets by giving higher probabilities to dissimilar items) [56] method to ensure diversity. For random grouping, we sample logs randomly from the partition. Before sampling, we ensure that duplicates are removed from the log partition. The result is shown in Table V.

TABLE V
RESULT WITH DIFFERENT SAMPLING METHODS

	GA	**FGA**	**PA**	**FTA**
LogBatcher$^+$	0.945	0.926	0.917	0.811
w/ random sampling	$0.937_{(\downarrow 0.9\%)}$	$0.904_{(\downarrow 2.4\%)}$	$0.906_{(\downarrow 1.2\%)}$	$0.800_{(\downarrow 1.4\%)}$
w/ similarity sampling	$0.921_{(\downarrow 2.6\%)}$	$0.872_{(\downarrow 6.2\%)}$	$0.899_{(\downarrow 2.0\%)}$	$0.793_{(\downarrow 2.3\%)}$

The diversity-based DPP algorithm achieves the best results because it provides LLM with sufficiently diverse logs. Random sampling only resulted in a slight decrease in performance because it can also select diverse logs to some extent. In contrast, similarity-based sampling decreases FGA and FTA by 6.2% and 2.3% respectively. The results demonstrate that the diversity-based sampling method used in LogBatcher$^+$ is effective.

3) Chunk Size: To evaluate the impact of Chunk size on the performance of LogBatcher$^+$, we select the chunk size of 1, 1000, 2000, 5000, 10000, and 20000 (setting the Chunk size to 1 means removing the partitioning component). The results are shown in Figure 10. It can be seen that when the Chunk size approaches 1, the performance

drops significantly. The optimal Chunk size is found to be around 1000, where LogBatcher$^+$ achieve the best GA and FTA. When the Chunk size exceeds 10000, the performance of LogBatcher$^+$slightly decreases. Considering larger Chunk sizes can lead to less efficient on clustering, in our experiments we set the default Chunk size to 10000.

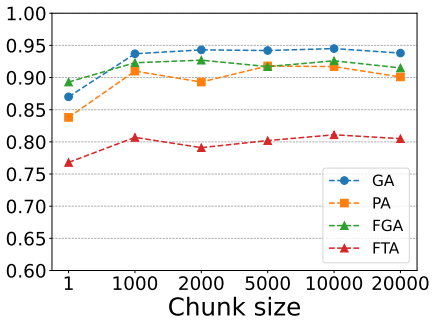

Fig. 10. Average accuracy with different chunk sizes

4) LLM Selection: In our experiments, we use GPT-4o-mini as the default LLM. We also evaluate the performance of our approach with different LLMs. We select two LLMs with different model parameter sizes, including Llama 8B and DeepSeek-V3. Table VI shows the average metrics of LogBatcher$^+$ with different LLMs. It is evident that LogBatcher$^+$ performs well even on a smaller LLM with a parameter count of 8B, achieving an average GA of 0.899 and PA of 0.841. Overall, the larger the model's parameters, the better the performance. These findings demonstrate that LogBatcher$^+$ can be effectively applied to various LLMs with robust performance.

TABLE VI
THE PERFORMANCE OF LOGBATCHER WITH DIFFERENT LLMS

	GA	FGA	PA	FTA
Llama3 (8B)	0.899	0.844	0.841	0.677
DeepSeek-V3	0.950	0.921	0.913	0.814
GPT-4o-mini	0.945	0.926	0.917	0.811

VII. DISCUSSION

A. Advantages of LogBatcher$^+$ over LogBatcher

Although the original version of LogBatcher achieved promising GA and PA, the average FGA and FTA were only 0.863 and 0.627, which is 6.3% and 18.4% lower than those of LogBatcher$^+$. This performance gap is primarily due to the original LogBatcher's failure to handle low-frequency template logs effectively. In contrast, the variable-aware prompting in LogBatcher$^+$ facilitates the parsing of these unseen logs. Moreover, the original LogBatcher employs a simple list as its cache, resulting in an average processing time of 1,535.5 seconds for 3.6 million logs. Thanks to its efficient cache design, LogBatcher$^+$ completes the same task in just 329.1 seconds on average, reducing the time overhead to only 21.4% of the original. We have confirmed the effectiveness and necessity of LogBatcher$^+$ on 14 large-scale datasets.

B. Practicality of LogBatcher$^+$

LogBatcher$^+$ is designed for more practical log parsing with Large Language Models (LLMs). Previous work [18], [35] tends to train a smaller model to perform log parsing. However, it still requires a substantial amount of labeled data for training, which is hard to obtain. Training smaller models via transfer learning (including knowledge distillation) does not guarantee high accuracy. Compared to other LLM-based log parsers, LogBatcher$^+$ does not need any training/fine-tuning process and labeling effort. Nevertheless, our experiments show that LogBatcher$^+$ can still achieve superior accuracy. Additionally, our method eliminates the need to select demonstrations for each query, significantly reducing the LLM invocation overhead. This leads to a notable improvement in cost-effectiveness and model robustness. It is also worth noting that LogBatcher$^+$ is compatible with many LLMs such as Llama and DeepSeek.

A large amount of logs could be generated in production, so it is crucial to ensure that the log parser can perform online parsing, which means it can handle streaming log data. LogBatcher$^+$ can buffer a batch of streaming logs for parsing instead of processing each log individually. After the clustering and sorting process, the logs within the clusters will be parsed through the caching or querying stage, so no training process is needed.

C. Threats to Validity

We have identified the following major threats to validity.

Data Leakage. The data leakage problem of LLM-based log parsers mainly manifests in two aspects: data leakage during the training process of the LLM itself, and the demonstrations during in-context learning disclosing the ground truth templates. Recent studies imply that there is a low probability of direct memorization of LLMs for the log parsing task as without in-context learning, the LLMs' performance significantly drops [19], [21]. Additionally, LogBatcher$^+$ does not require any training process or labeled data, no template is included in the prompt context, thus substantially eliminating the threat of leaking ground-truth templates within the prompt context. Overall, the probability of data leakage in our experiments is negligible.

The Quality of Ground Truth Data. To fairly evaluate the effectiveness of LogBatcher$^+$, the annotation quality for ground truth templates is critical. The datasets used in our experiments are from LogPAI [10], which were manually labeled. It has been observed that labels in the original Loghub-2k datasets [68] have some errors and inconsistent labeling styles [59]. To mitigate this threat, we use the new datasets Loghub-2.0 [16], which is corrected by [59]. The results confirm that LogBatcher$^+$ is effective on large-scale datasets.

Randomness. Bias from randomness may affect the evaluation in two aspects: (1) the randomness of LLMs, and (2) the randomness introduced during the experiments, including the permutation of logs in a batch and the random sampling of logs. To mitigate the instability of LLM outputs, we set the $temperature$ of LLMs to 0. To mitigate the latter threat,

we repeat the experiments five times and report the average results.

VIII. Conclusion

To overcome the limitations of existing log parsers, we propose LogBatcher$^+$, a cost-effective LLM-based log parser that requires no training process or labeled demonstrations. LogBatcher$^+$ leverages latent characteristics of log data and reduces the LLM inference overhead by batching a group of logs and variables. We have conducted extensive experiments on the public log dataset and the results show that LogBatcher$^+$ is effective and efficient for log parsing. We believe this *demonstration-free*, *training-free*, and *cost-effective* log parser has potential to make LLM-based log parsing more practical.

Data Availability: Our source code and experimental data are publicly available at https://github.com/LogIntelligence/LogBatcher.

References

[1] 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.

[2] V.-H. Le and H. Zhang, "Log-based anomaly detection with deep learning: How far are we?" in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1356–1367.

[3] ——, "Log-based anomaly detection without log parsing," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 492–504.

[4] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 1795–1812.

[5] A. Oprea, Z. Li, T.-F. Yen, S. H. Chin, and S. Alrwais, "Detection of early-stage enterprise infection by mining large-scale log data," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 45–56.

[6] A. Lyons, J. Gamba, A. Shawaga, J. Reardon, J. Tapiador, S. Egelman, and N. Vallina-Rodríguez, "Log:{It's} big,{It's} heavy,{It's} filled with personal data! measuring the logging of sensitive information in the android ecosystem," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2115–2132.

[7] T. Jia, L. Yang, P. Chen, Y. Li, F. Meng, and J. Xu, "Logsed: Anomaly diagnosis through mining time-weighted control flow graph in logs," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. IEEE, 2017, pp. 447–455.

[8] V.-H. Le and H. Zhang, "Prelog: A pre-trained model for log analytics," *Proceedings of the ACM on Management of Data*, vol. 2, no. 3, pp. 1–28, 2024.

[9] S. He, Q. Lin, J.-G. Lou, H. Zhang, M. R. Lyu, and D. Zhang, "Identifying impactful service system problems via log analysis," in *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 60–70.

[10] 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.

[11] 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.

[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] H. Dai, H. Li, C. S. Chen, W. Shang, and T.-H. Chen, "Logram: Efficient log parsing using n-gram dictionaries," *IEEE Transactions on Software Engineering*, 2020.

[15] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2009, pp. 1255–1264.

[16] Z. Jiang, J. Liu, J. Huang, Y. Li, Y. Huo, J. Gu, Z. Chen, J. Zhu, and M. R. Lyu, "A large-scale evaluation for log parsing techniques: How far are we?" in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024.

[17] S. Petrescu, F. Den Hengst, A. Uta, and J. S. Rellermeyer, "Log parsing evaluation in the era of modern software systems," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 379–390.

[18] 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.

[19] Z. Jiang, J. Liu, Z. Chen, Y. Li, J. Huang, Y. Huo, P. He, J. Gu, and M. R. Lyu, "Lilac: Log parsing using llms with adaptive parsing cache," 2024.

[20] V.-H. Le and H. Zhang, "Log parsing: How far can chatgpt go?" in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1699–1704.

[21] 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.

[22] S. Kabinna, C.-P. Bezemer, W. Shang, M. D. Syer, and A. E. Hassan, "Examining the stability of logging statements," *Empirical Software Engineering*, vol. 23, no. 1, pp. 290–333, 2018.

[23] Z. Ma, D. J. Kim, and T.-H. Chen, "Librelog: Accurate and efficient unsupervised log parsing using open-source large language models," *arXiv preprint arXiv:2408.01585*, 2024.

[24] X. Li, H. Zhang, V.-H. Le, and P. Chen, "Logshrink: Effective log compression by leveraging commonality and variability of log data," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[25] Y. Xiao, V.-H. Le, and H. Zhang, "Demonstration-free: Towards more practical log parsing with large language models," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 153–165.

[26] 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.

[27] M. Nagappan, K. Wu, and M. A. Vouk, "Efficiently extracting operational profiles from execution logs using suffix arrays," in *2009 20th International Symposium on Software Reliability Engineering*. IEEE, 2009, pp. 41–50.

[28] S. Yu, P. He, N. Chen, and Y. Wu, "Brain: Log parsing with bidirectional parallel tree," *IEEE Transactions on Services Computing*, 2023.

[29] Z. M. Jiang, A. E. Hassan, P. Flora, and G. Hamann, "Abstracting execution logs to execution events for enterprise applications (short paper)," in *2008 The Eighth International Conference on Quality Software*. IEEE, 2008, pp. 181–186.

[30] 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.

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

[32] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *2009 ninth IEEE international conference on data mining*. IEEE, 2009, pp. 149–158.

[33] L. Tang, T. Li, and C.-S. Perng, "Logsig: Generating system events from raw textual logs," in *Proceedings of the 20th ACM international conference on Information and knowledge management*, 2011, pp. 785–794.

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

[35] Y. Liu, X. Zhang, S. He, H. Zhang, L. Li, Y. Kang, Y. Xu, M. Ma, Q. Lin, Y. Dang *et al.*, "Uniparser: A unified log parser for heterogeneous log data," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 1893–1901.

[36] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019, pp. 4171–4186.

[37] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 7871–7880.

[38] J. Guo, J. Li, D. Li, A. M. H. Tiong, B. Li, D. Tao, and S. Hoi, "From images to textual prompts: Zero-shot visual question answering with frozen large language models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 10 867–10 877.

[39] Z. Shao, Z. Yu, M. Wang, and J. Yu, "Prompting large language models with answer heuristics for knowledge-based visual question answering," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 14 974–14 983.

[40] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[41] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[42] T. Khot, H. Trivedi, M. Finlayson, Y. Fu, K. Richardson, P. Clark, and A. Sabharwal, "Decomposed prompting: A modular approach for solving complex tasks," in *The Eleventh International Conference on Learning Representations*, 2023.

[43] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[44] A. Zhong, D. Mo, G. Liu, J. Liu, Q. Lu, Q. Zhou, J. Wu, Q. Li, and Q. Wen, "Logparser-llm: Advancing efficient log parsing with large language models," in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2024, pp. 4559–4570.

[45] W. Zhang, H. Guo, A. Le, J. Yang, J. Liu, Z. Li, T. Zheng, S. Xu, R. Zang, L. Zheng, and B. Zhang, "Lemur: Log parsing with entropy sampling and chain-of-thought merging," 2024.

[46] Z. Cheng, J. Kasai, and T. Yu, "Batch prompting: Efficient inference with large language model apis," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: Industry Track*, 2023, pp. 792–810.

[47] D. Hidvégi, K. Etemadi, S. Bobadilla, and M. Monperrus, "Cigar: Cost-efficient program repair with llms," *arXiv preprint arXiv:2402.06598*, 2024.

[48] "Openai chatgpt," 2024. [Online]. Available: https://platform.openai.com/docs/models/gpt-3-5-turbo

[49] H. Mi, H. Wang, Y. Zhou, M. R.-T. Lyu, and H. Cai, "Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 6, pp. 1245–1255, 2013.

[50] "Pricing," 2024. [Online]. Available: https://openai.com/pricing

[51] 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.

[52] M. Mizutani, "Incremental mining of system log format," in *2013 IEEE International Conference on Services Computing*. IEEE, 2013, pp. 595–602.

[53] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *kdd*, vol. 96, no. 34, 1996, pp. 226–231.

[54] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information processing & management*, vol. 24, no. 5, pp. 513–523, 1988.

[55] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "Dbscan revisited, revisited: why and how you should (still) use dbscan," *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 3, pp. 1–21, 2017.

[56] A. Kulesza, B. Taskar *et al.*, "Determinantal point processes for machine learning," *Foundations and Trends® in Machine Learning*, vol. 5, no. 2–3, pp. 123–286, 2012.

[57] 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.

[58] Q. Qin, R. Aghili, H. Li, and E. Merlo, "Preprocessing is all you need: Boosting the performance of log parsers with a general preprocessing framework," *arXiv preprint arXiv:2412.05254*, 2024.

[59] 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.

[60] "A large collection of system log datasets for ai-powered log analytics," 2023. [Online]. Available: https://github.com/logpai/loghub

[61] Z. Ma, A. R. Chen, D. J. Kim, T.-H. Chen, and S. 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*. ACM, Apr. 2024. [Online]. Available: http://dx.doi.org/10.1145/3597503.3639150

[62] X. Yi, V.-H. Le, and H. Zhang, "Logbatcher repository," 2024. [Online]. Available: https://github.com/LogIntelligence/LogBatcher

[63] Y. Wu, S. Yu, and Y. Li, "Log parsing with self-generated in-context learning and self-correction," *arXiv preprint arXiv:2406.03376*, 2024.

[64] OpenAI, "Gpt-4 technical report," *ArXiv*, vol. abs/2303.08774, 2023.

[65] "40mini," 2024. [Online]. Available: https://platform.openai.com/docs/models/gpt-4o-mini

[66] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[67] C. Baquero, "The energy footprint of humans and large language models," 2024. [Online]. Available: https://cacm.acm.org/blogcacm/the-energy-footprint-of-humans-and-large-language-models/

[68] J. Zhu, S. He, P. He, J. Liu, and M. R. Lyu, "Loghub: A large collection of system log datasets for ai-driven log analytics," 2023.