Demonstration-Free: Towards More Practical Log Parsing with Large Language Models

Yi Xiao
Chongqing University
Chongqing, China
yixiao@cqu.edu.cn

Van-Hoang Le
The University of Newcastle
NSW, Australia
hoang.le@newcastle.edu.au

Hongyu Zhang*
Chongqing University
Chongqing, China
hyzhang@cqu.edu.cn

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 divide logs 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. We have conducted experiments on 16 public log datasets and the results show that LogBatcher is effective and efficient for log parsing.

KEYWORDS

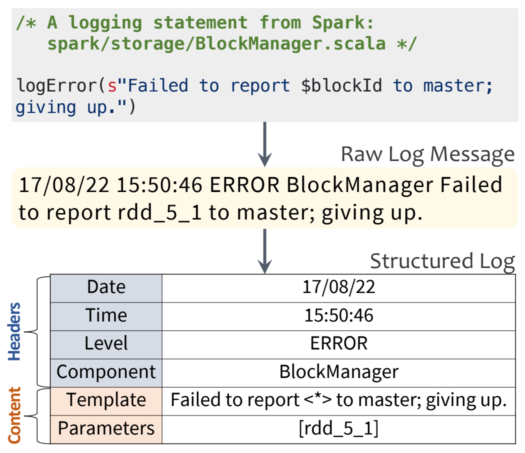
Log Parsing, Batch Prompting, Large Language Models

ACM Reference Format:
Yi Xiao, Van-Hoang Le, and Hongyu Zhang. 2024. Demonstration-Free: Towards More Practical Log Parsing with Large Language Models. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3694994

1 INTRODUCTION

Software-intensive 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 [24, 25, 60], ensuring application security [33, 38, 43], and diagnosing errors [14, 16, 28].

To facilitate various downstream analytics tasks, log parsing, which parses free-text into a structured format [62], 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 [13]. 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.



```
/* A logging statement from Spark:
   spark/storage/BlockManager.scala */

logError(s"Failed to report $blockId to master;
giving up.")
```

```
17/08/22 15:50:46 ERROR BlockManager Failed
to report rdd_5_1 to master; giving up.
```

Date	17/08/22
Time	15:50:46
Level	ERROR
Component	BlockManager
Template	Failed to report <*> to master; giving up.
Parameters	[rdd_5_1]

Figure 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

*Hongyu Zhang is the corresponding author

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3694994

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 [6, 8, 12, 35] 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 unsatisfactory [18, 45]. 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 [17, 27]. 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 [26].

To address these limitations, recent studies [17, 26, 55] 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 [17, 26]. Furthermore, demonstrations can be quickly outdated as the volume and format of logs rapidly change [20, 60]. 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.

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 [30] 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 the public LogPai dataset [62]. The results show that LogBatcher outperforms state-of-the-art baselines in terms of both accuracy and LLM inference cost. It can achieve an average Group Accuracy [62] of 0.974 and Message-Level Accuracy of 0.904, which are significantly higher than the best-performing supervised LLM-based log parser (i.e., LILAC [17]). Moreover, LogBatcher is robust across diverse log datasets without the need for demonstrations, and can substantially reduce the cost of LLM invocation by at least 106%.

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, which allows LLMs to better incorporate the latent commonalities and variabilities among log messages. Furthermore, the token consumption of LLMs is reduced.
(3) We conduct a comprehensive evaluation on the public LogPai dataset [62]. Experimental results show that LogBatcher outperforms state-of-the-art baselines in terms of both accuracy and LLM invocation cost.

2 BACKGROUND AND RELATED WORK

2.1 Log Parsing

Log parsing is one of the first steps for log analysis tasks [62]. 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 [40, 56] or designing handcrafted regular expressions to extract log templates and parameters [62]. However, these approaches are impractical due to the inaccessibility of the source code (especially for third-party libraries [62]) 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 [6, 12, 19, 58] assume that log templates inherit some common patterns which emerge constantly across the entire log dataset. Some parsers [6, 39, 51] extract log templates by identifying the constant parts of log messages through the mining of frequent patterns, for example, Logram [6] 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 [10, 49, 50] employ clustering techniques to group logs and extract the constant portions of log messages for log parsing. Heuristics-based log parsers [12, 19, 58] leverage unique characteristics from log messages to extract common templates efficiently. For example, AEL [19] 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 [58] 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 [32] 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 [27] 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 [26].

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 [17, 26, 55] 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.

2.2 Log Parsing with Large Language Models

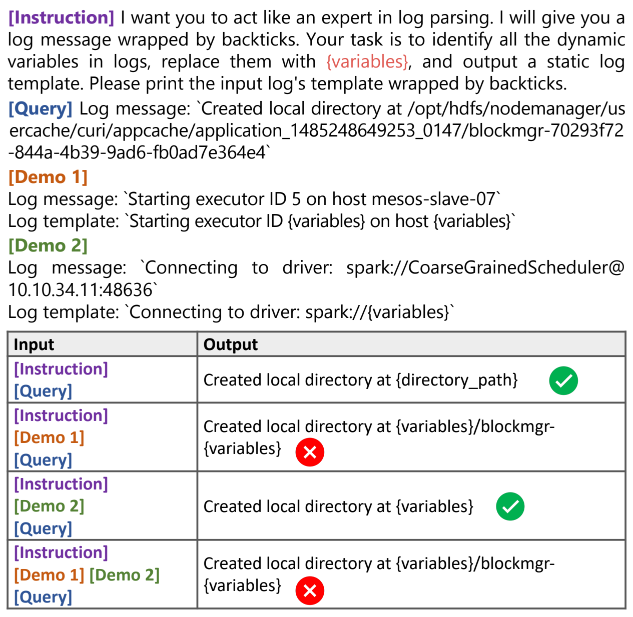
Large language models (LLMs) have achieved remarkable success in various natural language processing [7, 29] and computer vision tasks [11, 48]. In-context learning is a promising prompt engineering method for adopting LLMs without fine-tuning them [31]. In-context learning typically requires an *instruction* 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 [22, 53, 54].

Le and Zhang [26] validated the potential of LLMs in log parsing and obtained promising results. DivLog [55] and LILAC [17] 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 [59] invokes LLM to merge generated similar templates, improving the accuracy of log parsing groupings. However, it requires extensive hyperparameter tuning for specific datasets. It has been found that these LLM-based log parsers have outperformed semantic-based log parsers (e.g., LogPPT [27] and UniParser [32]) in terms of parsing accuracy [17, 55].

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 [5, 15] 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.

3 A MOTIVATING EXAMPLE

Recently, several studies [17, 26, 55] 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} ✗

Figure 2: Selecting in-context demonstrations for log parsing on Spark (*Results are produced using gpt-3.5-turbo [2] with instruction and demonstrations adopted from [17]*)

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 [17, 26]. Hence, to overcome this problem, recent studies [17, 55] 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 [20, 60]. 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 [17]. As shown in Figure 3, regarding widely-used group accuracy (GA) [62] and message-level accuracy (MLA) metrics [32], 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.

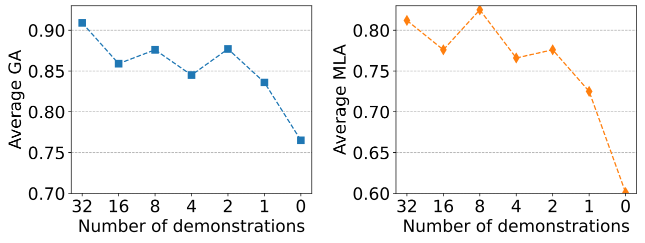


Figure 3: Impact of different numbers of demonstrations

LLM invocation cost. Log data can be generated in a massive volume in production. For example, Mi et al. [37] 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 [2] with a prompt consisting of one instruction (66 tokens in total[1]) and one log message (55 tokens in total) will cost (55 + 66) × 100,000,000 × (0.50/1,000,000) = \$6,050 because the price of GPT-3.5-Turbo API services is \$0.5 per 1M tokens [3]. Due to the large amount of log messages, the cost for LLM-based log parsing could pose a significant financial burden in practice.

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.

4 METHODOLOGY

Drawing upon the observations described in Section 3, 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 allow 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 [37, 52], we divide the raw log data into multiple chunks before analysis. Log chunks are processed in parallel, 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.

[1]We reference the newest pricing of text tokens from OpenAI's *tiktoken* package: https://github.com/openai/tiktoken

4.1 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 [9] to partition logs. Figure 5 illustrates the log partitioning process.

4.1.1 *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 $L = \{L_1, L_2, \ldots, L_n\}$, each log L_i is tokenized into a set of tokens $\{t_{i1}, t_{i2}, \ldots, t_{im}\}$.

4.1.2 *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 [60], we adopt the Frequency-Inverse Document Frequency (TF-IDF) [46] 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:

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

4.1.3 *Clustering & Sorting.*

LogBatcher adopts the DBSCAN algorithm (Density-Based Spatial Clustering of Applications with

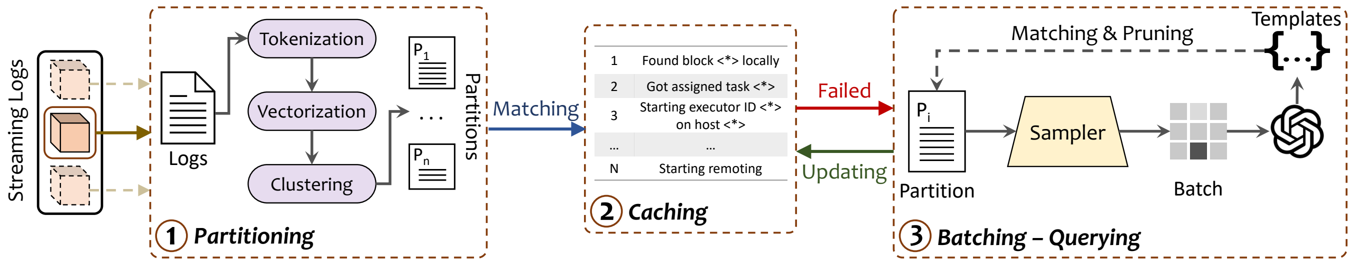


Figure 4: An overview of LogBatcher

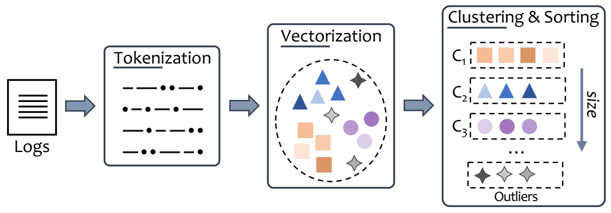


Figure 5: Log partitioning through clustering

Noise) [9] 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 [47]; (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.

4.2 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 leverage a simple caching mechanism to store previously parsed log templates and match them with logs in the current partition. 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 into the cache. Each new item in the cache contains three values: (1) a newly generated template from LLM, (2) a reference log that can match the template, and (3) the matching frequency (how many logs the template has matched). We detail the usage of these values below.

To match logs with the template, some log parsers [12, 17] select logs with a similarity above a certain threshold to the template and consider them as matches, which could result in mismatches. In our approach, we perform regular expression matching. Specifically, this involves replacing "<*>" in the log templates with the generic matching symbol "`(.?)`", allowing regular expressions to check if the logs and templates match exactly and return all the corresponding variables. Additionally, inspired by [49], the reference log is used to verify whether its length is consistent with that of the target log, making our caching more precise. To enhance caching efficiency, we also dynamically sort the templates in the cache so that the frequently occurring templates can be checked first.

4.3 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, 49]. 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.

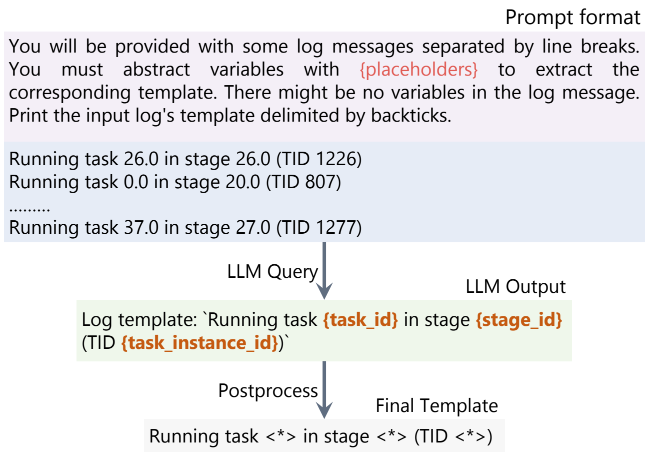
4.3.1 Batching. After partitioning, logs in each partition already exhibit commonalities in their semantics and syntax. We sample a set of logs from each partition to form an input batch for LLMs. To this end, we adopt a diversity-based sampling method to select logs that maximize the sample diversity. 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 [23] 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.

4.3.2 Prompting Design. A common in-context learning paradigm consists of three parts: instruction, demonstration and query. Since our method is demonstration-free, our prompt consists only of instruction and query. Following previous work [17, 26], we design and use the prompt format, as shown in Figure 6. However, different from them, we provide LLMs with the input in the form of a batch as follows:

(1) *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 logs may not contain variables to avoid over-parsing, where the LLM tries to find variables in every log.

(2) *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.



Prompt format

You will be provided with some log messages separated by line breaks. You must abstract variables with {placeholders} to extract the corresponding template. There might be no variables in the log message. Print the input log's template delimited by backticks.

```
Running task 26.0 in stage 26.0 (TID 1226)
Running task 0.0 in stage 20.0 (TID 807)
.........
Running task 37.0 in stage 27.0 (TID 1277)
```

Log template: `Running task {task_id} in stage {stage_id} (TID {task_instance_id})`

```
Running task <*> in stage <*> (TID <*>)
```

Figure 6: An illustration of our prompt design

4.3.3 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. [21] customized some heuristic rules, to correct the identified template. Some related works [17, 27] 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, [21] only considers decimal numbers in the logs as variables, but in reality, hexadecimal numbers appear just as frequently.

4.3.4 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 4.2, 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: $C$: Parsed Cluster
       $T$: Identified Template
Output: $C_{new}$: New Cluster
regex ← convertToRegex($T$)
$C_{new} \leftarrow \emptyset$
foreach $log \in C$ do
    if $match(log, regex)$ then
        $C \leftarrow C \setminus \{\log\}$
        $C_{new} \leftarrow C_{new} \cup \{\log\}$
    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.

5 EXPERIMENTAL DESIGN

5.1 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 four state-of-the-art unsupervised data-driven log parsers (i.e., Drain [12], AEL [19], Brain [58], and Logram [6]) and two LLM-based supervised log parsers (i.e., DivLog [55] and LILAC [17]). We adopt the implementation of these methods from their public replication packages [17, 55, 62]. DivLog and LILAC are recently proposed to leverage the in-context learning capacity of LLMs for log parsing. For a fair comparison, we use the same settings from LILAC [17] to reproduce the results of both DivLog and LILAC, in which 32 candidates are sampled from the log data and 3 demonstrations are selected as the parsing context for each queried log.

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. Specifically, 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$_{\text{caching}}$: we directly remove the caching component; and (3) w/o$_{\text{batching}}$: we only use one log entry as the LLM input.

RQ3. How does LogBatcher perform with demonstrations? Although LogBatcher is demonstration-free, it can easily be extended to use labeled demonstration examples as other supervised LLM-based approaches do. In this RQ, we adopt the same setting from LILAC [17] to select a few demonstrations for each LLM invocation. Specifically, we first sample 32 labeled candidate logs from each dataset and then select the most similar candidate logs as demonstrations for each LLM query. We compare LogBatcher with LILAC (the top-performing supervised LLM-based log parser) to evaluate the effectiveness of our approach with different numbers of demonstration examples.

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 batch sizes, and different LLMs to evaluate the performance of LogBatcher.

5.2 Datasets

We conduct experiments on 16 public log datasets (Loghub-2k [1]) originated from the LogPai project [62]. These datasets cover logs from distributed systems, standalone software, supercomputers, PC operating systems, mobile systems, microservices, etc. Zhu et al. [62] 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 [1, 18, 21]. Therefore, following existing work [27, 34, 55], we use the version of the datasets corrected by Khan et al. [21]. Furthermore, as the Proxifier dataset has many different versions, we calibrated some labels according to the guidelines proposed by [18]. The datasets we used in our experiments are publicly available at our webpage [57].

5.3 Evaluation Metrics

Following recent studies [26, 27, 32], we use three 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.

Message Level Accuracy (MLA): Message Level Accuracy [32] 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.

Edit Distance (ED): Edit Distance assesses the performance of template extraction in terms of string comparison [41]. It calculates the minimum number of actions needed to convert one template into another. We apply normalized Edit Distance [36], which computes the mean Edit Distance of all compared template pairs in the dataset (parsed templates vs ground truth templates).

Due to space constraints, we provide the evaluation results in terms of other metrics, i.e., F1 score of Group Accuracy (FGA) [18] and F1 score of Template Accuracy (FTA) [21], on our webpage. In addition, to assess the efficiency of our proposed approach, we follow recent studies [5, 15] 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.

5.4 Baselines

Based on the recent benchmark studies [18, 21], we select four state-of-the-art unsupervised data-driven log parsers (i.e., Drain [12], AEL [19], Brain [58], and Logram [6]) and two recent LLM-based supervised log parsers (i.e., DivLog [55] and LILAC [17]). We adopt the latest version of these methods from their publicly available replication packages [17, 55, 62]. We did not include semantic-based log parsers (such as Uniparser and LogPPT) in our comparison because previous studies [17, 55] have shown that LLM-based log parsers (e.g., LILAC and DivLog) significantly outperform them with the same amount of labeled data. To ensure a fair comparison, we replicate the results of both DivLog and LILAC using the same settings as in LILAC [17]. Specially, 32 labeled candidates are sampled from the log data and three of them are selected as the ICL demonstrations for each queried log. We also compare LogBatcher with the unsupervised variant of LILAC [17] (i.e., LILAC $_{\text{w/o ICL}}$) by removing the labelled ICL demonstrations from the query context.

5.5 Implementation and Settings

Implementation. In our experiments, we set the default LLM to *GPT-3.5-Turbo* (version 0125[2]), which is widely used in recent research [26, 55]. We conduct our experiments on a Ubuntu 20.04 LTS server with Python 3.8. We invoke the LLM through the Python library provided by OpenAI [42].

Settings. We adopt the implementation of DBSCAN provided by sklearn [44] 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 [17, 18, 21], we repeat all experiments 5 times and report the average results.

6 RESULTS AND ANALYSIS

6.1 RQ1: How does LogBatcher perform compared to the baselines?

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

6.1.1 Effectiveness. Table 1 provides a comparative analysis of various log parsing methods across multiple datasets in terms of Group Accuracy (GA), Message-Level Accuracy (MLA), and Edit Distance (ED). For each dataset, the highest accuracy of each metric is highlighted in **bold**. Experimental results show that LogBatcher significantly outperforms other unsupervised log parsers, including Drain [12], AEL [19], Brain [58], and Logram [6]. Specifically, LogBatcher exceeds the highest GA, MLA, and ED of these methods by 9.1%, 36.0%, and 4.8% on average, respectively. Compared to supervised LLM-based log parsers, i.e., DivLog [55] and LILAC [17], LogBatcher also achieves superior performance. Specifically, LogBatcher significantly outperforms DivLog in terms of all three metrics. For example, it achieves better GA on all datasets, i.e., 1.6% (Apache) to 74.8% (OpenStack) higher than DivLog. Compared to LILAC, the top-performing LLM-based log parser, LogBatcher achieves better or comparable parsing accuracy. We compare LogBatcher with LILAC in two settings: LILAC$_{\text{w/o ICL}}$ (i.e., without

[2]https://platform.openai.com/docs/models/gpt-3-5-turbo

Table 1: Comparison with the state-of-the-art log parsers

	Drain			**AEL**			**Brain**			**Logram**			***DivLog***			**LILAC $_{\text{w/o ICL}}$**			***LILAC***			**LogBatcher**		
	GA	MLA	ED	GA	MLA	ED	GA	MLA	ED	GA	MLA	ED	GA	MLA	ED	GA	MLA	ED	GA	MLA	ED	GA	MLA	ED
HDFS	0.998	0.999	0.999	0.998	0.999	0.999	0.998	0.959	0.997	0.930	0.961	0.993	0.930	0.996	0.999	**1.000**	0.943	0.999	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
Hadoop	0.948	0.613	0.882	0.869	0.606	0.901	0.950	0.158	0.751	0.694	0.195	0.708	0.683	0.744	0.915	0.958	0.843	0.928	0.991	**0.958**	**0.986**	**0.989**	0.888	0.953
Spark	0.920	0.398	0.963	0.905	0.398	0.952	0.998	0.376	0.950	0.470	0.296	0.915	0.738	0.960	0.983	0.998	0.806	0.990	**0.999**	**0.983**	**0.998**	0.998	0.973	0.989
Zookeeper	0.967	0.799	0.981	0.965	0.800	0.981	0.989	0.779	0.987	0.956	0.805	0.970	0.955	0.979	0.998	0.989	0.374	0.886	**1.000**	0.987	**0.999**	0.995	**0.988**	0.995
BGL	0.963	0.479	0.885	0.957	0.474	0.883	**0.996**	0.426	0.891	0.702	0.282	0.785	0.736	0.950	**0.990**	0.941	0.870	0.955	0.983	**0.972**	0.989	0.994	0.950	**0.992**
HPC	0.887	0.662	0.872	0.904	0.680	0.880	0.945	0.660	0.973	**0.978**	0.751	0.870	0.935	0.980	0.997	0.911	0.641	0.913	0.970	**0.994**	**0.999**	0.953	0.946	0.995
Thunderbird	0.957	0.180	0.941	0.945	0.180	0.943	0.971	0.060	0.932	0.554	0.097	0.826	0.234	0.879	0.978	0.957	0.852	0.960	**0.984**	**0.913**	**0.983**	0.897	0.838	0.950
Windows	0.997	0.466	0.948	0.691	0.158	0.840	0.997	0.463	**0.976**	0.694	0.141	0.915	0.710	0.715	0.903	0.694	0.020	0.692	0.696	**0.685**	0.897	**0.997**	0.644	0.871
Linux	0.422	0.217	0.750	0.405	0.205	0.745	0.358	0.176	0.770	0.186	0.125	0.684	0.484	0.620	0.935	0.298	0.344	0.903	0.298	0.422	0.926	**0.995**	**0.974**	**0.990**
Android	0.885	0.750	**0.972**	0.773	0.540	0.876	0.960	0.253	0.924	0.795	0.436	0.822	0.737	0.677	0.952	0.931	0.481	0.890	0.953	0.627	0.923	**0.967**	**0.791**	0.955
HealthApp	0.901	0.375	0.749	0.893	0.368	0.744	**1.000**	0.261	0.871	0.833	0.677	0.850	0.876	0.984	0.997	0.901	0.866	0.945	0.998	**0.988**	**0.998**	0.984	0.980	0.970
Apache	**1.000**	0.978	0.996	**1.000**	0.978	0.996	**1.000**	0.984	0.996	1.000	0.972	0.995	0.984	0.985	0.997	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	0.999
Proxifier	0.765	0.704	0.980	0.826	0.690	0.972	0.527	0.704	0.945	0.531	0.477	0.816	0.531	0.993	**0.999**	0.730	0.512	0.941	**1.000**	0.995	0.999	**1.000**	**1.000**	**1.000**
OpenSSH	0.789	0.594	0.919	0.547	0.729	0.965	**1.000**	0.287	0.948	0.802	0.928	0.960	0.749	0.987	**0.993**	0.492	0.439	0.910	0.753	0.805	0.983	**0.996**	**0.975**	0.987
OpenStack	0.224	0.105	0.693	0.249	0.034	0.718	0.492	0.112	0.937	0.315	0.071	0.724	0.220	0.437	0.873	0.717	0.400	0.815	**1.000**	0.977	0.991	0.968	**0.984**	**0.993**
Mac	0.814	0.392	0.896	0.765	0.284	0.835	**0.949**	0.383	0.902	0.759	0.359	0.843	0.712	0.549	0.898	0.834	**0.626**	**0.915**	0.805	0.562	0.892	0.857	0.543	0.881
Average	0.840	0.544	0.902	0.793	0.508	0.889	0.883	0.440	0.922	0.700	0.473	0.855	0.701	0.839	0.963	0.834	0.626	0.915	0.902	0.867	**0.973**	**0.974**	**0.904**	0.970

any labeled demonstration) and the default LILAC (i.e., with 32 labeled candidates and three ICL demonstrations, which is the default setting of LILAC). LogBatcher exhibits a substantially higher accuracy than LILAC$_{\text{w/o ICL}}$ in terms of three metrics on almost all datasets. Compared to the default setting of LILAC, LogBatcher still outperforms by 7.2% and 3.6% in terms of average GA and MLA. It also achieves comparable results in terms of ED (0.3% lower on average). It is worth noting that without any labeled demonstrations, LogBatcher can still achieve the best average GA and MLA, and the second-best average ED. Overall, the experimental results confirm that LogBatcher is effective for the log parsing task.

6.1.2 *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 [62]. 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 7 shows the results. It is obvious that LogBatcher consistently achieves the narrowest interquartile range (IQR) across all three metrics, indicating the stable performance of LogBatcher. Specifically, LogBatcher yields a median of 0.99 for GA robustness, 0.94 for PA robustness and 0.99 for ED robustness, which are better or comparable to the top-performing baseline, 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.

6.1.3 *Efficiency.*

In recent work, LLM-based parsers have concentrated on selecting suitable demonstrations for the query [17, 55], which has led to demonstrations being significantly longer than the query itself. Conversely, our approach improve the efficiency of using LLMs by adding more diverse logs to the query through *batch-prompting*. To compare the efficiency between LogBatcher and other LLM-based parsers, we calculate the token consumption for all LLM-based log parsing baselines (i.e., DivLog and LILAC). The results are illustrated in Table 2. It is obvious that, in terms of total token consumption and average token consumption per invocation, our method achieves the best results on most datasets. For example, LogBatcher exhibits the lowest $\mathbf{T}_{\text{total}}$ and $\mathbf{T}_{\text{invoc}}$ on 14 out of 16 datasets, with an average of 173 tokens per invocation, which is 26.4% and 44.0% lower than DivLog and LILAC, respectively. We notice that the total token consumption of DivLog is extremely high although its average token consumption per invocation is lower than LILAC. This is because DivLog queries LLMs for each log message, making parsing costly in practice. In contrast,

Table 2: Efficiency of LLM-based Log parsers (#tokens)

	$\mathbf{T}_{\text{total}}$			$\mathbf{T}_{\text{invoc}}$		
	DivLog	LILAC	LogBatcher	DivLog	LILAC	LogBatcher
HDFS	706689	**4793**	6101	353	**342**	436
Hadoop	491299	33519	**15695**	246	308	**141**
Spark	409054	10587	**4993**	205	294	**156**
Zookeeper	360558	14858	**6930**	180	310	**122**
BGL	420346	35061	**15692**	210	297	**139**
HPC	288954	9179	**5076**	144	255	**110**
Thunderbird	522583	43546	**18781**	261	283	**107**
Windows	461847	14459	**5721**	231	295	**114**
Linux	444417	28972	**9161**	222	287	**82**
Android	430660	32574	**17967**	215	256	**115**
HealthApp	351681	16393	**6980**	176	264	**94**
Apache	364608	1549	**950**	182	258	**158**
Proxifier	494994	6724	**4160**	247	480	**347**
OpenSSH	490727	7921	**4206**	245	317	**162**
OpenStack	686037	14924	**13798**	343	364	**337**
Mac	581290	109260	**51002**	291	339	**155**
Average	469109	24020	**11701**	235	309	**173**

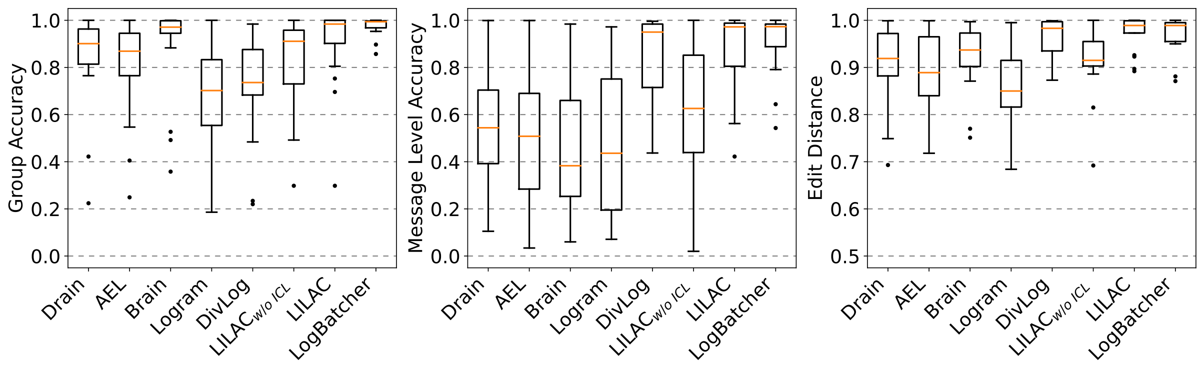


Figure 7: Robustness comparison between baselines and LogBatcher

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 more log messages 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 [4] 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 the carbon footprint by 2× and 40× compared to DivLog and LILAC when parsing all logs in the subject datasets. Besides, using the method for calculating the cost of invoking GPT-3.5-Turbo API described in Section 3, LogBatcher only costs $0.008 per invocation, which is 1.3× and 1.8× lower than DivLog and LILAC.

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.

6.2 RQ2: How do different modules contribute to LogBatcher?

This RQ gives a comprehensive explanation of each module's contribution. Table 3 shows the results. It is clear that removing any of the three modules will affect performance to some extent.

Table 3: Ablation study results

	GA	MLA	ED
Full LogBatcher	0.974	0.904	0.970
w/o$_{\text{partitioning}}$	0.790$_{(\downarrow 23.3\%)}$	0.770$_{(\downarrow 17.4\%)}$	0.896$_{(\downarrow 8.3\%)}$
w/o$_{\text{caching}}$	0.830$_{(\downarrow 17.3\%)}$	0.803$_{(\downarrow 12.6\%)}$	0.935$_{(\downarrow 3.7\%)}$
w/o$_{\text{batching}}$	0.928$_{(\downarrow 5.0\%)}$	0.724$_{(\downarrow 24.9\%)}$	0.910$_{(\downarrow 6.6\%)}$

6.2.1 *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 drops by 23.3%. 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 MLA decreased by 17.4%.

6.2.2 *Caching.*

Due to the inherent limitations of the clustering method, logs belonging to the same template can be divided into different partitions. Within these partitions, our method achieves higher parsing accuracy for larger ones. When the caching module is removed, the results from these larger partitions cannot be used to guide the parsing of smaller ones. In other words, the smaller partitions are parsed independently, leading to poorer overall model performance. For example, removing caching decreases the GA and MLA of LogBatcher by 17.3% and 12.6%, respectively, confirming the usefulness of the caching mechanism.

6.2.3 *Batching.*

The proposed batching module is designed primarily to provide the LLM with diverse logs, expecting that the LLM can learn from the variability existing among the log data. The results demonstrate the importance of batching for the entire parsing process. Without batching, the MLA achieved by LogBatcher significantly drops by 24.9%, indicating that the LLM is less effective when the data provided has no diversity. This indicates that the batching module is the most 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.

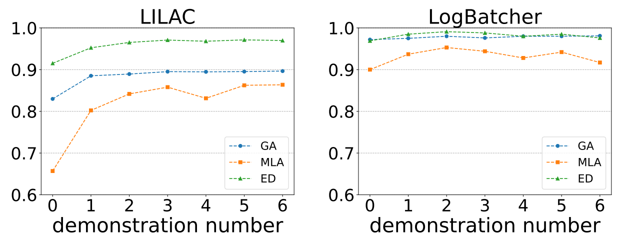


Figure 8: Average accuracy over different numbers of demonstrations

6.3 RQ3: How does LogBatcher perform with with demonstrations?

This RQ is to evaluate how LogBatcher performs with demonstrations. As shown in Figure 8, the demonstrations obtained through the sampling method adopted by [17] indeed help improve performance. For example, the results of LogBatcher in terms of MLA increase by 5.8% when using only two demonstrations. We notice that the performance varies to some extent when the number of demonstrations increases. In contrast, the performance of LogBatcher remains stable and high across different numbers of demonstrations. This is because LogBatcher already achieves good performance in the 0-shot setting, thus not requiring excessive reliance on examples. Furthermore, with the same number of examples, LogBatcher still significantly outperforms LILAC.

6.4 RQ4. How do different settings affect LogBatcher?

6.4.1 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 [17] and meanshift clustering [34]. The results are shown in Table 5. It is clear that LogBatcher achieves the best performance when using the default clustering method (i.e., DBSCAN). It achieves the highest GA, MLA, and ED (i.e., 0.974, 0.904, 0.970, respectively), which are 4.3%, 11.6%, and 2.8% higher than hierarchical clustering, and 3.3%, 4.6%, and 1.0% 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 [18].

Table 4: Results with different clustering methods

	GA	MLA	ED
LogBatcher	0.974	0.904	0.970
w/ Hierarchical clustering	0.934$_{(\downarrow 4.3\%)}$	0.810$_{(\downarrow 11.6\%)}$	0.944$_{(\downarrow 2.8\%)}$
w/ Meanshift clustering	0.943$_{(\downarrow 3.3\%)}$	0.864$_{(\downarrow 4.6\%)}$	0.960$_{(\downarrow 1.0\%)}$

6.4.2 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 [5], 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) [23] 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 5.

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 PA and MLA by 3.9% and 8.8% respectively. The results demonstrate that the diversity-based sampling method used in LogBatcher is effective.

Table 5: Result with different sampling methods

	GA	MLA	ED
LogBatcher	0.974	0.904	0.970
w/ random sampling	0.963$_{(\downarrow 1.1\%)}$	0.890$_{(\downarrow 1.6\%)}$	0.964$_{(\downarrow 0.6\%)}$
w/ similarity sampling	0.937$_{(\downarrow 3.9\%)}$	0.831$_{(\downarrow 8.8\%)}$	0.953$_{(\downarrow 1.8\%)}$

6.4.3 Batch Size. To evaluate the impact of batch size on the performance of LogBatcher , we select the batch size of 1, 5, 10, 15, and 20 (setting the batch size to 1 means removing the batching component). The results are shown in Figure 9. It can be seen that when the batch size approaches 1, the performance drops significantly. The optimal batch size is found to be between 5 and 10. When the batch size exceeds 10, the performance of LogBatcher slightly decreases. Considering larger batch sizes can lead to higher LLM invocation overhead, in our experiments we set the default batch size to 10.

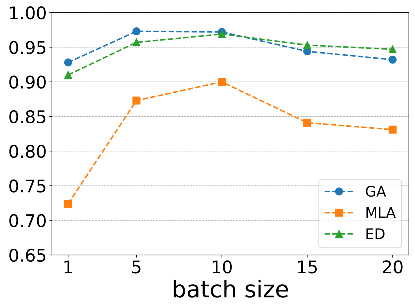


Figure 9: Average accuracy with different batch sizes

6.4.4 LLM Selection. In our experiments, we use ChatGPT (i.e., *GPT-3.5-Turbo*) 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 Codellama 7B and Llama3 70B. Table 7 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 7B, achieving an average GA of 0.936 and MLA of 0.8. 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.

7 DISCUSSION

7.1 Practicality of LogBatcher

LogBatcher is designed for more practical log parsing with Large Language Models (LLMs). Previous work [27, 32] 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

Table 6: Comparison with LILAC on large-scale datasets from Loghub-2.0

		LILAC $_{\text{w/o ICL}}$					**LILAC**					**LogBatcher**				
Dataset	#Log entries	GA	MLA	ED	T_{total}	T_{invoc}	GA	MLA	ED	T_{total}	T_{invoc}	GA	MLA	ED	T_{total}	T_{invoc}
HDFS	11,167,740	**1.000**	0.948	0.999	**5036**	**187**	**1.000**	**0.999**	**1.000**	16128	343	**1.000**	0.948	0.994	10276	214
Hadoop	179,993	0.915	0.840	0.960	42120	183	0.872	**0.832**	**0.947**	77464	335	**0.947**	0.802	0.937	**28685**	**117**
Spark	16,075,117	0.998	0.862	0.996	37027	177	**1.000**	0.973	0.995	76335	321	0.974	**0.997**	**0.999**	**34889**	**129**
Zookeeper	74,273	0.997	0.351	0.891	14270	172	**1.000**	0.687	0.936	22787	253	0.988	**0.934**	**0.995**	**8591**	**101**
BGL	4,631,261	0.884	0.892	0.967	52274	177	**0.894**	**0.958**	**0.989**	88829	261	0.952	0.864	0.957	**47556**	**114**
HPC	429,987	0.984	0.648	0.891	**10447**	**171**	0.869	0.705	0.906	15384	237	**0.990**	**0.983**	**0.993**	15986	188
Thunderbird	16,601,745	0.725	0.556	0.878	901863	197	**0.806**	0.558	**0.928**	491934	354	0.802	**0.564**	0.879	**159588**	**111**
Linux	23,921	0.802	0.693	0.963	60368	171	**0.971**	0.767	0.959	81110	234	0.953	**0.876**	**0.974**	**34056**	**91**
HealthApp	212,394	0.993	0.551	0.811	20709	171	**1.000**	0.729	0.879	30575	251	0.983	**0.970**	**0.989**	**13015**	**81**
Apache	51,977	0.997	0.965	0.981	5349	173	**1.000**	**0.999**	**1.000**	7417	256	0.997	0.972	0.991	**2914**	**91**
Proxifier	21,320	0.000	0.114	0.854	**2681**	**192**	**1.000**	**1.000**	**1.000**	3787	344	**1.000**	**1.000**	**1.000**	3540	322
OpenSSH	638,946	0.745	0.349	0.947	5164	178	0.690	**0.941**	**0.997**	11080	308	**0.925**	0.866	0.946	**3999**	**118**
OpenStack	207,632	**1.000**	0.485	0.944	**9456**	**197**	**1.000**	**1.000**	**1.000**	17915	373	**1.000**	0.980	0.994	14484	315
Mac	100,314	0.769	0.529	0.877	**116508**	197	**0.877**	**0.638**	**0.930**	219922	309	0.795	0.595	0.896	118264	**172**
Average	3,601,187	0.844	0.627	0.926	91662	182	0.927	0.842	0.962	82905	299	**0.950**	**0.882**	**0.967**	**35417**	**155**

Table 7: The performance of LogBatcher with different LLMs

	GA	**MLA**	**ED**
CodeLlama (7B)	0.936	0.800	0.939
Llama3 (70B)	0.925	0.853	0.966
GPT-3.5-Turbo	0.974	0.904	0.970

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 CodeLlama.

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. We also evaluate the applicability of LogBatcher to large-scale datasets. Specifically, we compare LogBatcher with LILAC on 14 large-scale log datasets from Loghub-2.0 [18], with the number of data entries ranging from 21 thousand to 16 million. The results in Table 6 show that LogBatcher achieves the best average PA, MLA, and ED. In addition, it also exhibits the lowest $\mathbf{T}_{\text{total}}$ and $\mathbf{T}_{\text{invoc}}$ on 10 out of 14 datasets. These results are consistent with those on the Loghub-2k datasets, indicating that our method generally incurs less overhead, achieves higher efficiency, and performs better overall.

7.2 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 [17, 55]. 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 [18], which were manually labeled by Zhu et al. [62]. It has been observed that the original labels have some errors and inconsistent labeling styles [21]. To mitigate this threat, we use the datasets corrected by [21]. Furthermore, we also perform experiments on another set of datasets Loghub-2.0 [61]. The results confirm that LogBatcher is effective on both sets of 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.

8 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. 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.

ACKNOWLEDGEMENTS

This work is supported by Australian Research Council (ARC) Discovery Projects (DP200102940, DP220103044).

REFERENCES

[1] 2023. A large collection of system log datasets for AI-powered log analytics. Retrieved August 31, 2023 from https://github.com/logpai/loghub

[2] 2024. OpenAI ChatGPT. Retrieved May 30, 2024 from https://platform.openai.com/docs/models/gpt-3-5-turbo

[3] 2024. Pricing. Retrieved September 15, 2024 from https://openai.com/pricing

[4] Carlos Baquero. 2024. The Energy Footprint of Humans and Large Language Models. arXiv:Communications of the ACM Retrieved Aug 11, 2024 from https://cacm.acm.org/blogcacm/the-energy-footprint-of-humans-and-large-language-models/

[5] Zhoujun Cheng, Jungo Kasai, and Tao Yu. 2023. Batch Prompting: Efficient Inference with Large Language Model APIs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: Industry Track*. 792–810.

[6] Hetong Dai, Heng Li, Che Shao Chen, Weiyi Shang, and Tse-Hsun Chen. 2020. Logram: Efficient log parsing using n-gram dictionaries. *IEEE Transactions on Software Engineering* (2020).

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. 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)*. 4171–4186.

[8] Min Du and Feifei Li. 2016. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 859–864.

[9] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, Vol. 96. 226–231.

[10] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 ninth IEEE international conference on data mining*. IEEE, 149–158.

[11] Jiaxian Guo, Junnan Li, Dongxu Li, Anthony Meng Huat Tiong, Boyang Li, Dacheng Tao, and Steven Hoi. 2023. 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*. 10867–10877.

[12] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 33–40.

[13] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R Lyu. 2021. A survey on automated log analysis for reliability engineering. *ACM computing surveys (CSUR)* 54, 6 (2021), 1–37.

[14] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. 2018. 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*. 60–70.

[15] Dávid Hidvégi, Khashayar Etemadi, Sofia Bobadilla, and Martin Monperrus. 2024. CigaR: Cost-efficient Program Repair with LLMs. *arXiv preprint arXiv:2402.06598* (2024).

[16] Tong Jia, Lin Yang, Pengfei Chen, Ying Li, Fanjing Meng, and Jingmin Xu. 2017. Logsed: Anomaly diagnosis through mining time-weighted control flow graph in logs. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. IEEE, 447–455.

[17] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R. Lyu. 2024. LILAC: Log Parsing using LLMs with Adaptive Parsing Cache. arXiv:2310.01796 [cs.SE]

[18] Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jiazhen Gu, Zhuangbin Chen, Jieming Zhu, and Michael R Lyu. 2024. 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*.

[19] Zhen Ming Jiang, Ahmed E Hassan, Parminder Flora, and Gilbert Hamann. 2008. Abstracting execution logs to execution events for enterprise applications (short paper). In *2008 The Eighth International Conference on Quality Software*. IEEE, 181–186.

[20] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, Mark D Syer, and Ahmed E Hassan. 2018. Examining the stability of logging statements. *Empirical Software Engineering* 23, 1 (2018), 290–333.

[21] Zanis Ali Khan, Donghwan Shin, Domenico Bianculli, and Lionel Briand. 2022. Guidelines for assessing the accuracy of log message template identification techniques. In *Proceedings of the 44th International Conference on Software Engineering*. 1095–1106.

[22] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2023. Decomposed Prompting: A Modular Approach for Solving Complex Tasks. In *The Eleventh International Conference on Learning Representations*.

[23] Alex Kulesza, Ben Taskar, et al. 2012. Determinantal point processes for machine learning. *Foundations and Trends® in Machine Learning* 5, 2–3 (2012), 123–286.

[24] Van-Hoang Le and Hongyu Zhang. 2021. Log-based Anomaly Detection Without Log Parsing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 492–504.

[25] Van-Hoang Le and Hongyu Zhang. 2022. Log-based anomaly detection with deep learning: How far are we?. In *Proceedings of the 44th International Conference on Software Engineering*. 1356–1367.

[26] Van-Hoang Le and Hongyu Zhang. 2023. Log Parsing: How Far Can ChatGPT Go?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1699–1704.

[27] Van-Hoang Le and Hongyu Zhang. 2023. Log parsing with prompt-based few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2438–2449.

[28] Van-Hoang Le and Hongyu Zhang. 2024. PreLog: A Pre-trained Model for Log Analytics. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–28.

[29] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. 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*. 7871–7880.

[30] Xiaoyun Li, Hongyu Zhang, Van-Hoang Le, and Pengfei Chen. 2024. Logshrink: Effective log compression by leveraging commonality and variability of log data. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[31] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.

[32] Yudong Liu, Xu Zhang, Shilin He, Hongyu Zhang, Liqun Li, Yu Kang, Yong Xu, Minghua Ma, Qingwei Lin, Yingnong Dang, et al. 2022. Uniparser: A unified log parser for heterogeneous log data. In *Proceedings of the ACM Web Conference 2022*. 1893–1901.

[33] Allan Lyons, Julien Gamba, Austin Shawaga, Joel Reardon, Juan Tapiador, Serge Egelman, and Narseo Vallina-Rodríguez. 2023. 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)*. 2115–2132.

[34] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2024. 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. https://doi.org/10.1145/3597503.3639150

[35] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. 2009. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1255–1264.

[36] Andres Marzal and Enrique Vidal. 1993. Computation of normalized edit distance and applications. *IEEE transactions on pattern analysis and machine intelligence* 15, 9 (1993), 926–932.

[37] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael Rung-Tsong Lyu, and Hua Cai. 2013. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems* 24, 6 (2013), 1245–1255.

[38] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. 2019. 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*. 1795–1812.

[39] Meiyappan Nagappan and Mladen A Vouk. 2010. 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, 114–117.

[40] Meiyappan Nagappan, Kesheng Wu, and Mladen A Vouk. 2009. Efficiently extracting operational profiles from execution logs using suffix arrays. In *2009 20th International Symposium on Software Reliability Engineering*. IEEE, 41–50.

[41] Sasho Nedelkoski, Jasmin Bogatinovski, Alexander Acker, Jorge Cardoso, and Odej Kao. 2020. Self-attentive classification-based anomaly detection in unstructured logs. In *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE, 1196–1201.

[42] OpenAI. 2023. GPT-4 Technical Report. *ArXiv* abs/2303.08774 (2023).

[43] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H Chin, and Sumayah Alrwais. 2015. 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, 45–56.

[44] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[45] Stefan Petrescu, Floris Den Hengst, Alexandru Uta, and Jan S Rellermeyer. 2023. Log parsing evaluation in the era of modern software systems. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 379–390.

[46] Gerard Salton and Christopher Buckley. 1988. Term-weighting approaches in automatic text retrieval. *Information processing & management* 24, 5 (1988), 513–523.

[47] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. *ACM Transactions on Database Systems (TODS)* 42, 3 (2017), 1–21.

[48] Zhenwei Shao, Zhou Yu, Meng Wang, and Jun Yu. 2023. 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*. 14974–14983.

[49] Keiichi Shima. 2016. Length matters: Clustering system log messages using length of words. *arXiv preprint arXiv:1611.03213* (2016).

[50] Liang Tang, Tao Li, and Chang-Shing Perng. 2011. LogSig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*. 785–794.

[51] Risto Vaarandi and Mauno Pihelgas. 2015. Logcluster-a data clustering and pattern mining algorithm for event logs. In *2015 11th International conference on network and service management (CNSM)*. IEEE, 1–7.

[52] Xuheng Wang, Xu Zhang, Liqun Li, Shilin He, Hongyu Zhang, Yudong Liu, Lingling Zheng, Yu Kang, Qingwei Lin, Yingnong Dang, et al. 2022. 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*. 1198–1208.

[53] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[54] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[55] Junjielong Xu, Ruichun Yang, Yintong Huo, Chengyu Zhang, and Pinjia He. 2024. DivLog: Log Parsing with Prompt Enhanced In-Context Learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[56] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 117–132.

[57] Xiao Yi, Van-Hoang Le, and Hongyu Zhang. 2024. LogBatcher repository. Retrieved Sep 11, 2024 from https://github.com/LogIntelligence/LogBatcher

[58] Siyu Yu, Pinjia He, Ningjiang Chen, and Yifan Wu. 2023. Brain: Log parsing with bidirectional parallel tree. *IEEE Transactions on Services Computing* (2023).

[59] Wei Zhang, Hongcheng Guo, Anjie Le, Jian Yang, Jiaheng Liu, Zhoujun Li, Tieqiao Zheng, Shi Xu, Runqiang Zang, Liangfan Zheng, and Bo Zhang. 2024. Lemur: Log Parsing with Entropy Sampling and Chain-of-Thought Merging. arXiv:2402.18205 [cs.SE]

[60] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. 2019. 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*. 807–817.

[61] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R. Lyu. 2023. Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics. arXiv:2008.06448 [cs.SE]

[62] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. 2019. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 121–130.