LILAC: Log Parsing using LLMs with Adaptive Parsing Cache*

ZHIHAN JIANG, The Chinese University of Hong Kong, China
JINYANG LIU, The Chinese University of Hong Kong, China
ZHUANGBIN CHEN, Sun Yat-sen University, China
YICHEN LI, The Chinese University of Hong Kong, China
JUNJIE HUANG, The Chinese University of Hong Kong, China
YINTONG HUO, The Chinese University of Hong Kong, China
PINJIA HE, The Chinese University of Hong Kong, China
JIAZHEN GU, The Chinese University of Hong Kong, China
MICHAEL R. LYU, The Chinese University of Hong Kong, China

Log parsing transforms log messages into structured formats, serving as the prerequisite step for various log analysis tasks. Although a variety of log parsing approaches have been proposed, their performance on complicated log data remains compromised due to the use of human-crafted rules or learning-based models with limited training data. The recent emergence of powerful large language models (LLMs) demonstrates their vast pre-trained knowledge related to code and logging, making it promising to apply LLMs for log parsing. However, their lack of specialized log parsing capabilities currently hinders their parsing accuracy. Moreover, the inherent inconsistent answers, as well as the substantial overhead, prevent the practical adoption of LLM-based log parsing.

To address these challenges, we propose LILAC, the first practical Log parsIng framework using LLMs with Adaptive parsing Cache. To facilitate accurate and robust log parsing, LILAC leverages the in-context learning (ICL) capability of the LLM by performing a hierarchical candidate sampling algorithm and selecting high-quality demonstrations. Furthermore, LILAC incorporates a novel component, an adaptive parsing cache, to store and refine the templates generated by the LLM. It helps mitigate LLM's inefficiency issue by enabling rapid retrieval of previously processed log templates. In this process, LILAC adaptively updates the templates within the parsing cache to ensure the consistency of parsed results. The extensive evaluation on public large-scale datasets shows that LILAC outperforms state-of-the-art methods by 69.5% in terms of the average F1 score of template accuracy. In addition, LILAC reduces the query times to LLMs by several orders of magnitude, achieving a comparable efficiency to the fastest baseline.

CCS Concepts: • **Software and its engineering → Software creation and management**.

Additional Key Words and Phrases: log parsing, log analysis, large language models

*Jiazhen Gu is the corresponding author.

Authors' addresses: Zhihan Jiang, The Chinese University of Hong Kong, Hong Kong, China, zhjiang22@cse.cuhk.edu.hk; Jinyang Liu, The Chinese University of Hong Kong, Hong Kong, China, jyliu@cse.cuhk.edu.hk; Zhuangbin Chen, Sun Yat-sen University, Zhuhai, China, chenzhb36@mail.sysu.edu.cn; Yichen Li, The Chinese University of Hong Kong, Hong Kong, China, ycli21@cse.cuhk.edu.hk; Junjie Huang, The Chinese University of Hong Kong, Hong Kong, China, junjayhuang@outlook.com; Yintong Huo, The Chinese University of Hong Kong, Hong Kong, China, ythuo@cse.cuhk.edu.hk; Pinjia He, The Chinese University of Hong Kong, Shenzhen, China, hepinjia@cuhk.edu.cn; Jiazhen Gu, The Chinese University of Hong Kong, Hong Kong, China, jiazhengu@cuhk.edu.hk; Michael R. Lyu, The Chinese University of Hong Kong, Hong Kong, China, lyu@cse.cuhk.edu.hk.

ACM 2994-970X/2024/7-ART7
https://doi.org/10.1145/3643733

ACM Reference Format:
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. *Proc. ACM Softw. Eng.* 1, FSE, Article 7 (July 2024), 23 pages. https://doi.org/10.1145/3643733

1 INTRODUCTION

Log messages are generated by logging statements in the source code to record the system events and statuses at runtime. Modern software systems produce a large volume of log data [Wang et al. 2022; Yao et al. 2021], facilitating various downstream tasks, such as anomaly detection [Ali et al. 2023; Liu et al. 2023a; Zhang et al. 2022, 2019; Zhao et al. 2021b], failure troubleshooting [Chen et al. 2021; Xu et al. 2009] and root cause analysis [Amar and Rigby 2019; Notaro et al. 2023; Wang et al. 2020]. As such, log analysis plays an essential role in the maintenance of software systems. Log parsing is the first and foremost step in log analysis, which extracts two parts of log messages: 1) *log templates* - constant parts that are explicitly written in logging statements; 2) *log parameters* - dynamic parts that are changeable in different executions. For example, a logging statement "`logging.info(f"Starting reading data from {file_path}")`" can generate a sequence of log messages with different `file_path`, such as "`Starting reading data from /etc/data/`". In the above example, the log template is "`Starting reading data from <*>`", and the log parameter indicates the path of data, *i.e.*, "`/etc/data/`".

Since the source code is generally inaccessible during system maintenance, a wide range of techniques (*i.e.*, log parsers) [He et al. 2017; Le and Zhang 2023b; Nagappan and Vouk 2010; Vaarandi 2003] have been proposed to distinguish the templates and parameters from log messages automatically. Existing log parsers can be categorized into two groups: *syntax-based* and *semantic-based*. Syntax-based log parsers [Dai et al. 2020; Du and Li 2016; He et al. 2017; Yu et al. 2023b] utilize specific features or heuristics (*e.g.*, log length, word length and frequency) to extract the constant parts of log messages as templates. In contrast, semantic-based log parsers [Huo et al. 2023; Le and Zhang 2022; Li et al. 2023b; Liu et al. 2022] employ deep learning models to learn semantics and system-specific patterns from labeled log data so as to parse new log messages.

Unfortunately, recent benchmark studies [Jiang et al. 2023; Khan et al. 2022; Petrescu et al. 2023] have revealed that the performance of existing log parsers in practice remains unsatisfactory. On the one hand, syntax-based log parsers heavily rely on crafted rules, while a significant performance degradation would happen when the log data deviate from the established rules. On the other hand, the deep learning models adopted by semantic-based log parsers are trained by limited labeled log messages. When parsing more complicated log messages that have different features from the training data, the models may fail to understand semantics and extract templates.

To address these limitations, we propose to leverage the powerful large language models (LLMs) to achieve effective log parsing. LLMs are trained by vast amounts of text data related to code [Peng et al. 2023b; Yang et al. 2023] and logging [Li et al. 2023a; Mastropaolo et al. 2022], thus having the potential to understand log messages comprehensively. For example, when processing a log message "`Process f3e2 write to /etc/smartd.conf failed.`", the LLM can accurately discern that "`f3e2`" and "`/etc/smartd.conf`" are parameters recording the process ID and the file path. Moreover, this process does not require manually designed rules (*e.g.*, regular expressions and delimiters), which makes LLMs promising components for log parsing. However, designing a practical LLM-based log parsing approach still faces the following challenges:

(1) Lack of specialized capability. LLMs are not specialized in log parsing. Although LLMs have a wealth of general knowledge through pre-training, they are not fine-tuned (*e.g.*, instruction tuning [Wei et al. 2021] and reinforcement learning with human feedback [MacGlashan et al. 2017])

for the log parsing task. Hence, the performance of directly querying LLMs to parse log messages may be compromised [Le and Zhang 2023a; Mudgal and Wouhaybi 2023].

(2) Inconsistent outputs of LLMs. As revealed by recent studies [Du et al. 2023; Mudgal and Wouhaybi 2023; Mündler et al. 2023; Peng et al. 2023a], LLMs may produce unstable outputs. In terms of log parsing, LLMs may generate different templates for log messages with the same template. This inconsistency will lead to a decline in grouping accuracy, a critical factor for certain downstream tasks, such as log compression [Li et al. 2023c; Rodrigues et al. 2021] and anomaly detection [Le and Zhang 2022; Zhang et al. 2019].

(3) Huge overhead of employing LLMs. LLMs have billions of weights and require huge computing resources (*e.g.*, GPUs) for inference. Therefore, compared to traditional parsing tools, the overhead of querying LLMs (*e.g.*, inference time and network latency) is notably high [Dettmers et al. 2022; Wang et al. 2023a]. Considering that modern software systems can produce tens of gigabytes of logs per hour [Li et al. 2023c; Wang et al. 2022; Zhu et al. 2019], directly employing LLMs for log parsing is impractical.

To tackle the aforementioned challenges, we propose LILAC, a **L**og pars**I**ng framework using **L**LMs with **A**daptive parsing **C**ache. LILAC consists of two main components, the *ICL-enhanced parser* and the adaptive *parsing cache*. The ICL-enhanced parser is designed to accurately parse queried log messages, while the parsing cache stores and adaptively refines the generated templates to ensure both efficiency and consistency. In particular, the ICL-enhanced parser leverages the in-context learning (ICL) capability to adapt LLMs to parse diverse log data. It first obtains high-quality demonstrations using the proposed effective and efficient *candidate sampling* and *demonstration selection* algorithms, and then utilizes the designed prompt format to guide the LLM to parse log messages accurately. The design of the parsing cache targets to address the issues of inconsistent outputs and huge overhead associated with LLMs. By prioritizing the cache matching operation, LILAC can avoid duplicated queries to LLMs, thereby enhancing the parsing efficiency. Moreover, the cache updating operation can adaptively refine the potential erroneous templates within the parsing cache to mitigate the inconsistency of LLMs.

We have conducted a comprehensive evaluation on public large-scale log datasets of Loghub-2.0 [Jiang et al. 2023] from the LogPAI team [Zhu et al. 2019]. The results show that LILAC achieves the highest average accuracy on all performance metrics, outperforming state-of-the-art baselines by 66.8% and 69.5% for the F1 score of grouping and template accuracy, respectively. Furthermore, LILAC exhibits remarkable robustness across diverse log datasets, consistently maintaining high performance when integrated with various language models. With regards to efficiency, LILAC has achieved a speed comparable to the most efficient baseline, Drain [He et al. 2017], significantly reducing the overhead of querying LLMs.

The main contributions of this work are summarized as follows:

- To the best of our knowledge, we propose the first practical LLM-based log parsing framework named LILAC. With effective and efficient candidate sampling and demonstration selection algorithms, LILAC exploits the ICL capability of LLMs, enabling accurate and robust log parsing.
- We introduce an adaptive parsing cache and design cache operations to mitigate the inefficiency and instability issues associated with the application of LLMs for log parsing.
- We extensively evaluate LILAC on public large-scale datasets. The results show that LILAC outperforms state-of-the-art methods in terms of accuracy while also achieving high efficiency.
- The source code of LILAC is publicly available at https://github.com/logpai/LILAC to benefit both practitioners and researchers in the field of log analysis.

2 BACKGROUND AND MOTIVATION

2.1 Log Parsing

Log parsing aims to convert semi-structured log messages into structured data, *i.e.*, extracting both the constant parts (*i.e.*, log templates) and the dynamic parts (*i.e.*, log parameters) from log messages. A straightforward method involves matching raw log messages with corresponding logging statements within the source code [Pecchia et al. 2015; Schipper et al. 2019]. However, this strategy is impractical when the source code is inaccessible, such as commercial software. Consequently, a variety of data-driven log parsers without requiring access to the source code have been proposed in the literature [He et al. 2017; Jiang et al. 2008; Yu et al. 2023a]. These log parsers can be categorized into two groups: syntax-based ones and semantic-based ones.

Unfortunately, recent studies have underscored that existing log parsers struggle when handling diverse log data [Jiang et al. 2023; Khan et al. 2022; Petrescu et al. 2023]. On the one hand, syntax-based log parsers heavily rely on pre-designed features and rules (*e.g.*, regular expressions), requiring a substantial amount of domain-specific knowledge. This limitation leads to a compromised performance when processing log data that does not adhere to these established rules. For instance, Drain [He et al. 2017], a leading syntax-based log parser, employs heuristics based on the assumption that all log parameters within specific templates possess an identical number of tokens. This assumption can lead to errors in the parsed templates when the parameter length exhibits flexibility. On the other hand, semantic-based log parsers typically adopt deep learning models to utilize the semantics within log messages. Hence, they are inherently limited by the quantity and quality of labeled data available for model training or tuning. This limitation can lead to a substantial degradation in their performance when processing complex and large-scale log data [Jiang et al. 2023; Xu et al. 2023b]. Additionally, the log messages generated in production systems are continually evolving, resulting in ever-changing characteristics of log data [Wang et al. 2022; Xu et al. 2023b]. This evolution may render training-based or tuning-based methods non-adaptive to the changes in log data, subsequently leading to unsatisfying practical performance.

2.2 Large Language Models

Large Language Models (LLMs) have demonstrated remarkable performance in the field of natural language processing. These models generally adopt the Transformer [Vaswani et al. 2017] architecture and are trained on extensive corpora using self-supervised objectives. LLMs are characterized by their large sizes, *e.g.*, the standard GPT-3 model [Brown et al. 2020] has 175 billion parameters. Recently, many studies (*e.g.*, SPINE [Wang et al. 2022] and Hue [Xu et al. 2023a]) have introduced the "human-in-the-loop" concept, indicating the need for external knowledge for effective log analysis. Given that LLMs already possess a substantial amount of pre-trained knowledge, it is promising to utilize LLMs for log parsing.

However, how to effectively apply LLMs to downstream tasks has emerged as a vital research topic. A common approach involves fine-tuning the model and updating the parameters using specific downstream datasets. Nonetheless, this method demands considerable computational resources and high-quality data, making it less feasible in specific scenarios. In contrast, *in-context learning* (ICL) presents an innovative alternative to utilize LLMs to perform downstream tasks [Dong et al. 2022; Liu et al. 2023c]. Specifically, in the ICL paradigm, the prompt to query LLMs typically comprises three parts: (1) *Instruction*: description of the specific task; (2) *Demonstrations*: several examples, *i.e.*, pairs of queries and corresponding ground-truth answers; (3) *Query*: the query that requires an answer from LLMs. Such a prompt can let LLMs gain task-specific knowledge by learning the input-output relationship of the task. Recent studies have demonstrated that ICL can aid LLMs in achieving remarkable performance in a variety of tasks such as logic reasoning [Wei et al. 2022]

and fact retrieval [He et al. 2022]. Therefore, in this paper, we intend to adopt LLMs with the ICL paradigm to achieve effective log parsing.

2.3 Challenges of Log Parsing with LLMs

Although utilizing LLMs for log parsing presents significant potential and some recent work [Le and Zhang 2023a; Liu et al. 2023b; Xu et al. 2023b] has investigated the LLM-based log parsing, these studies fall short in addressing the following three critical challenges, which prevent their practical adoption.

- *Specialization.* Although LLMs are imbued with a large volume of pre-trained knowledge, they are not specialized in the log parsing task. As a result, directly querying LLMs to perform log parsing could potentially result in a compromised performance. The ICL paradigm can facilitate the adoption of LLMs to log parsing without tuning. Specifically, the demonstrations within the prompt can impart task-specific knowledge to LLMs by the correlation between input and output. In practice, the initial phase of ICL involves sampling a small set of candidate log messages, from which the demonstrations for each query will be selected. Given the huge volumes and imbalanced frequencies [Jiang et al. 2023; Khan et al. 2022; Wang et al. 2022] of logs in real-world systems, it is quite challenging to select diverse candidate log messages for effective ICL. Though some sampling algorithms exist for few-shot log parsing [Le and Zhang 2023b; Xu et al. 2023b], all of them require pairwise computation between log messages or adopt random sampling, which can hardly choose diverse candidates efficiently. Hence, the efficient sampling of a set of diverse candidate log messages to enable effective ICL still presents a challenge.
- *Consistency.* Despite the strong capabilities of understanding and generating texts, LLMs may produce unstable answers, which has been identified and discussed in recent studies [Mündler et al. 2023; Peng et al. 2023a; Zheng et al. 2023]. Furthermore, due to the limitation of LLMs in parsing based solely on the semantics within a single query, they may exhibit inconsistency in determining whether a particular token is a parameter. These may lead LLMs to produce templates that are either more broad or more specific when parsing two log messages that share the same template but have distinct parameter values. For example, when parsing two distinct log messages, "`User root failed to kill the process 0xF28A`" and "`User user1 failed to kill the process 0x6C37`" individually, inconsistency may arise in the answers of the LLM. Specifically, the LLM may identify "`root`" in the first log message as a constant token while identifying "`user1`" in the second log message as a parameter. These inconsistent templates can precipitate a decrease in grouping accuracy, which would impact downstream tasks such as log compression and anomaly detection. Therefore, mitigating the inconsistency of LLMs to generate consistent log templates is yet to be resolved.
- *Efficiency.* Given that real-world systems generate substantial volumes of log data, *e.g.*, tens of gigabytes per hour [Jiang et al. 2023; Wang et al. 2022; Zhu et al. 2019], log parsers should process high volumes of data efficiently, *e.g.*, millions of log messages per minute. Therefore, efficiency is a critical aspect of practical log parsers. Since LLMs have billions of weights and require extensive resources (*e.g.*, high-performance GPUs) for inference, they are typically deployed on high-performance servers and provide query interfaces. Compared to traditional local-deployed parsing tools, utilizing LLMs inevitably introduces considerable overhead, including inference time and network latency [Dettmers et al. 2022; Wang et al. 2023a]. Existing work [Le and Zhang 2022, 2023a; Xu et al. 2023b] employs LLMs or other language models to process each log message individually, which is hard to meet the practical efficiency demands [Jiang et al. 2023; Mudgal and Wouhaybi 2023]. Consequently, how to achieve efficient LLM-based log parsing remains a challenge to be addressed.

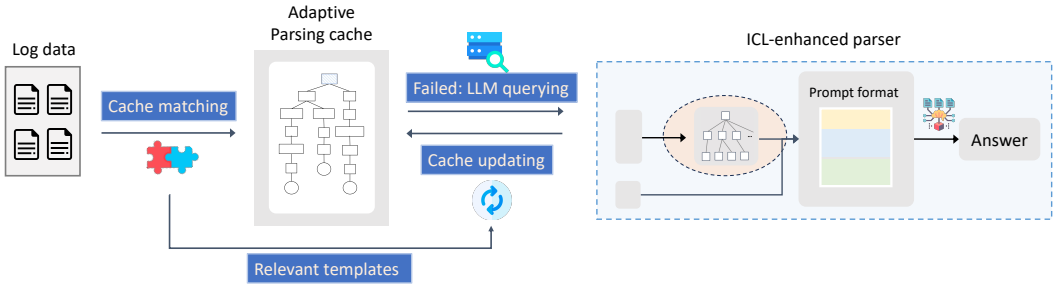

Fig. 1. The overall workflow of LILAC.

3 METHOD

3.1 Overview

In this section, we introduce LILAC, a log parsing framework using LLMs with adaptive parsing cache. LILAC consists of two components: the *ICL-enhanced parser* and the adaptive *parsing cache*. To specialize the LLM in log parsing and adapt it to various log data, the ICL-enhanced parser utilizes the ICL paradigm. Specifically, an efficient candidate sampling algorithm is performed to choose a candidate set of diverse log messages, from which effective demonstrations can be selected for the LLM. To address the inefficiency associated with the utilization of LLMs, LILAC introduces a novel component, the parsing cache, to store the parsed templates. Such a design is motivated by the following observation: The number of log templates is several orders of magnitude smaller than the number of log messages in real-world systems [Jiang et al. 2023; Liu et al. 2019; Wang et al. 2022]. For instance, the datasets in Loghub-2.0 [Jiang et al. 2023] contain over 50 million log messages, yet the total number of log templates is fewer than 3,500. Hence, through caching and matching the parsed log templates, LILAC can avoid duplicate LLM queries and significantly improve the parsing efficiency. Moreover, to ensure the consistency of the parsed results, LILAC adaptively refines the stored log templates within parsing cache based on the newly generated templates from the ICL-enhanced parser.

Fig. 1 overviews the workflow of LILAC. For each log message to parse, LILAC first performs the cache matching operation to check whether its corresponding template is already stored in the parsing cache. If so, LILAC directly used the matched template as the parsed result of this log message, thereby preventing duplicate queries of the LLM. Otherwise, the cache matching operation will retrieve several relevant templates from the parsing cache, which exhibit a high degree of correlation with the input log message. Since these relevant templates may be erroneous templates caused by mistakes of the LLM, LILAC will record them for the subsequent adaptive updating. Then, the ICL-enhanced parser selects high-quality demonstrations from the sampled candidate set to form the designed prompt. This prompt is then used to query the LLM to extract the template for this log. Based on both the generated template and its relevant templates, in the cache updating operation, LILAC will adaptively determine whether to insert the generated template as a new template to the parsing cache, or to refine an existing relevant template to achieve more precise and consistent parsed results.

3.2 ICL-enhanced parser

Fig. 2 presents the overall design of the ICL-enhanced parser adopted by LILAC. It employs the ICL paradigm to adapt the LLM to log parsing task without resource-intensive model tuning. Furthermore, it leverages system-specific features within demonstrations to facilitate more accurate log parsing. In particular, we propose an effective and efficient *candidate sampling* algorithm,

along with a *demonstration selection* algorithm, to obtain high-quality examples for effective ICL. Specifically, LILAC first performs the hierarchical candidate sampling algorithm to sample a small set of diverse and representative candidate log messages. During the online parsing, for each queried log, LILAC utilizes the KNN-based demonstration selection algorithm to choose similar demonstration examples. These demonstrations are integrated into the prompt following the designed format. Lastly, the ICL-enhanced parser inputs the prompt to the LLM and obtains the generated templates.

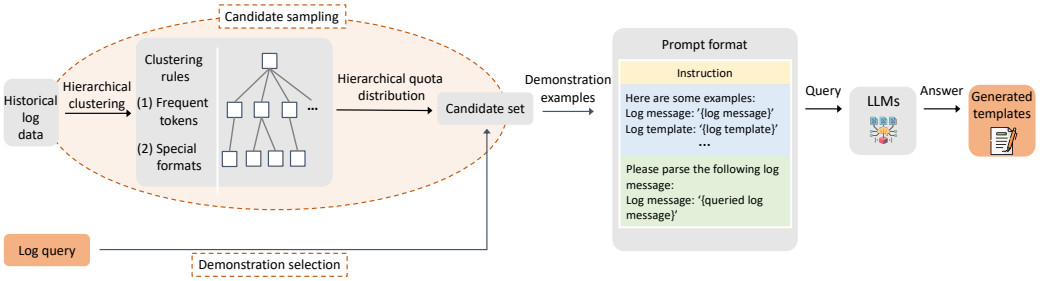

Fig. 2. The workflow of ICL-enhanced parser.

3.2.1 Candidate Sampling. A typical application of the ICL paradigm involves initially sampling a small set of candidate log messages from produced log data in the system. It is crucial to ensure that the candidate set is diverse to mitigate the potential risk of inductive bias [Le and Zhang 2023b; Xu et al. 2023b], since disproportionate demonstrations could cause the LLM to overfit to a specific example. Furthermore, these candidates ought to be representative, *i.e.*, they should be capable of representing more log messages within the log data to provide the LLM with more system-specific characteristics. In real-world applications, this sampling procedure can initially be performed on the historical log data and be executed periodically to maintain a dynamic candidate set, thereby facilitating adaptation to the continuous evolution of log data. Therefore, the efficiency of this sampling process is essential.

In the ICL-enhanced parser, we propose a hierarchical sampling algorithm to extract a small, diverse, and representative set of log messages from substantial log data, as illustrated in the left part of Fig. 2. This algorithm consists of two phases, *hierarchical clustering* and *hierarchical quota distribution*. It first groups the entire log data into hierarchical clusters based on the characteristics of log messages. Each cluster encompasses log messages with highly similar features, whereas log messages within different clusters exhibit divergent characteristics. Then, a hierarchical quota distribution approach is performed to select candidates from different clusters while assigning distinct priorities to each cluster based on its number of log messages.

Hierarchical clustering. Inspired by previous research [Jiang et al. 2023; Liu et al. 2019; Nagappan and Vouk 2010], we first utilize the top-K frequent tokens to group log messages. The intuition is that log messages that share the same frequent tokens are more likely to have the same templates. Specifically, we first tokenize each log message and then calculate all token frequencies. During the above process, stop words in the Scipy library [sci 2023] are excluded to eliminate irrelevant tokens. For each log message, tokens with top-K frequencies are selected, which form the basis for their categorization into different *coarse-grained clusters*. In other words, all log messages within the same coarse-grained clusters share the same top-K frequent tokens.

However, solely utilizing frequent tokens is insufficient to differentiate log messages with varying characteristics, *i.e.*, log messages that share the same top-K frequent tokens may correspond to different log templates. Thus, we leverage the special characters (*i.e.*, characters that are not

alphabets, numerals, or white space) to reflect the features of log messages, defining the set of special characters in a log message as its *special format*. Log messages originating from the same template typically share an identical special format. This is because the special characters in the constant parts (*i.e.*, the template) of a log message are invariably identical, and those in the dynamic parts (*i.e.*, the parameter) are generally congruent. For instance, the special format of "`Received block: blk_358 of size 6710 from /127.0.0.1`" is `{':','_','.','/'}`. For other log messages that share the same template, such as "`Received block: blk_729 of size 8199 from /127.0.0.2`", they would have an identical special format. Therefore, we use the special formats of log messages to perform fine-grained clustering. In detail, log messages in each coarse-grained cluster are further divided according to their special formats and constitute *fine-grained clusters*, wherein all log messages not only have identical top-K frequent tokens but also share the same log format.

Hierarchical quota distribution. In this phase, we aim to choose diverse and representative log messages as candidates from the fine-grained clusters. The core idea is to hierarchically distribute the quota of K_s candidates across all fine-grained clusters as evenly as possible to enhance diversity. Moreover, we assume that clusters with a larger number of log messages are more representative. Hence, in situations where achieving an equitable distribution is unattainable, priority is given to fine-grained clusters with more log messages.

Initially, we distribute the quota of K_s candidates uniformly across all coarse-grained clusters. Subsequently, within each coarse-grained cluster, we arrange all fine-grained clusters in descending order based on their priorities determined by the number of log messages they contain. Given a coarse-grained cluster that has been allocated K_c quotas and contains n sorted fine-grained clusters, denoted as $\{f_1, f_2, \cdots, f_n\}$, the quota assigned to cluster f_i is as follows:

$$S(f_i) = \begin{cases} \lfloor \frac{K_c}{n} \rfloor + 1 & \text{if } i \leq (K_c \bmod n) \\ \lfloor \frac{K_c}{n} \rfloor & \text{otherwise} \end{cases}$$

Recall that the design of the ICL-enhanced parser in LILAC is intended to leverage the ICL paradigm in few-shot scenarios, which suggests that the number of sampled candidates, K_s, is typically small. This means, in most cases, the number of fine-grained clusters surpasses the number of sampled candidates, *i.e.*, $N_f > K_s$. Hence, the quota allocation for each fine-grained cluster is also typically small (*e.g.*, 0 to 2). Lastly, we randomly select the assigned number of candidate log messages within each fine-grained cluster to ensure high efficiency.

3.2.2 Demonstration Selection. During the parsing process, to mitigate the interference of irrelevant information and enhance log parsing accuracy, we need to further select k demonstration examples from the K_s candidates to construct the prompt for ICL. These demonstration examples should exhibit similarity to the queried log message, providing the LLM similar patterns and semantics within the examples to parse this log message accurately [Gao et al. 2023; Xu et al. 2023b].

LILAC adopts k-Nearest Neighbors (kNN), a simple yet effective algorithm to select demonstration examples. For each queried log message l, we compute the similarities between it and all candidate log messages, *i.e.*, $sim(l, s_i)$, $i \in [1, K_s]$. We propose to measure the similarity between two log messages based on both tokens and special formats. In specific, given a log message l, we extract the characters of the tokens that are derived from l and the special characters within l, to form the *feature set* of l, *i.e.*, $F(l)$. Based on this, we can calculate the value of $sim(\cdot)$ of two log messages by using the Jaccard similarity [jac 2023] of their feature sets, *i.e.*, $sim(l_1, l_2) = \frac{|F(l_1) \cap F(l_2)|}{|F(l_1) \cup F(l_2)|}$. After computing all similarities, we select log messages from the candidate sets that exhibit the top-k highest similarities. These log messages, characterized by similar tokens and special characters to the queried log, are instrumental in aiding the LLM to comprehend the semantics and formats embedded within them.

3.2.3 Query Design. Following previous work [Le and Zhang 2023a; Xu et al. 2023b], we design and use the prompt format, as depicted in Fig. 3, to query the LLM to generate the log template for an individual log message. Specifically, the prompt encompasses the following three parts.

(1) *Instruction.* To provide the LLM with more task-specific information, we employ an instruction that briefly introduces the task, the concept of log parsing, and the formats of input and output.
(2) *Demonstration Examples.* Subsequently, we integrate several demonstrations, chosen by the demonstration selection algorithm, into the prompt. Each demonstration includes a pair of one log message and its log template. Since recent work [Gao et al. 2023; Zhao et al. 2021a] has pinpointed that LLMs with ICL are more prone to be influenced by the examples that are closer to the query, we arrange the demonstration examples in *ascending order* of similarity to the queried log, *i.e.*, those of higher similarities closer to the queried log. This is based on the intuition that examples with higher similarity may encompass more information pertinent to parsing the queried log.
(3) *Queried Log.* Last, we present the content of the log message to query the LLM.

Guided by the instruction and demonstration examples, the LLM could more precisely answer the log template of the queried log in the prompt, adhering to the correct format.

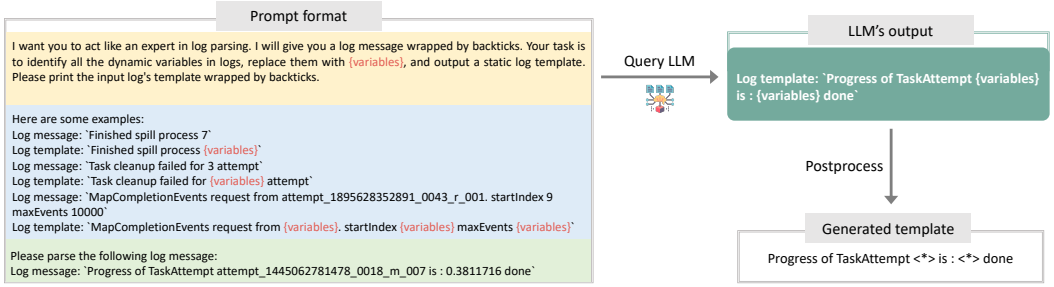

Fig. 3. The demonstration of our prompt design.

3.3 Adaptive Parsing Cache

The adaptive parsing cache is designed to guarantee the efficiency and consistency of LILAC. Specifically, LILAC adopts a tree structure to store the generated templates of the LLM, serving as the parsing cache. The left part of Fig. 4 demonstrates an example of parsing cache, which stores three log templates. In the parsing cache, all generated log templates are tokenized into a list of tokens, which are stored in the tree from top to bottom. Each intermediate node in the tree represents a token, with the "<*>" denoting the wildcard token that can match any length of tokens. Each leaf node of the tree represents a unique log template, which corresponds to the string obtained by concatenating all tokens contained in all intermediate nodes on the unique path from the root node to the leaf node. This tree structure allows for efficient storage and parallel retrieval of log templates. To retrieve a specific template, only one single traversal from the root to the leaf node is required, without the necessity to check each template sequentially (Sec. 3.3.1). Moreover, the tree structure can directly reflect the similarity among log templates, *i.e.*, templates within the same subtree share a common prefix. This can aid in filtering relevant templates of specific log messages, which will be further used for cache updating operation (Sec. 3.3.2).

Based on the tree structure of parsing cache, we further design two cache operations, *i.e.*, *cache matching* and *cache updating*. The cache matching operation is used to determine whether the template of the input log message has already been stored in parsing cache. The cache updating operation is designed to adaptively update templates stored in the parsing cache when the cache

matching fails and a new template is generated by the ICL-enhanced parser. Next, we illustrate the design details of these cache operations.

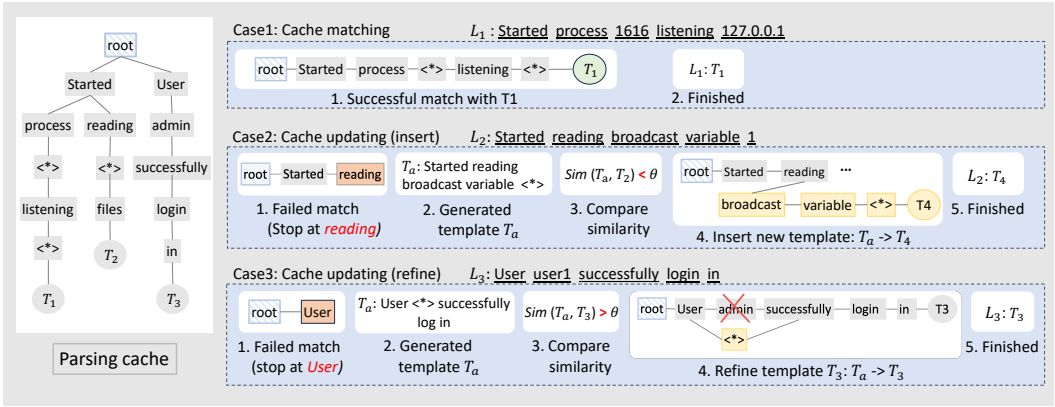

Fig. 4. The demonstration cases of cache matching and updating operations for the parsing cache.

3.3.1 Cache Matching. Given a new log message, LILAC first checks whether the corresponding template has been stored in the parsing cache through the cache matching operation, which can reduce duplicate queries to LLM and improve parsing efficiency. To match the input log message with the parsing cache, we first split the log content into a series of tokens by delimiters. Then, these tokens are read sequentially from the first to the last, with each being compared to intermediate nodes within the tree structure of parsing cache. Specifically, for the initial token, a search is conducted to verify its presence in the second layer of the tree since the first layer is the empty root node. If a match is found for the first token, the process continues with the second token and the children of the matched node. This procedure persists until all tokens have been read or until no further tokens can be matched. It is worth noting that the wildcard token "<*>" represents parameters of variable length, thus it can match more than one token. Consistent with existing work [Liu et al. 2019], we employ recursive processing to match the wildcard. Furthermore, to prevent overly broad matching, a limit is also imposed on the maximum number of tokens that a single "<*>" can match.

After the matching process, reaching a leaf node indicates an exact match of the template represented by the leaf node and the input log message, *i.e.*, the template of this log is stored in the parsing cache. Hence, there is no necessity to query the ICL-enhanced parser again, and the id of the leaf node is recorded as the template id of this log message. As shown in Case1 of Fig. 4, when the log message L_1 successfully matches the template T_1 stored in the parsing cache, LILAC directly marks T_1 as the parsed template of L_1. In some special cases, multiple matched templates may be returned. Consistent with previous research [He et al. 2017; Jiang et al. 2023], the template with the longest constant parts is selected, as it can match more non "<*>" characters, thereby indicating a higher likelihood of it being the template for this log message.

If no leaf node is reached after the recursive matching process, it will terminate at one or more internal nodes, referred to as *stop nodes*. In such cases, all templates within the subtrees of stop nodes form a list of relevant templates, denoted as $[T_1, T_2, \cdots, T_n]$. For example, in Case2 of Fig. 4, the cache matching process stops at the "`reading`" node, so the only template T_2 within the subtree is the relevant template. These relevant templates share a portion of prefixes identical to the current log message without an exact match. The failed matching may be caused by: (1) It is the first parsed log message of its respective template. (2) The LLM produces erroneous templates for previous

log messages with the same template. To discriminate the above two circumstances, the cache matching operation will return relevant templates for subsequent cache updating operation.

3.3.2 Cache Updating. When the cache matching of a specific log message fails, LILAC will query the ICL-enhanced parser to generate the template T_a. Although the LLM can correctly parse most log messages with the assistance of ICL, its unstable outputs and singular focus on the semantics of a single query may lead to the creation of erroneous and inconsistent templates. For example, when individually parsing two log messages, "`User admin successfully login in`" and "`User user1 successfully login in`", the LLM may erroneously interpret "`admin`" as a constant part, while considering "`user1`" as a parameter. However, when combining these two log messages for analysis, we can ascertain that both "`admin`" and "`user1`" are dynamic parameters, indicating the username.

To address this limitation and ensure the consistency of generated templates, we will compare the generated template with the relevant templates during the cache updating operation. If the newly generated template exhibits high similarity with an existing relevant template, these two log templates may be derived from the same ground-truth template. We then leverage the new template to refine the relevant template within the parsing cache. Otherwise, we will insert it as a new template into the parsing cache. In detail, after getting the newly generated log template T_a from the ICL-enhanced parser, we first discern whether T_a could potentially belong to the same ground-truth template as any relevant template in the parsing cache. We calculate the similarities between T_a and all relevant templates $\{T_1, T_2, \cdots, T_n\}$ returned by the cache matching operation. Given two templates T_1 and T_2, we split them into a list of tokens, denoted as L_1 and L_2. Then, the similarity between T_1 and T_2 is defined as: $Sim(T_1, T_2) = \frac{2 \times len(LCS(L_1, L_2))}{len(L_1)+len(L_2)}$, where the LCS is the longest common subsequence of two templates. We choose T_b from all relevant templates, which exhibits the highest similarity with T_a. (1) If the similarity is smaller than the pre-defined threshold (*e.g.*, 0.8 in our implementation), it implies that the new template T_a exhibits a low correlation with these relevant templates. As a result, LILAC directly *insert* T_a into the parsing cache. For example, in Case2 of Fig. 4, the similarity between T_a and T_2 is small, so T_a is inserted into the parsing cache as a new template. (2) If $Sim(T_a, T_b)$ exceeds the threshold, it indicates that T_a and T_b are highly similar and likely belong to the same ground-truth template. Such inconsistent templates may be caused by mistakes of LLMs. Therefore, we *refine* T_b by merging T_a to ensure the consistency. This is achieved by modifying the path of T_b within the tree of parsing cache, wherein the differing tokens are replaced with the "`<*>`". An example is shown in Case3 of Fig. 4, the similarity between T_a and T_3 is high, so we refine the "`admin`" node to "`<*>`". This creates a new refined template "`User <*> successfully log in`". In this manner, LILAC can adaptively update the parsing cache, utilizing both the answers of the LLM and historical templates within parsing cache, thus enhancing the accuracy of the parsed templates. Moreover, as the log templates within the parsing cache are considerably fewer than the log messages, and the cache matching operation selectively filters relevant templates, the overhead associated with cache updating is typically minimal.

4 EXPERIMENTAL SETUP

4.1 Research Questions

We evaluate LILAC on public large-scale log datasets by answering the following research questions:

- **RQ1:** How effective is LILAC in parsing log messages?
- **RQ2:** How does each design contribute to LILAC?
- **RQ3:** How capable is LILAC integrated with different LLMs?
- **RQ4:** How efficient is LILAC in processing large-scale log data?

4.2 Datasets and baselines

Our experiments are conducted using Loghub-2.0 [He et al. 2020; Jiang et al. 2023], a collection of large-scale datasets for log parsing from LogPAI [Zhu et al. 2019]. Loghub-2.0 contains ground-truth templates of 14 log datasets in Loghub [He et al. 2020] from a wide range of systems, including distributed systems, operating systems, and server-side applications. On average, each dataset in Loghub-2.0 contains 3.6 million log messages, all labeled with ground-truth log templates. Besides, the total number of log templates is about 3,500.

In accordance with recent benchmark studies [Jiang et al. 2023; Khan et al. 2022], we select four open-source and state-of-the-art log parsers for comparison with our method. The first two, AEL [Jiang et al. 2008] and Drain [He et al. 2017], are chosen due to their superior performance among all syntax-based log parsers. We also choose two latest semantic-based log parsers, UniParser [Liu et al. 2022] and LogPPT [Le and Zhang 2023b], considering the highest parsing accuracy they have achieved [Jiang et al. 2023]. To ensure a fair comparison, we use the implementations of all baselines from their replication repositories, choosing the default settings or hyper-parameters.

4.3 Metrics

Following recent studies [Jiang et al. 2023; Khan et al. 2022; Liu et al. 2022], we used the following four metrics in our experiments:

- *Grouping Accuracy (GA):* GA is computed as the ratio of correctly grouped log messages to the total count of log messages. A log message is considered to be correctly grouped if and only if its template aligns with the same set of log messages as that of the ground truth.
- *F1 score of Grouping Accuracy (FGA):* FGA is a template-level metric that focuses on the ratio of correctly grouped templates. Specifically, let N_g be the actual correct number of templates in the ground truth, and N_p be the number of templates that are generated by a log parser. If N_c is the number of templates that are correctly parsed by the log parser, then we can compute the Precision of Grouping Accuracy (PGA) as $\frac{N_c}{N_p}$ and the Recall of Grouping Accuracy (RGA) as $\frac{N_c}{N_g}$. The FGA is equal to their harmonic mean, *i.e.*, $\frac{2 \times GPA \times RGA}{PGA + RGA}$.
- *Parsing Accuracy (PA):* PA evaluates the capacity to extract the templates accurately, which is essential to downstream tasks such as anomaly detection [Liu et al. 2022]. PA is defined as the proportion of correctly parsed log messages to the total number of log messages. A log message is regarded to be correctly parsed if, and only if, all tokens of templates and variables are accurately identified.
- *F1 score of Template Accuracy (FTA):* Similar to FGA, FTA is a template-level metric that is calculated based on the proportion of correctly identified templates. It is computed as the harmonic mean of Precision and Recall of Template Accuracy. Differently, a template is regarded as correctly identified if and only if log messages of the parsed template share the same ground-truth template and all tokens of the template are the same as those of the ground-truth template.

4.4 Implementation and Environment

We conduct our experiments on an Ubuntu 20.04.5 LTS server with 256GB RAM and an NVIDIA GeForce GTX3090 since UniParser and LogPPT require GPU resources to perform log parsing. The default LLM in LILAC is set to ChatGPT (*gpt-3.5-turbo-0613*), primarily due to its popularity in recent research [Le and Zhang 2023a; Li et al. 2023a; Peng et al. 2023b; Xu et al. 2023b]. We call ChatGPT through the official API provided by OpenAI [ope 2023] and set its temperature to 0 so that ChatGPT would generate the same output for the same query to ensure reproducibility. Moreover, we also employ different LLMs to explore the generalizability of LILAC. To simulate the practical usage of LILAC, we use the sampling algorithm to select candidates from the first 20% of

the log messages in each dataset, and the default number of candidate samples and demonstration examples are set to 32 and 3, respectively. We also evaluate the performance of LILAC with different numbers of sampled candidates and demonstration examples.

We have implemented LILAC in Python and integrated it into previous benchmarks [Jiang et al. 2023; Khan et al. 2022; Zhu et al. 2019] so that we can fairly compare LILAC and all baselines in the same framework. For all experiments that exhibit randomness, we repeat them five times and report the median results following previous work [Jiang et al. 2023; Khan et al. 2022; Xu et al. 2023b] to avoid potential random bias.

5 EVALUATION RESULTS

5.1 RQ1: How effective is LILAC in parsing log messages?

Table 1. Accuracy comparison between baselines and LILAC on public datasets (%)

	AEL				Drain				UniParser				LogPPT				LILAC			
	GA	FGA	PA	FTA	GA	FGA	PA	FTA	GA	FGA	PA	FTA	GA	FGA	PA	FTA	GA	FGA	PA	FTA
Hadoop	82.3	11.7	53.5	5.8	**92.1**	78.5	54.1	38.4	69.1	62.8	**88.9**	47.6	48.3	52.6	66.6	43.4	87.2	**96.2**	83.2	**77.9**
HDFS	99.9	76.4	62.1	56.2	99.9	93.5	62.1	60.9	**100**	**96.8**	94.8	58.1	72.1	39.1	94.3	31.2	**100**	**96.8**	**99.9**	**94.6**
OpenStack	74.3	68.2	2.9	16.5	75.2	0.7	2.9	0.2	**100**	96.9	51.6	28.9	53.4	87.4	40.6	73.8	**100**	**100**	**100**	**97.9**
Spark	–	–	–	–	88.8	86.1	39.4	41.2	85.4	2.0	79.5	1.2	47.6	37.4	95.2	29.9	**100**	**90.1**	**97.3**	**75.9**
Zookeeper	99.6	78.8	84.2	46.5	99.4	90.4	84.3	61.4	98.8	66.1	**98.8**	51.0	96.7	91.8	84.5	80.9	**100**	**96.7**	68.7	**86.8**
BGL	91.5	58.7	40.6	16.5	**91.9**	62.4	40.7	19.3	91.8	62.4	94.9	21.9	24.5	25.3	93.8	26.1	89.4	**85.9**	**95.8**	**74.6**
HPC	74.8	20.1	74.1	13.6	79.3	30.9	72.1	15.2	77.7	66.0	94.1	35.1	78.2	78.0	**99.7**	76.8	**86.9**	**90.7**	70.5	**80.0**
Thunderbird	78.6	11.6	16.3	3.5	**83.1**	23.7	21.6	7.1	57.9	68.2	**65.4**	29.0	56.4	21.6	40.1	11.7	80.6	**79.3**	55.9	**57.2**
Linux	91.6	80.6	8.2	21.7	68.6	77.8	11.1	25.9	28.5	45.1	16.4	23.2	20.5	71.2	16.8	42.8	**97.1**	**93.1**	**76.5**	**74.0**
Mac	79.7	79.3	24.5	20.5	76.1	22.9	35.7	6.9	73.7	69.9	**68.8**	28.3	54.4	49.3	39.0	27.4	**87.6**	**82.5**	63.8	**55.3**
Apache	**100**	**100**	72.7	51.7	**100**	**100**	72.7	51.7	94.8	68.7	94.2	26.9	78.6	60.5	94.8	36.8	**100**	**100**	**99.6**	**86.2**
OpenSSH	70.5	68.9	36.4	33.3	**70.7**	**87.2**	58.6	48.7	27.5	0.9	28.9	0.5	27.7	8.1	65.4	10.5	69.0	83.8	**94.1**	**86.5**
HealthApp	72.5	0.8	31.1	0.3	86.2	1.0	31.2	0.4	46.1	74.5	81.7	46.2	99.8	94.7	**99.7**	82.2	**100**	**98.1**	72.9	**87.3**
Proxifier	97.4	66.7	67.7	41.7	69.2	20.6	68.8	17.6	50.9	28.6	63.4	45.7	98.9	87.0	**100**	95.7	**100**	**100**	**100**	**100**
Average	85.6	55.5	44.2	25.2	84.3	55.4	46.8	28.2	71.6	57.8	73.0	31.7	61.2	57.4	73.6	47.8	**92.7**	**92.4**	**84.2**	**81.0**

In this RQ, we conduct a comprehensive evaluation of the accuracy and robustness of LILAC in comparison to other state-of-the-art baselines on public datasets.

5.1.1 Accuracy. The accuracy is the most critical factor in the effectiveness of log parsers. We employ the default settings of all methods (*e.g.*, 32 sampled candidates for both LogPPT and LILAC) and apply them to all log datasets. The four selected metrics are shown in Table. 1, and the best results for each metric on each dataset are marked in **bold** font. The metrics for AEL on Spark are denoted as "–" since it cannot complete the parsing process of the Spark dataset within a reasonable time (*i.e.*, 12 hours), following previous works [Jiang et al. 2023; Khan et al. 2022].

According to the evaluation results, it is clear that LILAC outperforms all baselines on all average metrics. In specific, in terms of group-related metrics (*i.e.*, GA and FGA), LILAC achieves average scores of 92.7% and 92.4% on GA and FGA, outperforming Drain by 10.0% and 66.8%. However, the best baseline, Drain, achieves a GA of 84.3% but only an FGA of 55.4%. This is due to the imbalanced frequencies of templates in log datasets, and these log parsers may generate a large number of redundant and erroneous log templates, thereby leading to a markedly low PTA, which subsequently results in a low FTA. These redundant and erroneous templates can also seriously affect downstream tasks. Although the inherent instability of generative models causes the grouping-related metrics of UniParser and LogPPT to be inferior to other syntax-based log parsers, the designs of the parsing cache within LILAC are able to mitigate this issue, achieving superior grouping metrics.

When considering the metrics related to parsing ability (*i.e.*, PA and FTA), LogPPT has achieved the highest PA of 73.6% and FTA of 47.8% among all baselines. However, without the tuning process, LILAC has achieved superior parsing metrics, with a PA of 84.2% and an FTA of 81.0%, which

outperforms LogPPT by 14.4% and 69.5%, respectively. For the most stringent and comprehensive metric, the FTA, LILAC surpasses all baselines across all datasets. Given the strict definitions of correctly parsed and correctly identified, achieving such high metrics signifies that LILAC indeed possesses a strong capacity to distinguish between log templates and parameters.

5.1.2 Robustness. The robustness of log parsers is also an essential factor in evaluating their effectiveness. The strong robustness implies that log parsers can maintain a stable performance when dealing with log data of diverse characteristics, indicating a superior generalizability [Jiang et al. 2023; Le and Zhang 2023b; Xu et al. 2023b; Zhu et al. 2019]. To compare the robustness of LILAC with all baselines, we draw the box plot illustrating the distribution of each log parser's metrics across all datasets, as depicted in Fig. 5.

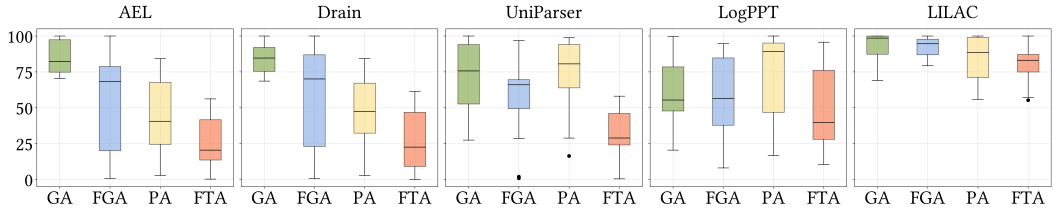

Fig. 5. Robustness comparison between baselines and LILAC on public datasets (%)

It is obvious that LILAC not only achieves the highest accuracy but also exhibits the least performance variance, as evidenced by its narrowest distribution range. This demonstrates that LILAC exhibits the strongest robustness when parsing various log data. Specifically, the standard deviations of LILAC for GA, FGA, PA and FTA are 9.3%, 6.9%, 15.2%, and 12.9%, respectively. In contrast, these values for LogPPT are 26.0%, 27.4%, 27.6%, and 27.3%. The strong robustness of LILAC is primarily derived from the vast pre-trained knowledge related to logs of LLMs. In addition, the ICL paradigm in the ICL-enhanced parser adapts the LLM to the system-specific characteristics of specific log datasets, thereby enhancing the robustness of parsing diverse log data.

Answer to RQ1: LILAC outperforms baseline methods on all metrics, with notable improvements of 66.8% and 69.5% for FGA and FTA, respectively, compared to Drain and LogPPT. Furthermore, LILAC exhibits the strongest robustness, reflected in the minimal performance variance when parsing diverse log data from different systems.

5.2 RQ2: How does each design contribute to LILAC?

In this RQ, we conduct a series of experiments to investigate the contributions of two designed modules within LILAC, *i.e.*, the ICL-enhanced parser and the parsing cache.

5.2.1 ICL-enhanced Parser. In the ICL-enhanced parser, we have designed an effective and efficient hierarchical candidate sampling algorithm, along with a kNN-based demonstration selection. In this section, we aim to investigate the individual contributions of these two designs and explore how different numbers of candidates or demonstrations will affect the performance of LILAC.

Contribution of ICL design choices. We first assess the individual contributions of the candidate sampling and demonstration selection algorithms. Specifically, we create the following four variants of LILAC and compare them with the original approach. 1) LILAC w/o ICL: remove the ICL design in ICL-enhanced parser, 2) LILAC w/ random selection: replace the kNN-based demonstration selection with random selection, 3) LILAC w/ random sampling: replace the candidate sampling algorithm with random sampling, 4) LILAC w/ LogPPT sampling: replace the candidate sampling algorithm with the adaptive sampling algorithm of LogPPT.

Table 2. Average accuracy comparison among LILAC with different strategies (%)

	GA	FGA	PA	FTA
LILAC	92.7	92.4	84.2	81.0
w/o ICL	83.5 (↓ 9.9%)	76.5 (↓ 17.2%)	62.6 (↓ 25.6%)	58.4 (↓ 27.9%)
w/ random selection	84.2 (↓ 9.2%)	80.6 (↓ 14.6%)	74.4 (↓ 11.6%)	66.7 (↓ 17.7%)
w/ random sampling	87.6 (↓ 5.5%)	80.9 (↓ 12.4%)	77.5 (↓ 8.0%)	68.3 (↓ 15.7%)
w/ LogPPT sampling	91.3 (↓ 1.5%)	84.8 (↓ 8.2%)	79.7 (↓ 5.3%)	74.9 (↓ 7.5%)

The evaluation results are depicted in Table. 2, in which the following observations can be made. (1) The absence of the ICL design substantially negatively impacts the performance of LILAC across all four metrics. For instance, the average FTA of LILAC experiences a considerable decrease of 27.9% when the ICL design is removed. The reason is that even though LLMs possess extensive pre-trained knowledge, their ability to effectively handle a wide range of log data remains limited in the absence of ICL capability. (2) Upon replacing the hierarchical candidate sampling and kNN-based demonstration selection algorithms with random strategies, there is a respective decrease of 17.7% and 15.7% in the average FTA, while the FGA values experience a reduction of 14.6% and 12.4%. This underscores the significance of the quality of candidate samples and demonstrations in influencing the performance of LLMs. (3) When we replace the original candidate sampling algorithm with that of LogPPT, there is a varying degree of decline across all four metrics, *e.g.*, the average FTA is reduced by 14.2%. The reason is that the sampling algorithm of LogPPT does not consider the issue of imbalanced template frequencies and the representativeness of the sampled candidates. In contrast, our proposed sampling algorithm is capable of sampling diverse and representative candidates, thereby effectively guiding LLMs to accurately parse the entire log dataset.

Impact of ICL parameter settings. In addition to the above-mentioned individual contributions of the designed algorithms, we also conduct experiments to evaluate the performance of LILAC using different numbers of sampled candidates and selected demonstrations.

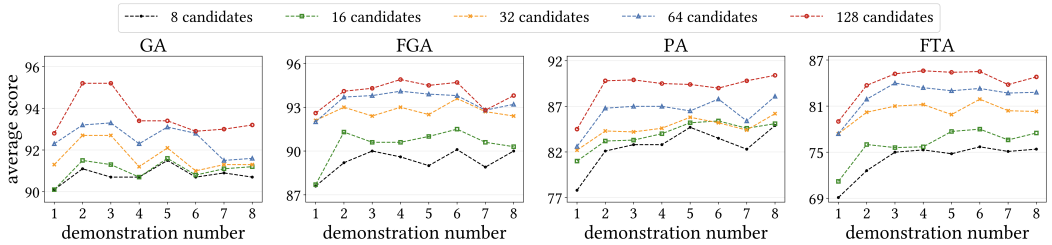

Fig. 6. Average accuracy among different numbers of sampled candidates and selected demonstrations. (%)

The results are shown in Fig. 6. It is clear that different numbers of both candidates and demonstrations can affect the performance of LILAC. Specifically, (1) Even though we only sampled 8 candidates and set the demonstration number greater than 3, the average FTA of LILAC is around 75%, which is much higher than LILAC without ICL design in Table. 2 (*i.e.*, 58.4%). This suggests that the ICL design can effectively adapt the LLM to parse various log data even when the quantity of labeled data is limited. (2) As the number of candidates increases, all four metrics of LILAC exhibit an improvement. For instance, when the number of demonstrations is set to 3, the average FTAs for candidate numbers 8, 32, and 128 are approximately 75%, 81%, and 85%, respectively. This is because the more sampled candidates can provide a broader range of semantic and pattern characteristics in log data, which can be selected to demonstrate LLMs, thereby enabling more precise template generation. (3) The performance of LILAC is influenced by the varying number of demonstrations

for each query under each setting of the candidate number. In particular, the performance of LILAC is lowest when only a single demonstration is used since a single demonstration can introduce inductive bias into the parsing process of LLMs. However, when the number of demonstrations exceeds three and continues to increase, the performance of LILAC exhibits fluctuations but tends towards stability. This is because our kNN-based algorithm is capable of selecting demonstrations that are not only similar to the queried log but also exhibit a high degree of consistency. In summary, although the performance of LILAC is influenced by the varying number of sampled candidates and selected demonstrations, an enhancement in the performance of LILAC is observed across all settings when compared to LILAC w/o ICL. Moreover, the most appropriate configuration of 32 candidates and 3 demonstrations is selected as the default setting in our other experiments.

Table 3. Average accuracy comparison between LogPPT and LogPPT with parsing cache (%)

	GA	FGA	PA	FTA
LogPPT (original)	61.2	57.4	73.6	47.8
w/ parsing cache	90.8 (↑ 48.4%)	89.0 (↑ 55.1%)	78.0 (↑ 6.0%)	67.4 (↑ 41.0%)

5.2.2 Parsing Cache. One of the design objectives of parsing cache is to mitigate the inconsistency in the answers of LLMs. To validate it, in this section, we evaluate the contribution of the parsing cache to the performance enhancement of LILAC. A direct approach is comparing the performance of the original LILAC with that of LILAC without the parsing cache. However, considering the substantial size of these log datasets (averaging 3.6 million log messages per dataset) and the overhead of querying the LLM, it is infeasible to utilize current LLMs for parsing these datasets without the aid of parsing cache, *i.e.*, processing line by line. Instead, we replace the ICL-enhanced parser with a smaller language model, RoBERTa, which is used by the latest semantic-based log parsers, LogPPT. Both RoBERTa and LLMs exhibit the common issue of unstable outputs, given that they are both generative language models. Consequently, comparing the original LogPPT and LogPPT with parsing cache can reflect the effectiveness of parsing cache in mitigating the instability associated with generative language models.

The results are presented in Table. 3. It is evident that the integration of parsing cache has substantially improved the performance of LogPPT. First, regarding the grouping-related metrics, the mean GA and FGA of LogPPT with parsing cache have risen by 48.4% and 55.1%, respectively, in contrast to the original LogPPT. This implies that by matching and adaptively updating parsing cache, LILAC can ensure the consistency of templates generated by language models, thereby improving the accuracy of grouping. Second, both the mean PA and FTA have demonstrated respective increases of 6.0% and 41.0%. This suggests that the specifically designed refinements of templates within parsing cache can accurately rectify the incorrect templates produced by language models based on historical templates.

Answer to RQ2: Both designs of ICL-enhanced parser and parsing cache significantly contribute to enhancing LILAC's overall performance. On the one hand, the proposed ICL strategies provide LLMs the capability of accurately parsing log messages. On the other hand, parsing cache is effective in mitigating the inconsistency inherent in language models.

5.3 RQ3: How capable is LILAC integrated with different LLMs?

In this RQ, we compare the performance of LILAC by employing different LLMs in ICL-enhanced parser. Specifically, we select three representative LLMs commonly used in research [Gao et al. 2023; Xu et al. 2023b], namely, ChatGPT, Davinci, and Curie. Both ChatGPT and Davinci possess a substantial model parameter count of 175B. ChatGPT, having been fine-tuned for conversational

Table 4. Average accuracy comparison among LILAC with different LLMs (%)

	GA	FGA	PA	FTA
ChatGPT	92.7	92.4	84.2	81.0
Davinci	91.9 (↓ 0.9%)	92.9 (↑ 0.5%)	87.1 (↑ 3.4%)	81.5 (↑ 0.6%)
Curie	90.1 (↓ 2.8%)	87.6 (↓ 5.2%)	77.8 (↓ 7.6%)	71.2 (↓ 12.1%)

tasks, provides a superior generation speed. Conversely, Davinci has enhanced capabilities in executing text-generation tasks. Furthermore, Curie is distinguished by the smallest parameter size, amounting to 13B.

Table. 4 demonstrates the average metrics of LILAC with different LLMs, from which we can find consistently high performance across all LLMs. In detail, both LILAC with ChatGPT and Davinci have achieved exceedingly high average metrics, due to their vast parameter volume and extensive pre-training knowledge. We have also observed that the majority of these four metrics for Davinci marginally surpass those of ChatGPT, *e.g.*, FTA augmented by 0.6%. The reason could be that Davinci is more focused on text-generation tasks, which aligns with the log parsing task. Furthermore, we can observe that the performance of LILAC with Curie is the most inferior, *e.g.*, the average FTA of Curie is 12.1% lower than that of ChatGPT. This is due to the limited model parameters and pre-training knowledge of Curie, signifying a poorer text processing capability, as well as a weaker ICL capacity [Wang et al. 2023b]. However, LILAC with a comparatively smaller LLM can still achieve an accuracy surpassing all existing log parsers. These results demonstrate that LILAC can be generally applied to different LLMs, maintaining high accuracy.

Answer to RQ3: The performance of LILAC can be influenced by the capabilities of LLMs. Nevertheless, LILAC is able to consistently achieve high performance, even when utilizing relatively smaller LLMs.

5.4 RQ4: How efficient is LILAC in processing large-scale log data?

Efficiency is of paramount importance in the practical application of log parsers, given the substantial volume of logs [Le and Zhang 2023b; Wang et al. 2022; Zhu et al. 2019]. LILAC encompasses two primary time costs, *i.e.*, the time of the candidate sampling process and the parsing process. In this RQ, we assess the efficiency of these two procedures within LILAC, utilizing the public Loghub-2.0 datasets as described in Sec. 4.2.

Table 5. Average sampling time of LILAC and LogPPT algorithms on large-scale datasets (seconds)

	8 candidates	16 candidates	32 candidates	64 candidates	128 candidates
LogPPT	284.1	629.3	1303.9	2779.6	5396.9
LILAC	19.2	19.3	19.3	19.3	19.4
Speed up (↑)	14.8 ×	32.6 ×	67.6 ×	144.0 ×	278.2 ×

5.4.1 Candidate Sampling Efficiency. Although Xu et al. [2023b] have proposed a DPP-based sampling algorithm for log parsing, calculating pair-wise distances between all log messages makes it infeasible for execution on large-scale datasets. In this section, we perform the sampling algorithm of LILAC and LogPPT on all datasets and calculate the average sampling time. The results are shown in Table. 5, from which we can conclude that the efficiency of the sampling algorithm within LILAC significantly surpasses that of the LogPPT. For instance, when sampling 32 candidates, the algorithm of LogPPT requires more than 1300 seconds, whereas LILAC only necessitates 19.3 seconds, achieving a speedup of 67.6. Moreover, the time cost of the LogPPT sampling algorithm

increases linearly with the number of sampled candidates as it employs an iterative approach. In contrast, the time cost of the LILAC sampling algorithm remains stable regardless of the number of candidates sampled. The reason is that the time overhead of LILAC's sampling algorithm is almost on hierarchical clustering, which is efficient for handling extensive log data.

5.4.2 Parsing Efficiency. In this section, our primary focus is on assessing the efficiency of the parsing process within LILAC. More specifically, we have recorded the execution times for all baselines and LILAC with ChatGPT on all log datasets. Furthermore, we have separately recorded the cache operation time of the parsing cache within LILAC and the time expended on querying the ICL-enhanced parser. Specifically, the cache operation time encompasses the time cost for cache matching and cache updating operations, whereas the query time represents the total duration from initiating a query to the ICL-enhanced parser to receiving the generated templates. We calculate the average parsing time across all log datasets and plot a bar chart. The detailed parsing times are available in our replication package [rep 2023]. According to the results in Fig. 7, we can see that LILAC exhibits efficiency comparable to that of Drain, the most efficient syntax-based log parser currently available. In detail, LILAC requires 569.6 seconds to process an average of 3.6 million log messages, while this time of Drain is 425.4. Conversely, other semantic-based log parsers, including UniParser and LogPPT, despite the utilization of GPU acceleration, only achieve low efficiencies, trailing LILAC by 4.03 and 7.19 times, respectively.

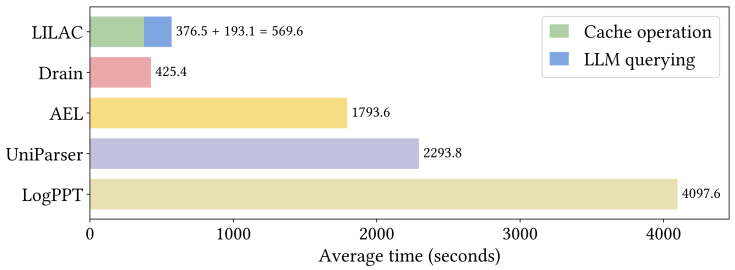

Fig. 7. Efficiency of baselines and LILAC on large-scale datasets

Besides, across all datasets, the average processing time for parsing cache in LILAC is 376.5s, accounting for approximately 66.1% of the total time. This is less than the average processing time of Drain, suggesting that the cache matching and updating operations of the parsing cache are highly efficient. Correspondingly, the time consumed by querying the ICL-enhanced parser averages at 193.1s, representing 33.9% of the total time. We further conduct a statistical analysis on the number of queries to the LLM. The mean value of query numbers is 279.7, while the average number of ground-truth templates is 249. The observed difference is caused by the incorrect templates generated by the LLM, which results in failed matching of the parsing cache and consequently leads to unnecessary queries. However, LILAC can effectively keep this number minimal, thereby ensuring efficiency. When compared with the existing LLM-based log parsing approaches [Le and Zhang 2023a; Liu et al. 2023b; Xu et al. 2023b], which necessitate over 3.6 million queries, LILAC markedly diminishes the number of queries to LLMs. This makes the application of LLMs for log parsing practically feasible.

Answer to RQ4: LILAC substantially reduces the number of queries to the LLM by matching the parsing cache to prevent duplicate queries. Hence, the efficiency of LILAC surpasses semantic-based methods by 4.03 to 7.19 times and is comparable to the fastest syntax-based methods.

6 DISCUSSION

6.1 Practicality of LILAC

LILAC is designed to leverage the power of LLMs for log parsing in production systems. To reduce the cost of querying LLMs and alleviate the inherent instability of query results, LILAC adopts the adaptive parsing cache. In addition, for each query, since the number of tokens in a single log message or template is generally small (*e.g.*, tens to hundreds of tokens), LILAC would not incur a substantial cost of querying LLMs. Additionally, LILAC introduces effective candidate sampling and demonstration selection algorithms to facilitate the ICL capability of LLMs in log parsing.

It is worth noting that LILAC is compatible with traditional language models, such as RoBERTa. According to the experimental results in Sec. 5.2.2, LILAC integrated with traditional language models can also achieve higher performance than state-of-the-art log parsing methods. When using traditional language models, users can utilize the proposed candidate sampling algorithm to obtain high-quality data for model training or tuning. We believe the above features make LILAC a practical framework that can be deployed in real-world systems.

6.2 Threats to Validity

Data Leakage. Since LLMs are trained on huge volumes of data, one potential threat is the data leakage problem. Particularly, the adopted LLM in LILAC may have been trained on open-source log datasets, leading to the memorization of ground-truth templates as opposed to performing inference. However, according to our experiments, the performance of LILAC without ICL is significantly inferior to LILAC with ICL, implying a low probability of direct memorization. Furthermore, LILAC employs the *gpt-turbo-3.5-0613* model for most of the experiments. It is noteworthy that updates for this model were discontinued before the ground-truth templates in Loghub-2.0 were publicly available. Therefore, the probability of data leakage within our experiment is negligible.

Privacy Issue. From the perspective of enterprises, log messages are sensitive data, as they often encompass a substantial amount of customer and service information. Employing external LLMs to process internal log data may pose risks to privacy and security problems. Actually, LILAC is a general framework that can support a variety of language models. Users can integrate their own language models into LILAC, thereby avoiding privacy issues.

Manual Labeling Effort. To utilize the ICL capability of LLMs, manual annotation is required to provide the ground-truth templates of the sampled log messages. To alleviate the labeling effort associated with ICL, we propose an efficient candidate sampling algorithm designed to sample a compact set of diverse and representative log messages. Our experimental results proved that even with a small number of labeled log messages (*e.g.*, 32), LILAC can yield a significantly improved performance.

7 RELATED WORK

Log parsing has emerged as an active research topic in recent years [He et al. 2016; Jiang et al. 2023; Khan et al. 2022; Zhu et al. 2019]. Existing log parsers can be divided into two groups: syntax-based and semantic-based log parsers. In specific, syntax-based log parsers can be further subdivided into three categories. *(1) Frequency-based parsers:* These log parsers [Dai et al. 2020; Nagappan and Vouk 2010; Vaarandi 2003; Vaarandi and Pihelgas 2015] utilize frequent patterns of token position or n-gram information to distinguish the templates and parameters in log messages. *(2) Similarity-based parsers:* These log parsers [Hamooni et al. 2016; Shima 2016; Tang et al. 2011] compute similarities between log messages to cluster them into different groups and then extract the constant parts of log messages. *(3) Heuristics-based parsers:* These log parsers [Du and Li 2016; He et al. 2017; Jiang et al. 2008; Makanju et al. 2009; Messaoudi et al. 2018; Mizutani 2013; Wang et al. 2022] employ various heuristic algorithms or data structures to identify the log templates

based on designed characteristics. Semantic-based log parsers can achieve higher parsing accuracy by mining semantics from log messages, which is crucial in some downstream tasks [Huo et al. 2023; Li et al. 2023b]. These methods typically necessitate labeled log data for model training or tuning. To be precise, a subset of these log parsers [Huo et al. 2023; Li et al. 2023b; Liu et al. 2022] formulate log parsing as a token classification problem, employing bidirectional long short-term memory for training. In addition, LogPPT [Le and Zhang 2023b] tunes a pre-trained language model (*e.g.*, RoBERTa) to perform log parsing.

However, recent benchmark studies [Jiang et al. 2023; Khan et al. 2022] have identified that the performance of these log parsers is found to be inadequate when dealing with large-scale, complex log data. This observation motivates our work, which aims to utilize the capabilities of LLMs for more accurate log parsing. Recently, several studies have been conducted to explore the utilization of LLMs for log analysis, specifically log parsing. The study by Le and Zhang [2023a] is the pioneer in investigating the performance of LLMs in log parsing, which demonstrates the potential of LLMs in accomplishing log parsing. Xu et al. [2023b] propose LogDiv, a method that leverages the ICL capability of LLMs to achieve more accurate log parsing. To adpot the ICL paradigm, LogDiv transforms all log messages into embeddings and computes pair-wise distances to sample log messages for demonstration, which is infeasible when dealing with an enormous volume of log messages. Besides, these existing methods solely employ LLMs to sequentially parse each log in a single query. Hence, they do not address the challenges of efficiency and consistency inherent in utilizing LLMs to log parsing. This makes them impractical for utilization in real-world scenarios. In contrast, our proposed method, LILAC, addresses these issues by combining the adaptive parsing cache with the ICL-enhanced parser, enabling accurate and efficient LLM-based log parsing.

8 CONCLUSION

In this paper, we present LILAC, a practical log parsing framework using LLMs with adaptive parsing cache. To utilize the ICL capability to adapt LLMs to parse various log data, LILAC adopts effective and efficient candidate sampling and demonstration selection algorithms to select high-quality demonstrations. Besides, LILAC employs the adaptive parsing cache to store log templates, and specifically tailored cache matching and adaptive updating operations help mitigate the inherent inconsistency and inefficiency of LLMs. Extensive experiments on large-scale log datasets demonstrate that LILAC outperforms all state-of-the-art baselines with high efficiency. We believe that LILAC would benefit both practitioners and researchers in the field of log analysis.

ACKNOWLEDGMENT

The work described in this paper was supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14206921 of the General Research Fund). We extend our sincere gratitude to the anonymous reviewers for their constructive feedback.

REFERENCES

2023. Jaccard index - Wikipedia. https://en.wikipedia.org/wiki/Jaccard_index [Online; accessed 1 Aug 2023].

2023. OpenAI API. https://openai.com/blog/openai-api [Online; accessed 1 Aug 2023].

2023. The repository of LILAC. https://github.com/logpai/LILAC [Online; accessed 29 Jan 2024].

2023. Scipy. https://scipy.org/ [Online; accessed 1 Aug 2023].

Shan Ali, Chaima Boufaied, Domenico Bianculli, Paula Branco, Lionel Briand, and Nathan Aschbacher. 2023. An Empirical Study on Log-based Anomaly Detection Using Machine Learning. *arXiv preprint arXiv:2307.16714* (2023).

Anunay Amar and Peter C Rigby. 2019. Mining historical test logs to predict bugs and localize faults in the test logs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 140–151.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information*

processing systems 33 (2020), 1877–1901.

An Ran Chen, Tse-Hsun Chen, and Shaowei Wang. 2021. Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs. *IEEE Transactions on Software Engineering (TSE)* 48, 8 (2021), 2905–2919.

Hetong Dai, Heng Li, Che-Shao Chen, Weiyi Shang, and Tse-Hsun Chen. 2020. Logram: Efficient Log Parsing Using *n* n-Gram Dictionaries. *IEEE Transactions on Software Engineering (TSE)* 48, 3 (2020), 879–892.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339* (2022).

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey for in-context learning. *arXiv preprint arXiv:2301.00234* (2022).

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.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. 2023. Improving Factuality and Reasoning in Language Models through Multiagent Debate. *arXiv preprint arXiv:2305.14325* (2023).

Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, and Michael R Lyu. 2023. Constructing Effective In-Context Demonstration for Code Intelligence Tasks: An Empirical Study. *arXiv preprint arXiv:2304.07575* (2023).

Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. 2016. Logmine: Fast pattern recognition for log analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management (CIKM)*. 1573–1582.

Hangfeng He, Hongming Zhang, and Dan Roth. 2022. Rethinking with retrieval: Faithful large language model inference. *arXiv preprint arXiv:2301.00303* (2022).

Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. 2016. An evaluation study on log parsing and its use in log mining. In *2016 46th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 654–661.

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.

Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. 2020. Loghub: A large collection of system log datasets towards automated log analytics. *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)* (2020).

Yintong Huo, Yuxin Su, Cheryl Lee, and Michael R Lyu. 2023. Semparser: A semantic parser for log analytics. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 881–893.

Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jiazhen Gu, Zhuangbin Chen, Jieming Zhu, and Michael R Lyu. 2023. A Large-scale Benchmark for Log Parsing. *arXiv preprint arXiv:2308.10828* (2023).

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.

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 (ICSE)*. 1095–1106.

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 (ICSE)*. 1356–1367.

Van-Hoang Le and Hongyu Zhang. 2023a. Log Parsing: How Far Can ChatGPT Go? *arXiv preprint arXiv:2306.01590* (2023).

Van-Hoang Le and Hongyu Zhang. 2023b. Log Parsing with Prompt-based Few-shot Learning. *arXiv preprint arXiv:2302.07435* (2023).

Xiaoyun Li, Hongyu Zhang, Van-Hoang Le, and Pengfei Chen. 2023c. LogShrink: Effective Log Compression by Leveraging Commonality and Variability of Log Data. *arXiv preprint arXiv:2309.09479* (2023).

Yichen Li, Yintong Huo, Zhihan Jiang, Renyi Zhong, Pinjia He, Yuxin Su, and Michael R Lyu. 2023a. Exploring the Effectiveness of LLMs in Automated Logging Generation: An Empirical Study. *arXiv preprint arXiv:2307.05950* (2023).

Zhenhao Li, Chuan Luo, Tse-Hsun Chen, Weiyi Shang, Shilin He, Qingwei Lin, and Dongmei Zhang. 2023b. Did We Miss Something Important? Studying and Exploring Variable-Aware Log Abstraction. *arXiv preprint arXiv:2304.11391* (2023).

Jinyang Liu, Junjie Huang, Yintong Huo, Zhihan Jiang, Jiazhen Gu, Zhuangbin Chen, Cong Feng, Minzhi Yan, and Michael R Lyu. 2023a. Scalable and Adaptive Log-based Anomaly Detection with Expert in the Loop. *arXiv preprint arXiv:2306.05032* (2023).

Jinyang Liu, Jieming Zhu, Shilin He, Pinjia He, Zibin Zheng, and Michael R Lyu. 2019. Logzip: Extracting hidden structures via iterative clustering for log compression. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 863–873.

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

Yilun Liu, Shimin Tao, Weibin Meng, Jingyu Wang, Wenbing Ma, Yanqing Zhao, Yuhang Chen, Hao Yang, Yanfei Jiang, and Xun Chen. 2023b. LogPrompt: Prompt Engineering Towards Zero-Shot and Interpretable Log Analysis. *arXiv preprint*

arXiv:2308.07610 (2023).

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 (WWW)*. 1893–1901.

James MacGlashan, Mark K Ho, Robert Loftin, Bei Peng, Guan Wang, David L Roberts, Matthew E Taylor, and Michael L Littman. 2017. Interactive learning from policy-dependent human feedback. In *International conference on machine learning*. PMLR, 2285–2294.

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 (KDD)*. 1255–1264.

Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. 2022. Using deep learning to generate complete log statements. In *Proceedings of the 44th International Conference on Software Engineering*. 2279–2290.

Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. 2018. A search-based approach for accurate identification of log message formats. In *Proceedings of the 26th Conference on Program Comprehension*. 167–177.

Masayoshi Mizutani. 2013. Incremental mining of system log format. In *2013 IEEE International Conference on Services Computing*. IEEE, 595–602.

Priyanka Mudgal and Rita Wouhaybi. 2023. An Assessment of ChatGPT on Log Data. *arXiv preprint arXiv:2309.07938* (2023).

N Mündler, J He, S Jenko, and M Vechev. 2023. Self-contradictory hallucinations of large language models: Evaluation, detection and mitigation.

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)*. IEEE, 114–117.

Paolo Notaro, Soroush Haeri, Jorge Cardoso, and Michael Gerndt. 2023. LogRule: Efficient Structured Log Mining for Root Cause Analysis. *IEEE Transactions on Network and Service Management* (2023).

Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. 2015. Industry practices and event logging: Assessment of a critical software development process. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, Vol. 2. IEEE, 169–178.

Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng, Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou Yu, Weizhu Chen, et al. 2023a. Check your facts and try again: Improving large language models with external knowledge and automated feedback. *arXiv preprint arXiv:2302.12813* (2023).

Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R Lyu. 2023b. Generative Type Inference for Python. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 988–999.

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.

Kirk Rodrigues, Yu Luo, and Ding Yuan. 2021. {CLP}: Efficient and Scalable Search on Compressed Text Logs. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 183–198.

Daan Schipper, Maurício Aniche, and Arie van Deursen. 2019. Tracing back log data to its log statement: from research to practice. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 545–549.

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

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 (CIKM)*. 785–794.

Risto Vaarandi. 2003. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM)(IEEE Cat. No. 03EX764)*. Ieee, 119–126.

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.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

Lingzhi Wang, Nengwen Zhao, Junjie Chen, Pinnong Li, Wenchi Zhang, and Kaixin Sui. 2020. Root-cause metric location for microservice systems via log anomaly detection. In *2020 IEEE international conference on web services (ICWS)*. IEEE, 142–150.

Xindi Wang, Yufei Wang, Can Xu, Xiubo Geng, Bowen Zhang, Chongyang Tao, Frank Rudzicz, Robert E Mercer, and Daxin Jiang. 2023b. Investigating the Learning Behaviour of In-context Learning: A Comparison with Supervised Learning. *arXiv preprint arXiv:2307.15411* (2023).

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 (FSE)*. 1198–1208.

Yiding Wang, Kai Chen, Haisheng Tan, and Kun Guo. 2023a. Tabi: An Efficient Multi-Level Inference System for Large Language Models. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 233–248.

Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652* (2021).

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.

Junjielong Xu, Qiuai Fu, Zhouruixing Zhu, Yutong Cheng, Zhijing Li, Yuchi Ma, and Pinjia He. 2023a. Hue: A user-adaptive parser for hybrid logs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 413–424.

Junjielong Xu, Ruichun Yang, Yintong Huo, Chengyu Zhang, and Pinjia He. 2023b. DivLog: Log Parsing with Prompt Enhanced In-Context Learning. *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)* (2023).

Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. 2009. Largescale system problem detection by mining console logs. *Proceedings of SOSP'09* (2009).

Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, DongGyun Han, and David Lo. 2023. What Do Code Models Memorize? An Empirical Study on Large Language Models of Code. *arXiv preprint arXiv:2308.09932* (2023).

Kundi Yao, Mohammed Sayagh, Weiyi Shang, and Ahmed E Hassan. 2021. Improving state-of-the-art compression techniques for log management tools. *IEEE Transactions on Software Engineering (TSE)* 48, 8 (2021), 2748–2760.

Siyu Yu, Ningjiang Chen, Yifan Wu, and Wensheng Dou. 2023a. Self-supervised log parsing using semantic contribution difference. *Journal of Systems and Software* 200 (2023), 111646.

Siyu Yu, Pinjia He, Ningjiang Chen, and Yifan Wu. 2023b. Brain: Log Parsing with Bidirectional Parallel Tree. *IEEE Transactions on Services Computing (TSC)* (2023).

Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. 2022. DeepTraLog: Trace-log combined microservice anomaly detection through graph-based deep learning. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. 623–634.

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 (FSE)*. 807–817.

Nengwen Zhao, Honglin Wang, Zeyan Li, Xiao Peng, Gang Wang, Zhu Pan, Yong Wu, Zhen Feng, Xidao Wen, Wenchi Zhang, et al. 2021b. An empirical investigation of practical log anomaly detection for online service systems. In *Proceedings of the 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering (FSE)*. 1404–1415.

Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021a. Calibrate before use: Improving few-shot performance of language models. In *International Conference on Machine Learning*. PMLR, 12697–12706.

Shen Zheng, Jie Huang, and Kevin Chen-Chuan Chang. 2023. Why does chatgpt fall short in providing truthful answers. *ArXiv preprint, abs/2304.10513* (2023).

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.

Received 2023-09-29; accepted 2024-01-23