ULog: Unsupervised Log Parsing with Large Language Models through Log Contrastive Units

Junjie Huang
The Chinese University of Hong Kong
Hong Kong, China

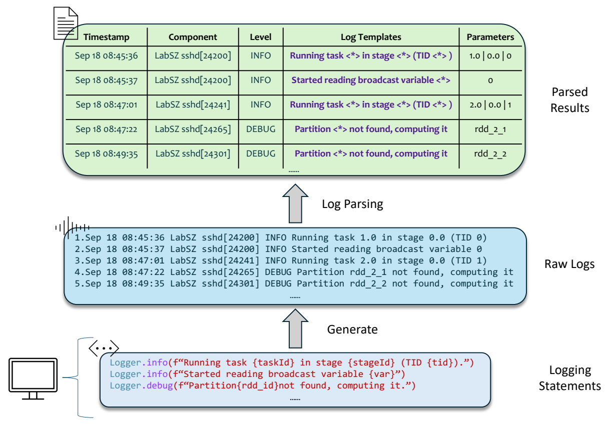
Zhihan Jiang
The Chinese University of Hong Kong
Hong Kong, China

Zhuangbin Chen
Sun Yat-sen University
China

Michael R. Lyu
The Chinese University of Hong Kong
Hong Kong, China

ABSTRACT

Log parsing serves as an essential prerequisite for various log analysis tasks. Recent advancements in this field have improved parsing accuracy by leveraging the semantics in logs through fine-tuning large language models (LLMs) or learning from in-context demonstrations. However, these methods heavily depend on labeled examples to achieve optimal performance. In practice, collecting sufficient labeled data is challenging due to the large scale and continuous evolution of logs, leading to performance degradation of existing log parsers after deployment. To address this issue, we propose ULog, an unsupervised LLM-based method for efficient and off-the-shelf log parsing. Our key insight is that while LLMs may struggle with direct log parsing, their performance can be significantly enhanced through comparative analysis across multiple logs that differ only in their parameter parts. We refer to such groups of logs as *Log Contrastive Units (LCUs)*. Given the vast volume of logs, obtaining LCUs is difficult. Therefore, ULog introduces a hybrid ranking scheme to effectively search for LCUs by jointly considering the *commonality* and *variability* among logs. Additionally, ULog crafts a novel parsing prompt for LLMs to identify contrastive patterns and extract meaningful log structures from LCUs. Experiments on large-scale public datasets demonstrate that ULog significantly outperforms state-of-the-art log parsers in terms of accuracy and efficiency, providing an effective and scalable solution for real-world deployment.



Timestamp	Component	Level	Log Templates	Parameters
Sep 18 08:45:36	LabSZ sshd[24200]	INFO	Running task <*> in stage <*> (TID <*>)	1.0 | 0.0 | 0
Sep 18 08:45:37	LabSZ sshd[24200]	INFO	Started reading broadcast variable <*>	0
Sep 18 08:47:01	LabSZ sshd[24241]	INFO	Running task <*> in stage <*> (TID <*>)	2.0 | 0.0 | 1
Sep 18 08:47:22	LabSZ sshd[24265]	DEBUG	Partition <*> not found, computing it	rdd_2_1
Sep 18 08:49:35	LabSZ sshd[24301]	DEBUG	Partition <*> not found, computing it	rdd_2_2

......

```
1.Sep 18 08:45:36 LabSZ sshd[24200] INFO Running task 1.0 in stage 0.0 (TID 0)
2.Sep 18 08:45:37 LabSZ sshd[24200] INFO Started reading broadcast variable 0
3.Sep 18 08:47:01 LabSZ sshd[24241] INFO Running task 2.0 in stage 0.0 (TID 1)
4.Sep 18 08:47:22 LabSZ sshd[24265] DEBUG Partition rdd_2_1 not found, computing it
5.Sep 18 08:49:35 LabSZ sshd[24301] DEBUG Partition rdd_2_2 not found, computing it
                                ......
```

```
Logger.info(f"Running task {taskId} in stage {stageId} (TID {tid}).")
Logger.info(f"Started reading broadcast variable {var}")
Logger.debug(f"Partition{rdd_id}not found, computing it.")
                                ......
```

Figure 1: An example of log parsing procedure.

1 INTRODUCTION

Log messages record events, transactions, or activities generated by software applications and operating systems at runtime [15, 26, 46]. They provide valuable insights for system performance monitoring and reliability assurance. Various tools have been developed to conduct automated log analysis tasks, including anomaly detection [2, 6, 52–54], root cause analysis [3, 38, 45], and failure diagnosis [5, 50]. *Log parsing*, which transforms semi-structured log messages into structured formats [21], serves as a critical preliminary step in log analysis. Typically, a raw log message contains two parts: 1) *log templates*: constant parts that describe the main content of the logged event; 2) *log parameters*: dynamic parts that contain the parameters (determined at runtime) associated with the event. Figure 1 demonstrates some log messages generated by their logging statements and parsed into structured data.

A straightforward approach to log parsing involves matching raw log messages with their corresponding logging statements in the source code [4, 39, 40]. However, in practice, source code is not always accessible, particularly for commercial software and third-party libraries. Consequently, various data-driven log parsers have been proposed to directly extract templates and parameters from raw logs. They can be generally categorized into two types: *syntax-based parsers* and *semantic-based parsers*. Syntax-based parsers [14, 44, 48, 51] resort to statistical or heuristic rules (*e.g.*, log length, word frequency) to identify common parts among logs as the templates. However, these methods often struggle to accurately recognize templates when log messages deviate from the manually-crafted rules. To address these limitations, deep learning models have been introduced in this field to leverage more advanced features of log data, *i.e.*, their semantics. For example, LogPPT [23] fine-tunes a pre-trained language model (*e.g.*, RoBERTa [28]) based on manually labelled log templates, aiming for better performance. With the increasing popularity of large language models (LLMs) for log analysis, some of the latest work [18, 49] also employ LLMs for log parsing. These LLM-based log parsers leverage the extensive pre-trained knowledge of LLMs and utilize the in-context learning (ICL) paradigm [8, 11]. By providing labelled examples as demonstrations, they specialize LLMs for the task of log parsing.

Despite effective, existing semantic-based log parsers largely depend on labelled examples to achieve optimal performance [32]. However, collecting sufficient labelled data is challenging in real world, which hinders their applicability and scalability in practice. On one hand, production log data are often large in volume, *e.g.*, hundreds of millions of logs per hour according to recent studies [46].

Manually annotating log messages of such scale from diverse systems is labor-intensive and error-prone, which demands substantial domain expertise to ensure accuracy and consistency [19]. On the other hand, the log data of real-world systems are continuously evolving over time [46, 49]. New log messages and log templates can emerge frequently, reflecting changes in system behavior, updates, and new feature deployments. This dynamic nature necessitates constant re-annotation and adaptation of the parsers, making it difficult to maintain a high accuracy. Consequently, the performance of existing semantic-based log parsers may degrade significantly without continuous and significant manual intervention. As demonstrated in our empirical study (Section 2.1), when the proportion of labelled examples decreases by 70%, the F1 score of template accuracy (FTA) of the state-of-the-art semantic-based log parsers, LogPPT [23] and LILAC [18], drops by 33% and 15%, respectively.

To address this label-demanding problem, we propose ULog, an unsupervised LLM-based log parser, which can generalize to any new log dataset without manual annotations or prior knowledge of the specific log formats. The core idea behind LUNAR lies in leveraging LLMs' ability to perform comparative analysis on multiple log messages that vary in their parameter parts. While LLMs may struggle with direct log parsing, the comparison can derive valuable insights by identifying and interpreting patterns across these variations. For instance, by contrasting logs such as "`session opened for user news`" and "`session opened for user test`," LLMs can easily infer that the tokens "news" and "test" are likely to represent a username parameter. We refer to such group of logs as a Log Contrastive Unit (LCU), which are similar enough to facilitate meaningful comparisons, yet diverse enough to highlight the variable parameters. To find LCUs, we introduce a hybrid ranking scheme by jointly considering the *commonality* and *variability* among logs. However, given the vast volume of logs, evaluating every possible combination of logs is impractical. Thus, we introduce a hierarchical sharder to first divide logs into different buckets based on log length and top-k frequent tokens. The LCU selection and subsequent log parsing can then be efficiently performed in each bucket in parallel. With the selected LCU, ULog crafts a novel specialized parsing prompt without the need of labelled templates for ICL. The prompt specifies task intention and output constraints in detail, and provides representative parameter examples to inform LLMs of the parameter characteristics.

We evaluate the performance of ULog against a range of label-free (unsupervised) parsers and label-dependent parsers. The experiments are conducted on 14 large-scale log parsing datasets in Loghub-2.0 [19] from LogPAI [55]. The results show that ULog substantially outperforms the unsupervised baselines, surpassing Brain [51] and LILAC w/o ICL [18] by 46.2% and 26.9% in FTA, respectively. When compared with label-dependent methods, ULog achieves performance on par with the current state-of-the-art parser, LILAC. Moreover, with parallelization capability, ULog attains a parsing speed comparable to most syntax-based baselines and superior than semantic-based baselines, facilitating efficient parsing of large-scale log data. Our evaluation shows the potential of ULog for deployment in real-world production systems, where the efficiency, accuracy and generalizability are critical concerns.

To sum up, the main contributions of this work are as follows:

- To the best of our knowledge, we propose the first unsupervised LLM-based log parser dubbed ULog, which instructs LLMs to make comparisons on log contrastive units (LCUs).
- To enable efficient LCU sampling, we introduce a hierarchical sharding scheme, which reduces sample overhead and allows parallel parsing. Moreover, we propose a hybrid method to measure the LCU by balancing the commonality and variability.
- We evaluate ULog on large-scale public datasets. The results show that ULog significantly outperforms unsupervised parsers and achieves comparable performance with state-of-the-art LILAC in terms of accuracy and efficiency.

2 MOTIVATION

2.1 Limitations of Semantic-based Log Parsers

Recently, semantic-based log parsers [23, 30] have gained significant attention, outperforming traditional syntax-based methods by a considerable margin [19, 23]. The improvement stems from the use of language models (*e.g.*, RoBERTa [28] and ChatGPT [1]) to comprehend the semantics of log messages, enabling precise distinction between static template and varying parameters in the log messages. However, these parsers require labelled examples (*i.e.*, log messages and their corresponding ground-truth log templates) to tailor the language models for log parsing tasks. For instance, the state-of-the-art log parser, LILAC [18], employs labelled samples to perform in-context learning (ICL) to guide language models in producing templates, which is a crucial component to ensure its effectiveness.

Despite the promising results reported, implementing these semantic-based log parsers in practice is challenging due to the difficulty of collecting sufficient labelled data to achieve their full effectiveness. The reasons behind this are two-fold. Firstly, manually collecting labelled examples from log data is labor-intensive and error-prone, which requires extensive domain expertise. Secondly, software frequently undergoes changes [46, 49, 53], resulting in new semantics and patterns in the log data (*i.e.*, concept drift [10]). Consequently, the performance of these log parsers tends to degrade over time.

To quantitatively understand the impact of labelled examples on parsing performance, we re-evaluated two representative semantic-based log parsers, *i.e.*, LogPPT [23] and LILAC [18], using varying label proportions. Specifically, our study utilized large-scale log parsing datasets from Loghub-2.0 [19], which contain 50.4 million log messages from 14 real-world software systems. For each dataset, we randomly sampled different proportions of labelled log templates as the model-accessible oracles, which are then applied for fine-tuning or ICL. The proportion ranges among 75%, 50%, 25%, 10%, and 5%, simulating real-world scenarios where oracle labelled templates are scarce due to insufficient manual labeling or system evolution. We measured their effectiveness using the widely adopted F1 score of template accuracy (FTA) [21]. To reduce the bias introduced by randomness, we performed the experiments five times for each setting, following previous studies [19, 23, 49, 55]. We reported the average FTA scores, as shown in Figure 2.

We observe that the performance of both LogPPT and LILAC significantly declines as the label proportion decreases. For example, when the label proportion is 75%, LogPPT achieves a performance

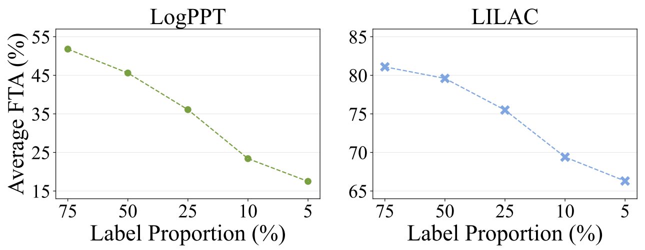


Figure 2: Empirical study on the influences of label proportion on semantic-based log parsers.

of over 50%. However, when the label proportion drops to 5%, its performance falls to about 17%, indicating a substantial decrease of 33%. Similarly, with a label proportion of 75%, LILAC attains an average FTA score of approximately 81%. When the proportion decreases to 25%, the score drops to about 75%. At a label proportion of just 5%, the score further declines to around 66%, representing a reduction of over 15% compared to the highest score. These results suggest that the proportion of labelled examples can significantly impact the performance of semantic-based log parsers, posing challenges to apply them into real-world software systems. Therefore, we aim to develop a label-free semantic-based log parser, which can avoid label insufficiency problems and generalize to unseen software without labelled templates.

2.2 Log Contrastive Unit for LLM-based Parsing

In this section, we present our motivation for developing a label-free, LLM-based unsupervised log parsing approach.

Existing solutions rely on labelled data to guide language models in inferring the parameters from log messages. In contrast, *our core insight is that the LLM itself can derive sufficient hints by comparing multiple logs.* For example, by comparing the logs: "session opened for user news" and "session opened for user test", LLMs can easily infer that "news" and "test" are likely username parameters, because these segments vary while the rest of the log message remains consistent. We aim to leverage such comparisons to guide LLM without the need for labelled data.

We define such grouped log messages as a *log contrastive unit (LCU)*, which consists of multiple log messages potentially sharing the same template, allowing LLMs to parse them through comparisons. Constructing an effective LCU that guides the LLM to produce accurate parsing results is non-trivial. Specifically, log messages within the same LCU need to exhibit both commonality and variability:

- Commonality: These log messages should share some common tokens. If they differ significantly, they are likely generated by different logging statements and are not different templates. For example, logs in LCU-3 differ greatly, thus failing to provide proper hints to LLMs.
- Variability: These log messages should differ in some of their tokens. If they are all identical, the LLM might mistakenly interpret all tokens as constants. For instance, the IP address in logs of LCU-4 is identical, which can mislead the LLM to regard the parameter parts as constants.

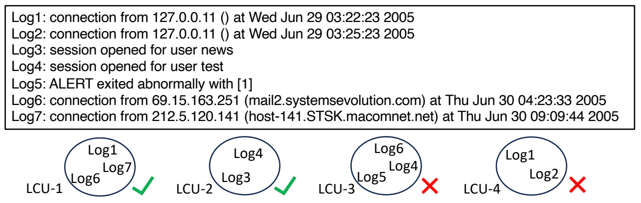

```
Log1: connection from 127.0.0.11 () at Wed Jun 29 03:22:23 2005
Log2: connection from 127.0.0.11 () at Wed Jun 29 03:25:23 2005
Log3: session opened for user news
Log4: session opened for user test
Log5: ALERT exited abnormally with [1]
Log6: connection from 69.15.163.251 (mail2.systemsevolution.com) at Thu Jun 30 04:23:33 2005
Log7: connection from 212.5.120.141 (host-141.STSK.macomnet.net) at Thu Jun 30 09:09:44 2005
```

Figure 3: Examples of log contrastive units (LCUs).

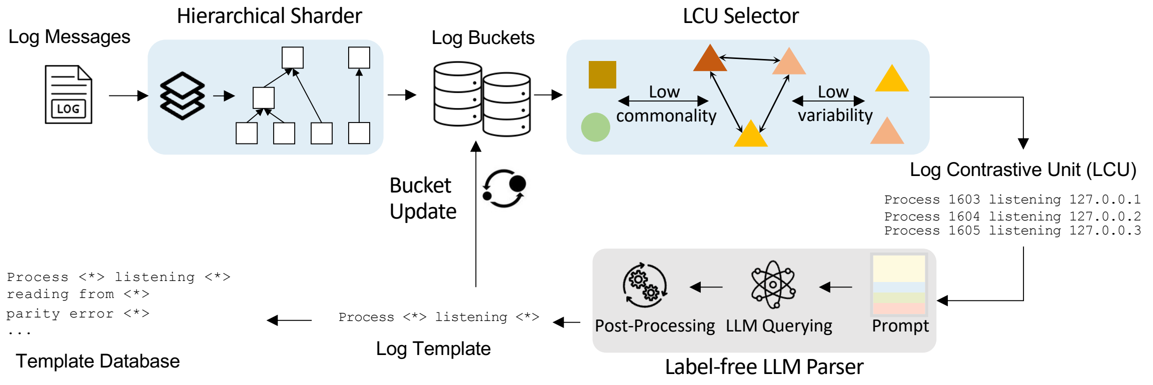
LCU-1 and LCU-2 are examples of well-formed LCUs, as log messages within them exhibit both commonality and variability, guiding the LLM to infer the templates and parameters through comparison. However, we observe the following challenges in practice:

- **Challenge 1: LCU Volume Explosion**: The sheer volume of log data generated by modern software systems can lead to an exponential increase in the number of potential LCUs. This explosion in combinations makes it challenging to scale the LCU-based approach efficiently.
- **Challenge 2: Balance of Commonality and Variability**: Log messages often exhibit a high degree of diversity, with templates containing vastly different tokens. This diversity makes it difficult to identify effective LCUs that strike a balance between commonality and variability. Finding the right combinations of log messages that share enough common tokens while still exhibiting variability is a complex task, further complicated by the dynamic nature of log data and the need for continuous adaptation.

3 METHODOLOGY

3.1 Overview

Figure 4 illustrates the overall framework of ULog, which consists of three main components: *hierarchical sharder*, *generative LCU ranker*, and *label-free LLM parser*. To address the first challenge, we propose the hierarchical sharder, which divides raw log messages into different buckets based on log length and top-k ranked tokens. By separating logs with low similarity into different buckets, this component reduces the sampling overhead and enables parallelization for efficient parsing (§3.2). To address the second challenge, we propose the generative LCU ranker, which operates within each bucket to continuously sample LCUs for the LLM to parse. This module computes a hybrid LCU score that jointly considers both the commonality and variability of the LCUs, guiding the sampling process to ensure effective LCUs (§3.3). Lastly, the label-free LLM parser constructs an organized prompt for the mined LCUs to query the LLM and obtain the templates from the LLM's response. We introduce a novel format for organizing the parsing prompt, which specifies the task intention and output constraints, and includes several representative parameter examples to inform the LLM of the parameter characteristics (§3.4). The combination of all components ensures that the LLM can effectively parse the logs without the need for labelled data, addressing both scalability and diversity challenges.



```
Process 1603 listening 127.0.0.1
Process 1604 listening 127.0.0.2
Process 1605 listening 127.0.0.3
```

```
Process <*> listening <*>
```

```
Process <*> listening <*>
reading from <*>
parity error <*>
...
```

Figure 4: The overall workflow of ULog.

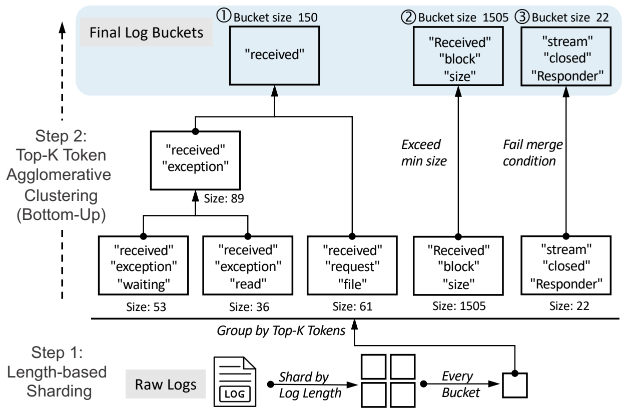

Final Log Buckets
① Bucket size 150
② Bucket size 1505
③ Bucket size 22
"received"
"Received" "block" "size"
"stream" "closed" "Responder"
Step 2: Top-K Token Agglomerative Clustering (Bottom-Up)
"received" "exception"
Size: 89
Exceed min size
Fail merge condition
"received" "exception" "waiting"
"received" "exception" "read"
"received" "request" "file"
"Received" "block" "size"
"stream" "closed" "Responder"
Size: 53
Size: 36
Size: 61
Size: 1505
Size: 22
Group by Top-K Tokens
Step 1: Length-based Sharding
Raw Logs
Shard by Log Length
Every Bucket

Figure 5: Hierarchical log sharding in ULog.

3.2 Hierarchical Log Sharder

Directly sampling LCUs from the whole log messages is overwhelming due to the large volume of log data, which often contain million of log messages. Hence, ULog first groups the log messages into multiple log buckets, where each bucket contain logs that share some similarity. By doing so, extremely different logs, which have less potentials to belong to the same template [14], are placed to different buckets. Specifically, we propose a hierarchical algorithm to shard log: 1) firstly divide the log messages by log length, and 2) progressively divide logs by top-k frequent tokens. Figure. 5 demonstrates the workflow of hierarchical log sharder.

3.2.1 Length-based Sharding. ULog first shards the log messages by the log length. Logs with the same number of tokens are grouped in a bucket, which is a widely adopted heuristic to conduct initial grouping in log analysis [14, 24]. Here we simply adopt the white-space as the delimiter. We do not use other delimiters such as ':' because most of these delimiters appear in parameters and introducing them can produce over-fragmented logs, leading to a noisy sharding result.

3.2.2 Top-k Token Agglomerative Clustering. However, simply sharding by log length is too coarse-grained since logs belonging to different templates could have the same length [18]. Thus ULog continuously shards each bucket into more purified ones where less templates are included in one bucket. Intuitively, the static parts of a log generally have higher occurrences than its parameter part, therefore logs that share the most frequent tokens have more potential to belong to the same template [18, 19, 27]. To achieve this, we introduce a *hierarchical agglomerative clustering* algorithm based on top-k frequent tokens, which builds the sub-buckets in a bottom-up way.

Specifically, for each bucket obtained from the first step, we first group logs into singleton clusters based on the same top-k tokens in each log. The top-k tokens of a log is an ordered list extracted based on the token frequency and position. The frequency is counted among logs in the bucket. The more frequent tokens takes a more preceding place in the list. However, some log messages can have multiple tokens with the same frequency, which leads to confusion in ranking; hence we additionally rank tokens with the same frequency by their positions in the logs, with the preceding token being ranked before the following token.

Then, we iteratively merge the singleton clusters to obtain final log buckets until a stopping criterion is met. In each iteration, if a cluster contains more than N logs, it will be directly nominated as a standalone bucket and no further merge operations will be conducted on it. For the remaining clusters, which contain less than N logs, we merge those with the same top-$(k-1)$ tokens to form a parent cluster. The iteration stops if all clusters exceed N logs, or no any two clusters satisfy the merge condition. Finally, we obtain a collection of buckets containing a smaller number of logs with higher relevance.

It is worth noting that the collected log buckets have a two-level hierarchy, where each hyper-bucket obtained by length-based sharding is disjointly separated to multiple final buckets. Therefore, buckets under the same level are mutually exclusive with each other, *i.e.*, every log belongs to one bucket on either sharding level. The advantages of this design are two-fold. Firstly, once a template is obtained, it can be leveraged to match the logs in the bucket while not required for other buckets. Secondly, this characteristic enables parallel parsing, where the buckets can be independently allocated on multiple executors. More details will be described in Section 3.6

3.3 Two-stage LCU Selector

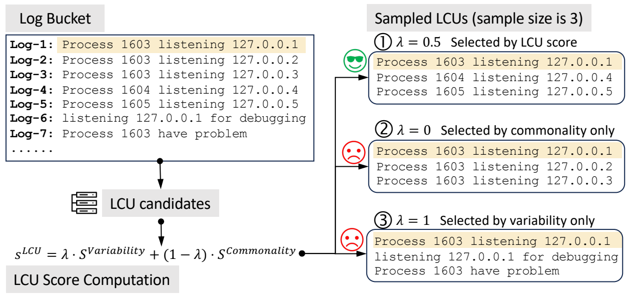
After log sharding, ULog iteratively select a group of similar logs for the LLM to make comparisons and extract templates. A suitable group of logs should be similar in tokens to each other, as logs generated by the same logging statement share several identical tokens. On this basis, the logs are expected to have as much variance as possible, enabling LLMs to make cross-log comparisons to infer parameters. For example, in Figure 6, group ① and ② are preferred than group ③ as their logs are more likely to belong to the same template. Moreover, group ① is better than ② due to its higher variance reflected in the diverse parameter value of process index.

We term such a group of logs as a log contrastive unit (LCU). Logs in each LCU have both commonality (*e.g.*, with common words or belonging to the same template) as well as variability (*e.g.*, with different tokens), so that it can be leveraged to infer the parameters. To collect LCUs, we leverage a two-step sampling approach: *stratified LCU generation* and *hybrid LCU nomination.*

3.3.1 Stratified LCU Generation. Given a bucket of logs, ULog first generate multiple candidate LCUs that share some similarity. A straightforward approach is to enumerate all possible log combinations. However, this method is computationally expensive due to the sheer volume of logs and the complexity of enumeration. To reduce computation overhead, we apply stratified sampling to collect a limited-sized log pool by sampling from different similarity levels before perform combination.

Given a log bucket, ULog first randomly selects a log message in the bucket as the anchor log. Next, ULog computes the pair-wise similarities of the anchor log to the remaining logs. We use Jaccard similarity, which is a widely adopted metric to measure pair-wise log similarity [14, 18]. Specifically, we first split the log l into tokens $T(l)$ with white-space delimiter and then compute the Jaccard similarity $JS(l_1, l_2) = \frac{T(l_1) \cap T(l_2)}{T(l_1) \cup T(l_2)}$. Then, we remove the logs with a similarity below a threshold s or equals to 1.0. The poor similarity indicates the less possibility to belong to the same template, while a similarity of 1.0 indicates duplication. After that, we create a pool of logs by stratified sampling from each similarity level to ensure balanced sampling. Specifically, we set the sample size of each level equal to the LCU size m to involve more diverse logs. Lastly, from the log pool, ULog generates multiple LCUs by combinatorial enumeration. Each LCU consists of an anchor log, as well as $m-1$ logs chosen from the log pool. Since we sample a small size of LCU (*e.g.*, $m = 3$ or $m = 4$) from a limited size of log pool, the computational cost of combinatorial enumeration is within an acceptable range (More details about efficiency can found in Section 5.3). If the log pool contain less than $m-1$ logs, we directly return the anchor with the pool as the candidate LCU. Finally, a candidate set of LCUs are generated and proceeded to hybrid rankding in the next step.

3.3.2 Hybrid LCU Ranking. After obtaining a small number candidate LCUs, ULog conducts hybrid ranking to select the most suitable LCU. The selected logs are required to exhibit variability, while maintain some commonality as we discussed in Section 2.2. By doing so, LLMs can be less likely to incorrectly recognize the frequent tokens as parameters, while also benefit from the contrastive


Log Bucket
LCU candidates
$s^{LCU} = \lambda \cdot S^{Variability} + (1-\lambda) \cdot S^{Commonality}$
LCU Score Computation
Sampled LCUs (sample size is 3)
① $\lambda = 0.5$ Selected by LCU score
② $\lambda = 0$ Selected by commonality only
③ $\lambda = 1$ Selected by variability only

```
Log-1: Process 1603 listening 127.0.0.1
Log-2: Process 1603 listening 127.0.0.2
Log-3: Process 1603 listening 127.0.0.3
Log-4: Process 1604 listening 127.0.0.4
Log-5: Process 1605 listening 127.0.0.5
Log-6: listening 127.0.0.1 for debugging
Log-7: Process 1603 have problem
......
```

```
Process 1603 listening 127.0.0.1
Process 1604 listening 127.0.0.4
Process 1605 listening 127.0.0.5
```

```
Process 1603 listening 127.0.0.1
Process 1603 listening 127.0.0.2
Process 1603 listening 127.0.0.3
```

```
Process 1603 listening 127.0.0.1
listening 127.0.0.1 for debugging
Process 1603 have problem
```

Figure 6: An example of expected LCU in LCU nomination.

information in the LCU. Specifically, we propose to compute the variability score and the commonality score to measure an LCU.

Firstly, the variability score of an LCU is computed as average of pair-wise distances to measure the overall variability of the logs. Suppose an LCU consists of L logs $LCU = \{l_1, l_2, ...l_L\}$, we first compute the distance between any two logs in the LCU based on Jaccard similarity:

$$dist(l_i, l_j) = 1 - JS(l_i, l_j). \tag{1}$$

Then we average the distances of every two logs in the LCU to obtain the variability score:

$$S_{LCU}^{Variability} = \frac{2}{L(L-1)} \sum_{i=1}^{L} \sum_{j=i+1}^{L} dist(l_i, l_j), \tag{2}$$

where the higher the variability score, the more diverse the LCU, indicating more contrastive information is included.

Simply ranking by variability could result in LCUs containing totally different logs (*e.g.*, LCU ③ in Figure 6). As the compensation, the commonality score of an LCU is introduced, which is computed as the average of absolute similarity difference to balance the variance of logs. Specifically, we P log pairs can be obtained from an LCU, then:

$$S_{LCU}^{Commonality} = \frac{2}{P(P-1)} \sum_{i=1}^{P} \sum_{j=i+1}^{P} (1 - |JS(p_i) - JS(p_j)|), \tag{3}$$

where $JS(p_i)$ is the Jaccard similarity of log pair p_i. The higher the commonality score, the more likely that logs belong to the same template, as the similarity difference between pair to pair is relatively low. Logs that share the same template but differ only in their parameters will exhibit exactly such a pattern. For example, in the LCU ② of Figure 6, the similarity between any two logs is the same as any other two logs within the LCU, which implies the highest commonality score of 1.

To jointly consider the variability and commonality scores, we combine them with linear interpolation with a weight of λ, to obtain the hybrid LCU score. The LCU with maximum interpolation score selected as the final LCU, which is computed as follows:

$$S_{LCU} = \lambda \cdot S_{LCU}^{Variability} + (1-\lambda) \cdot S_{LCU}^{Commonality}. \tag{4}$$

To present the LCU score in a straightforward way, we provide an example with a sample size of 3 in Figure 6. In this example, by selecting with variability score only, LCU ③ will be sampled.

However, LCU ③ is sub-optimal as the logs belong to three different templates. LLMs might mis-classify the parameter “1603” as this group fails to provide useful contrasts towards it. With the commonality score only, LCU ② will be sampled. However, this is also sub-optimal due to the identical parameter “1603”. This LCU could mislead LLMs to incorrectly recognize parameters due the identical tokens. In contrast, by combining the variability score and commonality score, we can obtain an optimal LCU, where each parameter has at least two values present. Using this LCU can be maximumly inform LLM to consider the changing tokens, thereby leading to an improved parsing accuracy.

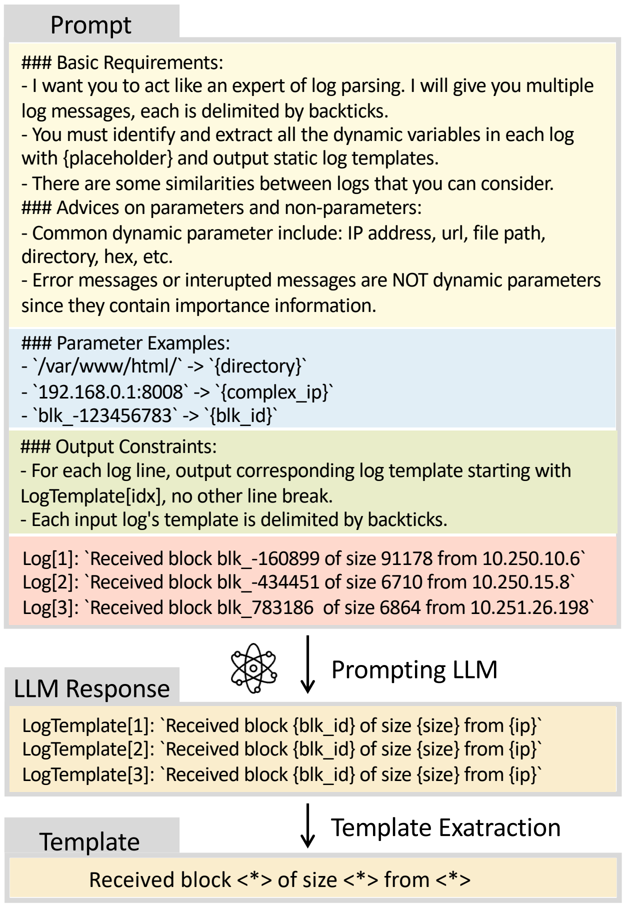
3.4 Label-free LLM Parser

Upon obtaining an LCU, ULog applies an LLM-based parser to extract the template without any labelled examples. The parser leverages the strong textual comprehension ability of LLMs, which have shown remarkable few-shot and zero-shot performance in various information extraction tasks, such as named entity recognition [47] and text classification [31]. Therefore, we believe LLMs have the potential to solve log parsing in an unsupervised way. Specifically, our LLM parser first creates a well-designed prompt for each LCU to instruct the LLM (§3.4.1) and then extract the template from the LLM response (§3.4.2).

3.4.1 Prompt Design. As LLMs are not specifically tuned for log parsing, existing LLM-based log parsers leverage exemplars of log and its labelled template to instruct the task [18, 49]. These demonstrations could specify task intents, output format, and parameter information of the domain, which are useful to enhance log parsing [32, 35]. However, in this work, we focus on developing an unsupervised parser where no labelled templates are provided. Therefore, we need to design a more concrete prompt covering the aforementioned demanding information to facilitate accurate parsing. Specifically, our prompt contains four parts: task instructions, parameter examples, output constraints, and input LCUs. Figure 7 shows a prompt example.

Task Instruction. A clear and useful prompt should provide comprehensive instructions for LLMs to understand the task [13]. In LUNAR, we construct the task instruction of log parsing with two parts: basic requirements and general advice on parameters. For the basic requirements, the task input/output and the mapping mechanism from input to output are firstly specified. The instruction also guides the LLM to consider the similarity between the grouped input logs. For the general advice on parameters, high-level descriptions of parameter categories are provided. We also instruct the LLM not to consider the error messages as parameters since they also contain some important information.

Parameter Examples. Inspired by the entity examples used for unsupervised NER [17], we involve parameter examples to the prompt. These examples are used to provide domain knowledge and instruct the task, which consist of a parameter type and a parameter value. For instance, “`/var/www/html/`” is the value of a parameter “directory.” To obtain the parameter examples, we first manually check the regular expressions of used in previous studies [14, 25], then sample a value from matched parameters, and finally summarize the parameter type of the value [25]. In the prompt, the parameter value and type are connected by an



```
### Basic Requirements:
- I want you to act like an expert of log parsing. I will give you multiple log messages, each is delimited by backticks.
- You must identify and extract all the dynamic variables in each log with {placeholder} and output static log templates.
- There are some similarities between logs that you can consider.
### Advices on parameters and non-parameters:
- Common dynamic parameter include: IP address, url, file path, directory, hex, etc.
- Error messages or interupted messages are NOT dynamic parameters since they contain importance information.
### Parameter Examples:
- `/var/www/html/` -> `{directory}`
- `192.168.0.1:8008` -> `{complex_ip}`
- `blk_-123456783` -> `{blk_id}`
### Output Constraints:
- For each log line, output corresponding log template starting with LogTemplate[idx], no other line break.
- Each input log's template is delimited by backticks.
Log[1]: `Received block blk_-160899 of size 91178 from 10.250.10.6`
Log[2]: `Received block blk_-434451 of size 6710 from 10.250.15.8`
Log[3]: `Received block blk_783186  of size 6864 from 10.251.26.198`
```

```
LogTemplate[1]: `Received block {blk_id} of size {size} from {ip}`
LogTemplate[2]: `Received block {blk_id} of size {size} from {ip}`
LogTemplate[3]: `Received block {blk_id} of size {size} from {ip}`
```

```
Received block <*> of size <*> from <*>
```

Figure 7: Demonstration-free parsing prompt in ULog.

arrow, aiming to indicate the task intent to convert the value to a placeholder with the bracket.

Output Constraints. In output constraints, we explain the desired output formats. The prompt instructs the LLM to produce a template for every log message. Each generated template should be appended by a prefix of “LogTemplate[idx]” and delimited with backticks to facilitate template aggregation and extraction.

Queried LCUs. Finally, the LCU with multiple logs are incorporated to the prompt. We arrange the logs in a sequential order and apply a prefix schema (*i.e.*, “Log[]:”) and index indicator to indicate the order. In this way, the LLM can make straightforward comparisons and produce corresponding template for each log.

Guided by the task instruction, parameter examples, and output constraints, the LLM could more accurately generate the templates of the queried logs in the LCU.

3.4.2 Template Acquisition. After constructing the prompt, we leverage the LLM to identify the templates in the LCU. Specifically, we first leverage prompt to query an LLM to get a response. After that, we perform post-processing to extract valid templates from the response. Specifically, we extracts all backticked strings with a “LogTemplate” prefix, thanks to clear output constraints in the prompt. Then we aggregate the extracted templates into one template by selecting the most frequent template. Finally, we replace the bracketed parameters by the placeholder $< * >$.

3.5 Bucket Updating

After obtaining a valid template, ULog examines the logs that match the template. The template is assigned as the final prediction to these successfully matched logs, which are then removed from the buckets. The remaining log buckets are used for next iteration of parsing. Once all buckets are empty, the parsing terminates.

It is worth-noting that the the hierarchical sharding produces buckets at two levels, *i.e.*, sharded by length and further clustered by top-k tokens. A straightforward approach for bucket updating is to only examine and update the final bucket. However, logs in the same template can be placed in different buckets of equal length, potentially causing redundant iterations for one template, leading to prolonged parsing time and increased LLM querying expenses. To balance the parsing overhead, ULog conducts template examination and updating on the buckets under the first sharding level, meaning that buckets with identical log lengths are all examined once a template is obtained.

3.6 Parallelization

In previous sections, we have discussed the building components of ULog. In practice, scalability is a crucial issue when parsing vast amounts of log messages [46]. Therefore, we introduce the parallelization mechanism of ULog to improve the parsing speed.

As mentioned in §3.2, hierarchical sharding produces mutually exclusive log buckets at two levels. Consequently, logs in different buckets can be processed in a completely parallel fashion. However, since bucket updating is performed among all the buckets of the same length at the first level. Simply allocating buckets at the second level to produce template and update buckets can lead to *template conflict*, *i.e.*, different templates being assigned to the same log. To address the problem, the allocation is conducted at the first level. Specifically, we aim to allocate n executors for all first-level log buckets. Within each first-level log bucket, ULog iteratively selects the largest second-level bucket, selects an LCU, queries the LLM to obtain the template, and updates buckets under this level.

4 EXPERIMENT SETUP

We conduct extensive experiments to evaluate ULog by answering the following research questions (RQs):

- **RQ1:** How effective is ULog?
- **RQ2:** How do different settings affect ULog?
- **RQ3:** How efficient is ULog?

4.1 Datasets

We evaluate ULog on Loghub-2.0 [16, 19], a collection of large-scale benchmark datasets for log parsing provided by LogPAI team [55]. Loghub-2.0 is annotated with ground-truth templates for 14 diverse log datasets, which covers a variety of systems such as distributed systems, supercomputer systems, and server-side applications. Compared with the original Loghub [16], Loghub-2.0 is equipped with $3\times$ templates and $1,800\times$ log messages on average, which can support more comprehensive evaluation on accuracy and efficiency. On average, each dataset in Loghub-2.0 comprises 3.6 million log messages. In total, the collection contains approximately 3,500 unique log templates.

4.2 Baselines

We compare ULog with seven open-sourced state-of-the-art log parsers, including four unsupervised syntax-based parsers and three label-required semantic-based parsers. For unsupervised label-free parsers, we adopt three syntax-based methods and one LLM-based method. The syntax-based methods include AEL [20], Drain [14], and Brain [51], chosen for their superior performance compared to other syntax-based methods [19, 21, 55]. As there are currently no LLM-based log parsers proposed for label-free parsing, we adopt a label-free variant of LILAC [18] (LILAC w/o ICL) by removing the in-context learning (ICL) module. In terms of label-required log parsers, we select three state-of-the-art semantic-based log parsers: UniParser [30], LogPPT [23], and LILAC [18]. UniParser trains a long short-term memory (LSTM) model on labelled log data for log parsing. LogPPT uses labelled log data to perform prompt-based fine-tuning based on RoBERTa [28]. LILAC samples similar logs from a small set of labelled logs (*e.g.*, 32 logs) for each log, utilizing an ICL paradigm for LLMs to parse logs.

4.3 Evaluation Metrics

Following previous works [19, 21, 55], we evaluate ULog with the following four metrics.

- *Grouping Accuracy* (GA) [55] is a log-level metric that measures the the amount of log messages of a same template are grouped together by the parser. It is computed as the ratio of correctly grouped log messages over all log messages, where a log message is regarded as correctly grouped if and only if its predicted template have the same group of log messages as the oracle.
- Parsing Accuracy (PA) [7] is a log-level metric that measures the correctness of extracted templates and variables. It is defined as the ratio of correctly parsed log messages over all log messages, where a log message is considered to be correctly parsed if and only if all its static text and dynamic variables are identical with the oracle.
- F1 score of Grouping Accuracy (FGA) [19] is a template-level metric that measures the ratio of correctly grouped templates. It is computed as the harmonic mean of precision and recall of grouping accuracy, where the template is considered as correct if log messages of the predicted template have the same group of log messages as the oracle.
- F1 score of Template Accuracy (FTA) [21] is a template-level accuracy computed as the harmonic mean of precision and recall of Template Accuracy. A template is regarded as correct if and only if it satisfies two requirements: log messages of the predicted template have the same group of log messages as the oracle, and all tokens of the template is the same as the oracle template.

4.4 Environment and Implementation

We conduct all the experiments on a Ubuntu 20.04.4 LTS server with 256RAM and an NVIDIA A100 40G GPU. The LLM used in ULog is ChatGPT (gpt-3.5-turbo-0613) due to its popularity in recent log analysis studies [18, 32, 49]. We invoke the official API provided by OpenAI [1] and set the temperature to 0 to avoid randomness in token generation and ensure reproducibility. By default, we set the a top-k token number to 3 and minimum cluster size to 100 in our hierarchical log sharder. For the LCU selector, we use an LCU sample size of 3, a interpolation factor λ of 0.7, and a minimum similarity of 0.33, respectively. For parallelization, we distribute the jobs to 8 executors. We also conduct experiments of ULog with different LCU sample sizes and interpolation factors. As for baseline methods, we directly use the accuracy reported in previous work [19] and rerun them with their default parameters on the same environment to fairly compare the efficiency.

5 EVALUATION RESULTS

5.1 RQ1: Effectiveness

Setup: In the first research question (RQ), we evaluate the accuracy of ULog by comparing it with state-of-the-art log parsers, as accuracy is the most critical factor for log parsers. For comprehensiveness, we compare ULog with both unsupervised parsers, which do not require human-labelled log templates, and label-required semantic-based log parsers, which rely on labelled log templates to support training, fine-tuning, or in-context learning. Table. 1 presents the overall results on four metrics for ULog compared to baseline methods. The best results for each metric on each dataset are highlighted in **bold**, while the second-best results are underlined. If a specific parser could not complete the parsing process within a reasonable timeframe (*e.g.*, 12 hours), as per previous work [18, 19, 21, 55], we denote the score as "-".

Results: Overall, we observe that ULog achieves the best average grouping accuracy (GA), parsing accuracy (PA), and F1 score of template accuracy (FTA), as well as the second-best F1 score of grouping accuracy (FGA). Additionally, ULog demonstrates high accuracy across all 14 datasets, showcasing its robustness in parsing log data from diverse systems. For example, ULog achieves the highest FTA on 11 out of 14 datasets and the second-highest FTA on the remaining 3 datasets.

Among unsupervised log parsers, it is evident that ULog significantly outperforms the other four baselines across all four evaluation metrics. Among the syntax-based parsers (*i.e.*, AEL, Drain, and Brain), Brain achieves the highest scores in template-level metrics, with an average FGA of 71.8% and an average FTA of 35.4%. However, LUNAR exhibits a substantially higher average FGA of 91.4% and FTA of 81.6%, surpassing Brain by 19.6% and 46.2%, respectively. This is primarily due to the limitations of syntax-based parsers, which rely solely on manually crafted rules and thus struggle with complex and diverse log data. Furthermore, leveraging the extensive pre-trained knowledge of LLMs, LILAC w/o ICL demonstrates superior performance than syntax-based methods in PA, FGA, and FTA. However, our label-free LLM-based parser, ULog, significantly outperforms LILAC w/o ICL, with an average improvement of 15.2% in FGA and 26.9% in FTA, respectively. These substantial improvements underscore the effectiveness of ULog in harnessing the zero-shot capabilities of LLMs without labelled examples.

Compared with supervised baseline parsers, ULog demonstrate comparable performance to existing state-of-the-art parser LILAC. Specifically, compared to LILAC, ULog has achieved superior performance in average GA (93.4%), PA (88.4%), and FTA (81.6%), while showing a slightly lower average FGA score (91.4%). LILAC retrieves similar logs and labelled templates to provide demonstrations during parsing, which are often inaccessible in real-world systems. However, ULog achieves a comparable performance to the state-of-the-art while does not need labelled templates, which is more robust and generalizable in practice. Additionally, compared to UniParser and LogPPT, ULog has achieved substantial improvements with an average improvements of 30.2% and 28.7% in all four metrics. This result again highlights the advantages of ULog, which is not also effective, but also requires no labelled data for training. The comparable performance of ULog indicates its potential to be applied in real-world production systems, which can address the label dependency problems of semantic-based log parsers as mentioned in Section 2.1.

5.2 RQ2: Impact of different settings

5.2.1 *Designs.* In this research question, we first assess the individual contributions of the designs in the two components of ULog: the hierarchical log sharder and the two-stage LCU selector. To accomplish this, we have implemented several variants of ULog by either removing or replacing the designs in these components. Specifically, for the hierarchical log sharder, we created two variants by removing the respective clustering stages. Additionally, for the two-stage LCU selector, we replaced the hybrid LCU ranking algorithm with various alternatives: random selection, simple selection based on minimum or maximum similarity, and selection of consecutive log messages for querying the LLMs. To mitigate the impact of randomness, we also repeated experiment five times and calculated the mean scores as final results.

The average metric scores across all datasets are presented in Table 2. When the log length-based clustering is removed from the log sharder, the end-to-end accuracy of ULog drops from 93.4% to 87.7% and from 88.4% to 82.3%, respectively. Similar decreases are observed for the variant without top-k tokens clustering. These results indicate that both stages within the hierarchical log sharder significantly contribute to the overall performance of ULog. On the other hand, for the four variants related to the two-stage LCU selector, all four metrics show decreases across the board. For example, replacing the hybrid LCU ranking algorithm with random selection results in average GA and PA scores dropping by 7.4% and 6.6%, respectively. Notably, selecting the LCU based on maximum similarities among log messages has the most detrimental impact on ULog's performance, resulting in a 10.2% decrease in GA and an 8.7% decrease in PA. This is primarily because selecting only the most similar log messages as an LCU for prompting prevents the LLM from accurately identifying templates and parameters by contrasting the tokens of these log messages. These results demonstrate that our proposed two-stage LCU selector is effective in selecting LCUs, thereby enabling LLMs to accurately parse log messages.

5.2.2 *Configurations.* In addition to the two key designs of ULog mentioned above, we have identified two configurations that could significantly affect ULog's performance: the LCU sample size and the λ values for LCU nomination. These configurations directly determine the LCUs for each query provided to the LLMs, and thus, they may significantly impact the accuracy of the LLMs' parsed results. In this part, we explore different settings for the LCU sample

Table 1: Accuracy of ULog compared to state-of-the-art baselines on public datasets. (%)

Method	Metric	Proxifier	Apache	OpenSSH	HDFS	OpenStack	HPC	Zookeeper	HealthApp	Hadoop	Spark	BGL	Linux	Mac	Thunderbird	Average
Unsupervised Log Parsers																
AEL	GA	97.4	**100.0**	70.5	99.9	74.3	74.8	99.6	72.5	82.3	—	91.5	91.6	79.7	78.6	85.6
	PA	67.7	72.7	36.4	62.1	2.9	74.1	84.2	31.1	53.5	—	40.6	8.2	24.5	16.3	44.2
	FGA	66.7	**100.0**	68.9	76.4	68.2	20.1	78.8	0.8	11.7	—	58.7	80.6	79.3	11.6	55.5
	FTA	41.7	51.7	33.3	56.2	16.5	13.6	46.5	0.3	5.8	—	16.5	21.7	20.5	3.5	25.2
Drain	GA	69.2	**100.0**	70.7	99.9	75.2	79.3	99.4	86.2	92.1	88.8	91.9	68.6	76.1	83.1	84.3
	PA	68.8	72.7	58.6	62.1	2.9	72.1	84.3	31.2	54.1	39.4	40.7	11.1	35.7	21.6	46.8
	FGA	20.6	**100.0**	87.2	93.5	0.7	30.9	90.4	1.0	78.5	86.1	62.4	77.8	22.9	23.7	55.4
	FTA	17.6	51.7	48.7	60.9	0.2	15.2	61.4	0.4	38.4	41.2	19.3	25.9	6.9	7.1	28.2
Brain	GA	52.1	99.7	66.3	96.0	**100.0**	80.0	99.3	97.9	56.3	97.2	94.0	79.0	83.4	79.2	84.3
	PA	70.3	28.7	48.1	92.9	14.1	66.3	82.2	17.5	14.3	39.3	40.2	1.0	32.5	26.1	41.6
	FGA	73.7	93.3	75.9	75.9	**100.0**	44.7	79.8	87.2	52.8	20.8	75.6	75.1	75.4	74.8	71.8
	FTA	73.7	46.7	34.5	62.1	29.2	21.3	60.1	33.9	20.0	1.0	19.7	27.5	29.4	27.4	35.4
LILAC w/o ICL	GA	0.0	99.7	74.4	**100.0**	**100.0**	85.6	99.7	99.3	91.5	99.8	88.5	80.4	74.2	72.9	83.3
	PA	11.4	94.2	34.8	94.7	47.7	64.8	35.1	53.6	71.8	58.1	82.3	67.6	48.9	53.9	58.5
	FGA	0.0	93.3	69.7	65.8	**100.0**	89.0	96.0	96.2	91.5	89.0	83.3	82.5	77.8	32.8	76.2
	FTA	16.0	56.7	42.4	52.1	79.2	74.0	73.1	68.2	64.5	58.9	63.7	56.5	41.6	18.4	54.7
Label-required Log Parsers																
UniParser	GA	50.9	94.8	27.5	**100.0**	**100.0**	77.7	98.8	46.1	69.1	85.4	91.8	28.5	73.7	57.9	71.6
	PA	63.4	94.2	28.9	94.8	51.6	94.1	**98.8**	81.7	**88.9**	79.5	94.9	16.4	**68.8**	**65.4**	73.0
	FGA	28.6	68.7	0.9	**96.8**	96.9	66.0	66.1	74.5	62.8	2.0	62.4	45.1	69.9	68.2	57.8
	FTA	45.7	26.9	0.5	58.1	28.9	35.1	51.0	46.2	47.6	1.2	21.9	23.2	28.3	29.0	31.7
LogPPT	GA	98.9	78.6	27.7	72.1	53.4	78.2	96.7	99.8	48.3	47.6	24.5	20.5	54.4	56.4	61.2
	PA	**100.0**	94.8	65.4	94.3	40.6	**99.7**	84.5	**99.7**	66.6	95.2	93.8	16.8	39.0	40.1	73.6
	FGA	87.0	60.5	8.1	39.1	87.4	78.0	91.8	94.7	52.6	37.4	25.3	71.2	49.3	21.6	57.4
	FTA	95.7	36.8	10.5	31.2	73.8	76.8	80.9	82.2	43.4	29.9	26.1	42.8	27.4	11.7	47.8
LILAC	GA	**100.0**	**100.0**	69.0	**100.0**	**100.0**	**86.9**	**100.0**	**100.0**	87.2	**100.0**	89.4	**97.1**	**87.6**	80.6	92.7
	PA	**100.0**	99.6	**94.1**	99.9	**100.0**	70.5	68.7	72.9	83.2	97.3	95.8	76.5	63.8	55.9	84.2
	FGA	**100.0**	**100.0**	83.8	**96.8**	**100.0**	**90.7**	**96.7**	**98.1**	**96.2**	90.1	85.9	**93.1**	**82.5**	79.3	**92.4**
	FTA	**100.0**	**86.2**	86.5	**94.6**	**97.9**	80.0	86.8	87.3	**77.9**	75.9	74.6	**74.0**	55.3	57.2	81.0
Our proposed method																
LUNAR	GA	**100.0**	**100.0**	**78.0**	**100.0**	**100.0**	86.4	99.2	99.8	**93.7**	97.5	**94.9**	94.8	77.0	**85.6**	**93.4**
	PA	**100.0**	**99.8**	72.2	**100.0**	98.0	99.3	85.2	98.2	86.0	**99.6**	**98.2**	**85.8**	53.1	62.6	**88.4**
	FGA	**100.0**	**100.0**	**90.0**	**96.8**	**100.0**	80.5	92.0	96.2	88.9	**90.2**	**87.9**	92.4	81.8	**83.0**	91.4
	FTA	**100.0**	**86.2**	**90.0**	**94.6**	93.8	**81.6**	**87.4**	**88.0**	74.0	**76.4**	**80.1**	73.5	**57.0**	**59.9**	**81.6**

Table 2: Ablation study of components in ULog (%)

Metrics	**GA**	**PA**	**FGA**	**FTA**
ULog	93.4	88.4	91.4	81.6
Variants w.r.t Log Sharder				
w/o log length	87.7 (↓ 5.7%)	82.3 (↓ 6.1%)	88.6 (↓ 2.8%)	80.4 (↓ 1.2%)
w/o top-k tokens	91.6 (↓ 1.8%)	84.3 (↓ 4.1%)	90.8 (↓ 0.6%)	81.0 (↓ 0.6%)
Variants w.r.t LCU Selector				
w/ random selection	86.0 (↓ 7.4%)	81.8 (↓ 6.6%)	88.7 (↓ 2.7%)	79.8 (↓ 1.8%)
w/ minimum similarity	88.8 (↓ 4.6%)	83.9 (↓ 4.5%)	89.5 (↓ 1.9%)	80.2 (↓ 1.4%)
w/ maximum similarity	83.2 (↓ 10.2%)	79.7 (↓ 8.7%)	87.2 (↓ 4.2%)	78.6 (↓ 3.0%)
w/ consecutive selection	83.5 (↓ 9.9%)	80.2 (↓ 8.2%)	88.5 (↓ 2.9%)	80.0 (↓ 1.6%)

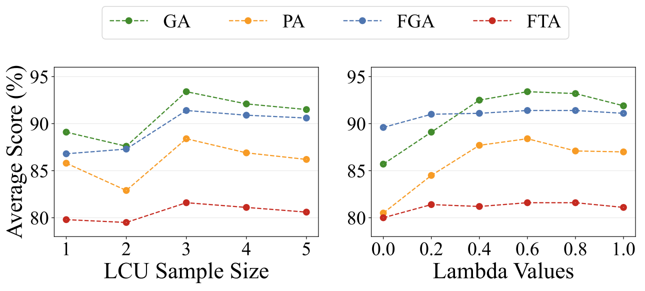


Figure 8: Sensitivity study of configurations in ULog.

size and the λ values to evaluate their impact on ULog's performance across all four evaluation metrics. Specifically, we first chose the default λ value (*i.e.*, 0.5) and varied the LCU sample sizes from 1 to 5. Additionally, we fixed the default LCU sample size (*i.e.*, 3) and varied the λ value from 0.0 to 1.0. The average metric scores under different settings across all datasets are presented in Figure 8.

Based on the results shown in the left sub-figure of Figure 8, we observe that the performance of ULog remains consistently high across different LCU sample sizes. Notably, when the LCU sample size is set to 3, all four metrics achieve their highest scores. Conversely, smaller sample sizes result in lower accuracy. For instance, with a sample size of 1, the average GA and PA scores drop to approximately 89% and 86%, respectively, which is 4.5% and 2.5% lower than the scores achieved with a sample size of 3. This suggests that the absence of contrastive log messages can impair the parsing ability of LLMs. Interestingly, when the sample size is 2, the accuracy decreases further compared to a sample size of 1. This occurs because with only 2 contrastive log messages, the LLM tends

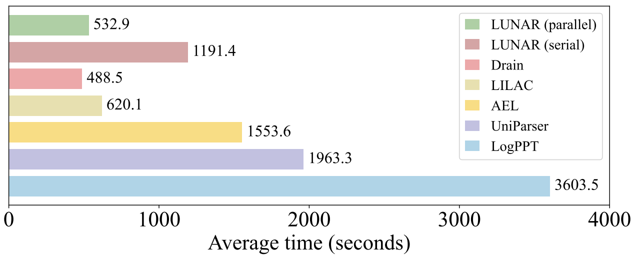


Figure 9: Efficiency of ULog and baselines (second).

to overfit to the differences between them, neglecting potential variable tokens in the log messages. On the other hand, increasing the LCU sample size to 4 or 5 results in a downward trend in accuracy. For example, the FTA score decreases from 81.6% to 80.6%. This indicates that larger LCU sample sizes may introduce additional noise and thus do not enhance the accuracy of ULog. Therefore, we have set the default LCU sample size to 3 in our experiments.

The λ values for LCU nomination take into account both variability and commonality scores when selecting LCUs for prompting. A higher λ value places more emphasis on variability, while a lower λ value prioritizes commonality. As shown in Figure 8, the optimal average accuracy is achieved when the λ value is set to 0.6, indicating a balanced consideration of variability and commonality in log messages within the LCU. When the λ value is low, such as 0.0 or 0.2, all metrics are significantly lower compared to a value of 0.6, underscoring the importance of variability in log messages within LCUs. For instance, at a λ value of 0.0, the average GA and PA scores are both 8% lower than at 0.6. Conversely, performance also declines when the λ value is too high. For example, both GA and PA scores drop by 1.5% when the λ value increases from 0.6 to 1.0. These results illustrate that both variability and commonality scores effectively filters log messages belonging to the same template within the LCUs, which enable LLMs to accurately parse the log template.

5.3 RQ3: Efficiency

Setup: Efficiency is an essential factor for log parsers in real-world usage, given the substantial volume of logs produced [46, 55]. In this RQ, we evaluate the efficiency of ULog and all other baseline parsers by apply them in the large-scale datasets within Logpub, which contains average 3.6 million log messages per dataset. Specifically, we recorded the execution times for all log parsers in parsing all log datasets within Logpub, and then compute the average parsing time across all datasets. Since ULog is designed to enable parallelism for processing log buckets, we compute the average parsing time for both serial mode and parallel model of ULog. The efficiency results are demonstrated in the Figure 9.

Results: The results demonstrate that ULog achieves higher efficiency compared to the most efficient semantic-based log parser, LILAC, and comparable efficiency to the most efficient syntax-based log parser, Drain. Specifically, the average parsing time for 3.6 million log messages in serial mode using ULog is 1191.4 seconds. In practice, we can leverage the parallel mode of ULog to achieve better efficiency, reducing the parsing time to just 532.9 seconds. For the most efficient syntax-based parser, Drain, which only takes an average of 488.5 seconds to parse each dataset, ULog achieve comparable efficiency, only slower by 9%. In contrast, the most efficient semantic-based parser, LILAC, takes an average of 620.1 seconds, which is 16.4% slower than ULog. Notably, even with GPU acceleration, other semantic-based parsers such as UniParser and LogPPT require significantly more time to parse large-scale log data. ULog outperforms them by 3.68 times and 6.75 times in efficiency, respectively. These results indicate that ULog is efficient in parsing large-scale log data, making it suitable for application in real-world production systems.

6 THREATS TO VALIDITY

We have identified the following major threats to validity:
Data leakage Given that large language models (LLMs) are trained on extensive datasets, one potential risk is data leakage. Specifically, the LLM used in ULog might have been trained on open-source log datasets, which could result in memorizing ground-truth templates rather than performing genuine inference. However, experimental results indicate that the performance of merely using LLMs (LILAC w/o ICL) is significantly lower compared to ULog, suggesting a low likelihood of direct memorization. Additionally, ULog utilizes the same LLM, *i.e.*, *gpt-turbo-3.5-0613* model for experiments, aligning with previous work, LILAC.
Randomness Randomness can influence the performance of ULog and other baseline methods. To mitigate this issue, we minimized the randomness of the LLM by setting the temperature to 0, ensuring consistent outputs for the same input text. Additionally, we conducted each experiment five times for every experimental setting and used the average of these results as the final outcome.
Implementation and settings To mitigate the bias of implementation and settings, in our evaluation, we compared our ULog with state-of-the-art approaches within the same evaluation framework. We adopted the implementations from their replication packages and benchmarks, using the parameters and settings (*e.g.*, number of log templates and similarity threshold) optimized by previous work [19, 55]. Moreover, the results of the baseline approaches are consistent with the best results in recent benchmarks.

7 RELATED WORKS

Log parsing is a critical preliminary step for various log analysis tasks [21], including anomaly detection and root cause analysis. Therefore, numerous efforts have been made to achieve accurate and efficient log parsing [7, 9, 33, 34, 36, 37, 41, 42, 44, 51]. These log parsers can be categorized into two types: unsupervised syntax-based and supervised semantic log parsers. Unsupervised syntax-based log parsers leverage predefined rules or heuristics to extract the constant parts of log messages as log templates. For instance, SLCT [43] was the first approach to use token frequencies to determine log templates and parameters. LogMine [12] employs a bottom-up clustering algorithm to segment log messages based on customized similarity measures, and extract log templates for each cluster. Furthermore, Drain [14] utilizes a fixed-depth prefix tree structure to effectively extract commonly occurring templates based on specific heuristics (*i.e.*, prefix tokens and log length). These syntax-based log parsers do not rely on manually labelled examples.

However, their parsing accuracy can significantly decline when log data do not conform to predefined rules.

On the other hand, semantic-based log parsers utilize neural networks [25, 30] or language models [22, 23] to identify log templates and parameters by understanding the semantics of log messages. They require human-labelled log templates to train or tune the models by learning the semantics and patterns in the labelled log data. UniParser [30] is one of the pioneering work in parsing logs with a focus on their semantic meaning. It integrates a BiLSTM-based semantics miner with a joint parser to identify log templates. Additionally, LogPPT [23] proposed identifying log templates and parameters using prompt-based few-shot learning, based on the RoBERTa model. Recently, with the rise of large language models (LLMs), a series of LLM-based log parsers have been developed to achieve more effective log parsing. These LLM-based log parsers leverage fine-tuning [29] or in-context learning [18, 49] to specialize LLMs for log parsing tasks, thereby achieving remarkable performance. However, these semantic-based log parsers heavily rely on labelled data, which limits their ability to generalize to different types of log data or evolving log data [19]. In contrast, our proposed unsupervised LLM-based log parser, ULog, does not require labelled examples, allowing it to generalize to diverse and evolving log data.

8 CONCLUSION

In this work, we propose an LLM-based unsupervised log parser named ULog, which leverages log contrastive units (LCUs) to facilitate effective comparisons by the LLM. To efficiently identify effective LCUs from large-scale log data, ULog employs a hierarchical sharder to divide logs into buckets, thereby reducing sampling overhead and enabling parallel computation. Additionally, ULog incorporates a hybrid LCU selector that jointly measures the commonality and variability of LCUs, which is crucial for prompting LLMs. Furthermore, ULog introduces an improved prompt format to guide LLMs in a zero-shot setting. Experimental results on 14 large-scale public datasets demonstrate that ULog achieves high parsing accuracy, significantly outperforming other unsupervised parsers and being comparable to state-of-the-art parsers that require labelled data. In terms of efficiency, ULog is also comparable to the fastest baseline parser, highlighting its potential for application in real-world systems.

REFERENCES

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

[2] 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).

[3] 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.

[4] Vincent Bushong, Russell Sanders, Jacob Curtis, Mark Du, Tomas Cerny, Karel Frajtak, Miroslav Bures, Pavel Tisnovsky, and Dongwan Shin. 2020. On matching log analysis to source code: A systematic mapping study. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*. 181–187.

[5] 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.

[6] Zhuangbin Chen, Jinyang Liu, Wenwei Gu, Yuxin Su, and Michael R Lyu. 2021. Experience report: Deep learning-based system log analysis for anomaly detection. *arXiv preprint arXiv:2107.05908* (2021).

[7] 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.

[8] 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).

[9] 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.

[10] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A survey on concept drift adaptation. *ACM computing surveys (CSUR)* 46, 4 (2014), 1–37.

[11] 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).

[12] 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.

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

[14] 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.

[15] Shilin He, Xu Zhang, Pinjia He, Yong Xu, Liqun Li, Yu Kang, Minghua Ma, Yining Wei, Yingnong Dang, Saravanakumar Rajmohan, et al. 2022. An empirical study of log analysis at Microsoft. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*. 1465–1476.

[16] 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).

[17] Guochao Jiang, Zepeng Ding, Yuchen Shi, and Deqing Yang. 2024. P-ICL: Point In-Context Learning for Named Entity Recognition with Large Language Models. *arXiv preprint arXiv:2405.04960* (2024).

[18] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R Lyu. 2023. LLMParser: A LLM-based Log Parsing Framework. *arXiv preprint arXiv:2310.01796* (2023).

[19] 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).

[20] 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.

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

[22] 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.

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

[24] Xiaoyun Li, Hongyu Zhang, Van-Hoang Le, and Pengfei Chen. 2023. LogShrink: Effective Log Compression by Leveraging Commonality and Variability of Log

Data. *arXiv preprint arXiv:2309.09479* (2023).

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

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

[27] 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.

[28] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[29] Yilun Liu, Shimin Tao, Weibin Meng, Jingyu Wang, Wenbing Ma, Yanqing Zhao, Yuhang Chen, Hao Yang, Yanfei Jiang, and Xun Chen. 2023. LogPrompt: Prompt Engineering Towards Zero-Shot and Interpretable Log Analysis. *arXiv preprint arXiv:2308.07610* (2023).

[30] 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.

[31] Xinxi Lyu, Sewon Min, Iz Beltagy, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2023. Z-ICL: Zero-Shot In-Context Learning with Pseudo-Demonstrations. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2304–2317.

[32] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2024. LLMParser: An Exploratory Study on Using Large Language Models for Log Parsing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[33] 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.

[34] 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.

[35] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the Role of Demonstrations: What Makes In-Context Learning Work?. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. 11048–11064.

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

[37] 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.

[38] 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).

[39] 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.

[40] Weiyi Shang. 2012. Bridging the divide between software developers and operators using logs. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 1583–1586.

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] Tingyu Xie, Qi Li, Yan Zhang, Zuozhu Liu, and Hongwei Wang. 2023. Self-improving for zero-shot named entity recognition with large language models. *arXiv preprint arXiv:2311.08921* (2023).

[48] Junjielong Xu, Qiuai Fu, Zhouruixing Zhu, Yutong Cheng, Zhijing Li, Yuchi Ma, and Pinjia He. 2023. 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.

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

[50] 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).

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

[52] 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.

[53] 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.

[54] Nengwen Zhao, Honglin Wang, Zeyan Li, Xiao Peng, Gang Wang, Zhu Pan, Yong Wu, Zhen Feng, Xidao Wen, Wenchi Zhang, et al. 2021. 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.

[55] 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.