Adaptive and Efficient Log Parsing as a Cloud Service

Zeyan Li
ByteDance Inc.
Beijing, China
lizeyan.42@bytedance.com

Jie Song
ByteDance Inc.
Seattle, USA
jie.song@bytedance.com

Tieying Zhang*
ByteDance Inc.
San Jose, USA
tieying.zhang@bytedance.com

Tao Yang
Xiongjun Ou
ByteDance Inc.
Shenzhen, China
yangtao.alan@bytedance.com
ouxiongjun@bytedance.com

Yingjie Ye
ByteDance Inc.
Xi'an, China
yeyingjie.bluarry@bytedance.com

Pengfei Duan
Muchen Lin
ByteDance Inc.
Chengdu, China
duanpengfei.1010@bytedance.com
linmuchen@bytedance.com

Jianjun Chen
ByteDance Inc.
San Jose, USA
jianjun.chen@bytedance.com

Abstract

Logs are a critical data source for cloud systems, enabling advanced features like monitoring, alerting, and root cause analysis. However, the massive scale and diverse formats of unstructured logs pose challenges for adaptable, efficient, and accurate parsing methods. This paper introduces *ByteBrain-LogParser*, an innovative log parsing framework designed specifically for cloud environments. *ByteBrain-LogParser* employs a hierarchical clustering algorithm to allow real-time precision adjustments, coupled with optimizations such as positional similarity distance, deduplication, and hash encoding to enhance performance. Experiments on large-scale datasets show that it processes 229,000 logs per second on average, achieving an 840% speedup over the fastest baseline while maintaining accuracy comparable to state-of-the-art methods. Real-world evaluations further validate its efficiency and adaptability, demonstrating its potential as a robust cloud-based log parsing solution.

CCS Concepts

• **Information systems** → **Data mining**.

Keywords

Log parsing, Hierarchical clustering, Cloud service

ACM Reference Format:
Zeyan Li, Jie Song, Tieying Zhang, Tao Yang, Xiongjun Ou, Yingjie Ye, Pengfei Duan, Muchen Lin, and Jianjun Chen. 2025. Adaptive and Efficient Log Parsing as a Cloud Service . In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3722212.3724427

*Corresponding author

SIGMOD-Companion '25, June 22–27, 2025, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3724427

```
print(f"release:lock={lock}, flg={flag}, tag={tag}, name={name}, ws={ws}")
print(f"acquire:lock={lock}, flg={flag}, tag={tag}, name={name}, ws={ws}")
```

Log statements

```
release:lock=2337, flg=0x0, tag="View Lock", name=systemui, ws=null
release:lock=187, flg=0x0, tag="*launch*", name=android, ws=WS{10113}
release:lock=62, flg=0x0, tag="WindowManager", name=android, ws=WS{1013}
acquire lock=23, flg=0x1, tag="View Lock", name=systemui, ws=null
acquire lock=1661, flg=0x1, tag="RILJ_ACK_WL", name=phone, ws=null
```

Log parsing

```
Template 1: release lock * flg * tag * name * ws *
Template 2: acquire lock * flg * tag * name * ws *
```

Figure 1: An example of log parsing

1 Introduction

System logs provide a rich source of information about the behavior and performance of distributed systems, capturing runtime events, execution flows, and operational states. These logs play a pivotal role in automated analysis tasks, including anomaly detection, fault diagnosis, and performance monitoring [6, 17, 21, 32, 35]. However, the unstructured nature and diverse formats of logs, combined with the vast scale of modern cloud systems, make extracting actionable insights challenging [7, 30, 34]. Therefore, most automated log analysis relies on log parsing to automatically extract structured log templates and variables from unstructured log records to address these challenges. Fig. 1 illustrates how parsing transforms code-generated raw log records into structured templates.

Cloud services like AWS *CloudWatch* and Azure *Monitor* provide tenants with foundational log management capabilities, including ingestion, indexing, querying, and basic analytical features. These platforms process millions of logs per second from diverse application components, aggregating them into massive, heterogeneous streams containing thousands of distinct log templates. The complexity stems from the diverse nature of modern cloud applications, where each component may generate numerous types of logs for different scenarios and states. While these platforms provide rule-based grouping, visualization, and alerting, they rely on manual configurations for log parsing, which becomes impractical as template numbers grow. Recognizing this gap, *Volcano Engine*, the cloud computing services from ByteDance, extends these capabilities by

introducing automated, out-of-the-box log parsing, which transforms raw logs into structured formats and enables users to query logs more intuitively. Based on these parsing results, we further provide multiple advanced analytics capabilities, including log anomaly detection (identifying abnormal changes in template quantities and newly emerged templates), template distribution comparison across different time periods, and automatic matching against a library of known failure scenarios. These out-of-the-box features allow users to quickly identify system issues, understand behavioral changes, and diagnose problems.

Delivering log parsing as a cloud service in large scale is fraught with challenges. These include:

(1) Adaptability: Applications require varying levels of parsing precision, even for logs within the same stream, necessitating a system that adjusts dynamically to user needs. For example, logs generated by *print(register callback for {}.format(email))* need be parsed as both *register callback for ** and *register callback for None* during debugging to discover unexpected *null*, while merging these into one template can be better in other scenarios. This diversity demands real-time adjustable parsing mechanisms.
(2) Compute Efficiency: The massive scale of cloud logs necessitates efficient algorithms for both offline training and online matching. Delays in log parsing can impede query response time and increase compute resource costs.
(3) Storage Efficiency: Log parsing adds additional storage costs for cloud tenants. To keep the service cost-effective, resource usage must be minimized without compromising performance.
(4) Parsing Accuracy: handling diverse and unpredictable log patterns with high precision, even in the absence of prior knowledge.

Existing techniques inadequately address these challenges. Heuristic rule-based methods [10, 12, 16] struggle to adapt to diverse log patterns. Frequent pattern mining methods [11, 25, 29] are sensitive to parameter tuning and preprocessing, which limits their robustness. Existing log clustering methods [22, 30] can fail to generate accurate templates and incur significant overhead. Deep learning approaches [13, 28] achieve high accuracy but require substantial labeled data and compute resources. LLM-based methods [14] offer flexibility and adaptability but suffer from high inference costs and latency. Additionally, most existing works struggle to compute log parsing results at varying precision levels in real-time.

To tackle the challenges of log parsing as a cloud service, we introduce *ByteBrain-LogParser* (referred to as *ByteBrain*), a comprehensive framework tailored for cloud service environments. *ByteBrain* is designed to balance adaptability and efficiency, addressing the diverse requirements of cloud tenants. The framework operates in two phases. In the offline training phase, logs are periodically collected and hierarchically clustered into a tree structure, where each node represents a log template. Deeper nodes correspond to more precise templates, enabling granular control over parsing precision. This phase leverages innovative techniques such as positional similarity distance and saturation scoring, which optimize the clustering process to achieve a balance between computational efficiency and parsing precision.

In the online matching phase, *ByteBrain* processes incoming logs in real time by matching them against pre-trained templates. The system allows users to adjust parsing precision dynamically, based on their operational needs, by specifying thresholds at query time. This capability enables seamless transition between coarse-grained and fine-grained parsing without requiring the reprocessing of logs. By efficiently managing the interplay between accuracy and resource utilization, *ByteBrain* ensures cost-effective operation for high-throughput environments.

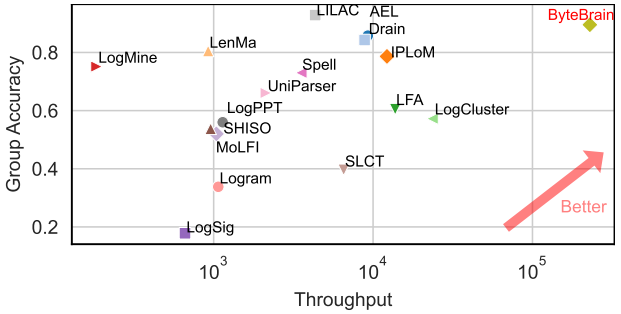

Figure 2: Our method meets the goal of high throughput and near-SOTA accuracy

Several key techniques enhance *ByteBrain*'s efficiency. Positional similarity distance quantifies log structural similarity through key variable positions, improving clustering accuracy. Hash encoding enables efficient storage and fast online parsing. Variable saturation metrics guide hierarchical clustering to optimize template generation and avoid redundant refinements. These techniques collectively enable *ByteBrain* to process diverse, large-scale log streams efficiently.

We evaluate *ByteBrain* on the widely-used LogHub and LogHub-2.0 datasets (see Section 5.1.1). It achieves an average accuracy of 0.98 and 0.90 on LogHub and LogHub-2.0 respectively, closely matching the SOTA (state-of-the-art) method accuracy of 0.99 and 0.93. Its key strength lies in throughput, processing 229,000 logs per second, which is 840.68% faster than the fastest baseline and outperforms others by 1-3 orders of magnitude. As shown in Fig. 2, this combination of near-SOTA accuracy and excellent efficiency positions *ByteBrain* as a highly competitive solution for log parsing as a cloud service. Additionaly, our ablation study (Section 5.4) validates each proposed technique. Industrial evaluations (Section 6) demonstrate its practical advantages in production. *ByteBrain* has been deployed in production on *Volcano Engine*'s *Torch Log Service* (TLS), with real-world evaluations confirming its performance and reliability in cloud computing environments.

In summary, the key contributions of this paper are as follows:

(1) An adaptive and efficient log parsing framework for cloud environments: Our system supports real-time, large-scale log parsing and allows users to adaptively adjust parsing precision to meet various operational requirements with low compute and storage overhead.
(2) An efficient hierarchical clustering-based log parsing algorithm: We introduce positional similarity distance, and variable saturation to enhance log template extraction while minimizing computational overhead and storage costs.
(3) Comprehensive evaluation on large-scale real-world datasets: Extensive experiments demonstrate the near-SOTA accuracy and unprecedented efficiency of our method.

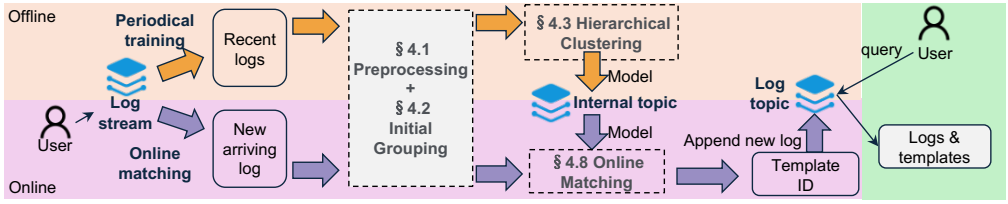

Figure 3: System design of *ByteBrain-LogParser*

2 Related Work

Log parsing remains critical for system management and analysis. Existing techniques are broadly categorized as syntax-based or semantic-based approaches. Each category offers distinct trade-offs in cloud environments where real-time processing, efficiency, and adaptability are paramount. Traditional industrial systems (LogStash [3], Splunk [4], CloudWatch [1], DataDog [2]) that rely on user-defined patterns fall outside our focus on automated parsing methods.

Syntax-based log parsers have been widely adopted in log parsing due to their simplicity and compute efficiency. These parsers primarily rely on predefined rules, heuristics, or patterns to extract structured templates from unstructured log data. Classic parsers such as *SLCT* [16], *Logram* [9], and *Drain* [12] employ various heuristic rules to parse log messages. For instance, Drain constructs a fixed-depth parse tree for message classification, while Logram leverages n-gram dictionaries for variable token identification. Another category of syntax-based methods utilizes frequent pattern mining or clustering techniques to identify recurring log structures. These methods exhibit several limitations: frequent pattern mining-based approaches struggle to identify low-frequency log patterns, while clustering-based methods often suffer from high computational overhead, parameter sensitivity, and suboptimal accuracy. For example, LogMine's [11] iterative clustering and merging process incurs substantial computational costs, while LogSig [27] requires precise specification of log category numbers. LogCluster's [19] word-frequency based clustering approach fails to differentiate between semantically distinct messages that share common word distributions. Notably, although SPINE [30] employs hierarchical clustering to automatically determine the number of clusters and emphasizes scalability, its log encoding and clustering algorithms impede template generation, limiting its viability as a cloud service.

Semantic-based log parsers have recently emerged, leveraging machine learning and deep learning models to capture deeper dependencies and semantic relationships within log data. Approaches like *UniParser* [21], *LogPPT* [18] and *LogStamp* [28] represent this category, utilizing custom deep learning models or pretrained language models such as *RoBERTa* [20] to learn semantic patterns from logs. These methods often provide higher parsing accuracy, particularly for complex and unstructured logs, as they do not rely on rigid token-based rules. However, semantic-based methods typically demand significant labeled data and compute resources, making them impractical for large-scale or real-time applications. Additionally, their inference costs can result in latency issues, limiting their cost efficiency in cloud environments.

The rise of Large Language Models (LLMs) offers a promising new direction for log parsing. LLMs, pre-trained on vast amounts of text data, have demonstrated the ability to understand complex language patterns, making them suitable for parsing log messages without the need for manually crafted rules or extensive labeled data. Early work, such as *DivLog* [31] and *LILAC* [14], explores the use of LLMs for log parsing. LILAC, in particular, introduces an adaptive parsing cache to mitigate the inefficiency and inconsistency issues commonly associated with LLMs, ensuring both high parsing accuracy and better compute efficiency. This represents a significant shift from traditional methods, as it leverages the in-context learning (ICL) capabilities of LLMs to dynamically adapt to different log formats without requiring large-scale retraining. However, these methods still require substantial resources for both pertraining and inference, making them less feasible for large-scale log parsing in cloud environments.

3 System Design

ByteBrain is designed as an adaptive log parsing system tailored for high-throughput cloud environments. The system adopts a two-phase approach, including offline training and online matching, to achieve efficiency while maintaining adaptability to diverse log patterns. Fig. 3 illustrates the system architecture.

Offline Training. A log topic, representing a single log stream, serves as the fundamental unit of our log service, where records are indexed, stored, and made available for analysis. Logs within each topic are processed in two phases. During offline training, logs are collected, preprocessed, initially grouped and hierarchically clustered into a tree structure, where each node represents a log template. This structure enables *ByteBrain* to dynamically adjust parsing precision by traversing the tree based on user-defined thresholds. The saturation score (see Section 4.5), which strictly increases with tree depth, quantifies template precision. Each node stores its metadata including template text, saturation score and parent-child relationships in an internal topic. This enables efficient navigation across precision levels while reducing reliance on external databases. Training is triggered upon reaching either a volume threshold or a time interval after last execution. Templates are unavailable for logs before first training completes. However, this limitation is negligible as we configured initial training to finish within 5 minutes, which is inconsequential compared to the typical lifecycle (months to years) of log topics. For exceptionally large log volumes, random sampling prevents out-of-memory (OOM) issues. The newly trained model is merged with the previous one. Templates with similarity scores above a given threshold are merged; otherwise, they remain separate child nodes.

Online Matching. The online phase processes incoming logs through preprocessing and initial grouping before matching them against the trained model. To optimize throughput and latency,

the system distributes matching tasks across multiple processing queues, leveraging the independent nature of template matching. Latency matters because template IDs must be computed along with other traditional text indices before logs can be written to the append-only log topic storage. For rare log messages that do not appear in the training data, they may fail to match any node in the clustering trees during the online matching phase. In such cases, we treat the log record itself as a temporary template and insert it into the clustering tree as an individual node. These unmatched logs are subsequently considered during the next training cycle, allowing the system to adaptively learn new patterns and update the clustering tree accordingly.

Query. Users can dynamically control template precision during queries by specifying a threshold. The system efficiently navigates the clustering tree, starting from the retrieved template IDs and traversing upward through ancestor nodes until identifying the coarsest templates that meet the specified threshold. This approach enables real-time precision adjustment without log reprocessing or redundant template storage, offering adaptive log parsing with minimal computation and storage overhead.

Parallel. The system leverages parallelization across all phases. During training, preprocessing tasks (tokenization, variable replacement, and hash encoding) and hierarchical clustering operate concurrently. Moreover, hierarchical clustering can be performed concurrently for each group obtained from initial grouping. The online phase parallelizes template matching across all logs. Query processing benefits from parallel execution of both precise template queries and ancestor node traversal. In production, we optimize resource utilization by limiting parallelization to 1-5 cores, based on topic scale requirements.

4 Algorithm

In this section, we introduce our log parsing algorithm. It starts by transforming unstructured logs into numerical vectors via tokenization and deduplication, followed by initial grouping based on lengths and prefixes to allow parallel processing. We then apply hierarchical clustering to each group, with each iteration generating nodes in a tree where deeper nodes represent more precise templates. We use saturation score to evaluate nodes and determine whether to terminate further clustering. In Section 4.8, we introduce the online matching algorithm.

4.1 Preprocessing

Preprocessing aims to transform log texts into numerical vectors. This is an important step as it bridges the gap between textual data and mathematical algorithms, which thus enables efficient computation. Key preprocessing steps include tokenization, common variable replacement, deduplication, and hash encoding.

4.1.1 Tokenization. Tokenization refers to the process of dividing each log record into a sequence of tokens. By default, we use the following regular expression to segment each log record.

```
(?::|//)|(?:(?:[\s\'\";=()\[\]{}?@&<>:\n\t\r,])|(?:[\.](\s+|$
    ))|(?:\\[\"\']))+
```

Listing 1: Python regular expression for tokenization

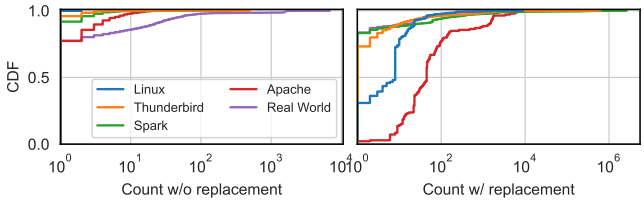

Figure 4: High log duplication with increased redundancy after variable replacement

This regular expression comprises four key components: `://` identifies URL protocol separators; `\s\'\";=()\[\]{}?@&<>:\n\t\r,` captures common delimiters including whitespace, quotes, and punctuation marks; `[\.](\s+|$)` targets sentence-ending periods while preserving those in numerical values; and `\\[\"\']` matches escaped quotation marks frequently used in log data.

We chose regular expressions for tokenization because of their efficiency, simplicity, and customizability. While effective tokenization is crucial for successful log parsing, regular expressions can only segment logs based on common delimiters and cannot account for semantic context. For example, when processing domain names, whether to use periods as delimiters depends on the specific analysis objectives. Nevertheless, regular expressions are fast and allow users to easily define custom tokenization rules for each topic. To maintain efficiency, we prohibit the use of high-complexity regex features in user-defined expressions, such as look around, which increases complexity from $O(n)$ to $O(2^n)$ in worst cases.

4.1.2 Common Variable Replacement. While we focus on automatic log parsing without requiring manual rules, we allow users to optionally specify regex patterns for obvious variables to optimize performance. These user-defined patterns typically target common, domain-specific variables that appear consistently across logs. Early replacement of these known variables significantly reduces the complexity for subsequent automatic parsing. For each topic, we provide default patterns for common variables, including timestamps, IP addresses, MD5 hashes, UUIDs and so on, while users can add domain-specific rules to further enhance efficiency.

4.1.3 Deduplication. Log data frequently contains a substantial number of duplicate records, a phenomenon that becomes more pronounced after replacing common variables. This redundancy not only increases storage overhead but also introduces inefficiencies in processing and analysis. For instance, in Fig. 4, we show the distribution of unique log counts across the LogHub 2.0 datasets (see Section 5.1.1). The prevalence of repeated log patterns underscores the opportunity for optimization through deduplication. In this context, deduplication involves identifying and collapsing duplicate log entries while maintaining a count of the occurrences of each unique log statement. This approach significantly improves compute efficiency by reducing redundant data.

4.1.4 Hash Encoding. For compute efficiency, tokens will be encoded into numerical vectors. A typical approach is bag-of-words encoding [30], which enables Euclidean distance of the encodings are computed and fed into K-means clustering to build log clusters. However, this encoding method disregards the order of tokens and

cannot directly generate template texts from the clusters. Alternatively, ordinal encoding, which assigns a unique numerical ID to each distinct token, has a major drawback: it requires storing a mapping between every token and its corresponding numerical ID. Given the potentially large number of distinct tokens in logs, this results in substantial storage overhead, significantly increasing the cost of the log parsing service (see Section 5.4.4).

To address these challenges, we propose to use hash encoding, which leverages a deterministic hash function to map each token to a 64-bit integer. Using the same hash function for both offline clustering and online matching, we eliminate the need to store token-to-ID mappings. Moreover, unlike ordinal encoding, hash encoding supports parallel processing of logs, as the hash function can independently process each token, thereby improving scalability.

The probability of hash collisions (i.e., two distinct tokens being mapped to the same hash value) is extremely low and can be considered negligible in practice. The collision probability can be approximated using the birthday problem formula. For a 64-bit hash function, the probability p of at least one collision after hashing n distinct tokens is given by:

$$p = 1 - \prod_{k=1}^{n-1}(1 - \frac{k}{2^{64}}) \approx 1 - \prod_{k=1}^{n-1} \underbrace{\exp(-\frac{k}{2^{64}})}_{\ln(1-x)\approx x \text{ when } x \text{ is small}} \quad (1)$$
$$= 1 - \exp(\frac{\sum_{k=1}^{n-1} k}{2^{64}} = 1 - \exp(\frac{n \cdot (n-1)}{2 \cdot 2^{64}})$$

For example, with 10 million distinct tokens, the collision probability is only 0.000271%, which is negligible. Considering that fields like timestamps and UUIDs are directly replaced with wildcards using regular expressions, the number of distinct tokens processed during the encoding stage is significantly smaller than the number of words in the original log text.

4.2 Initial grouping

Initial grouping organizes logs into distinct groups based on simple rules to ensure that logs unlikely to belong to the same template are separated early on. This allows for more efficient and parallel clustering in later stages.

We apply the following initial grouping strategies:

(1) Length: Logs with different token counts are assumed to belong to different templates.
(2) Prefix: Logs are grouped by comparing the first k tokens (configured by users and 0 by default), separating logs with differing prefixes into different groups.

4.3 Hierarchical Clustering

Hierarchical clustering is a core component of *ByteBrain*. Each initial group serves as the root node of a clustering tree, where the logs are iteratively partitioned into sub-nodes. At each iteration, the current node is divided into multiple subnodes based on a clustering algorithm tailored for log data. The clustering process ensures that logs within the same subnode exhibit higher structural similarity (higher saturation as described below) compared to those in the parent node. The precision of the log templates increases as nodes are partitioned further down the tree. The tree structure naturally organizes logs, making it easier to identify relationships between templates at different levels of precision. Fig. 5 below presents clustering tree examples.

The termination of the clustering process for a given node depends on the saturation score, which measures how well the logs in a node have been resolved into either constants or variables. Nodes with a high saturation score are considered sufficiently refined and do not need further split. The detailed algorithm for a single clustering process and the calculation of the saturation score are discussed in Section 4.4 and Section 4.5, respectively.

4.4 Single Clustering Process

In the single clustering process, our goal is to iteratively group logs in a manner that ensures the saturation score improves in every cluster. This process is inspired by K-Means Clustering and incorporates several modifications to better accommodate the unique characteristics of log data.

The process begins by selecting two logs as the initial cluster centers. Following the principles of K-Means++ [5], the first log is chosen randomly, while the second is selected as the log farthest from the first, based on the distance metric described below. Each of these logs forms the initial core of a cluster, with just one log in each cluster at the start. Such a strategy prevents similar logs from being incorrectly assigned to different clusters.

To calculate the distance $d(L, C)$ between a log L and a cluster C, we propose a positional similarity distance metric instead of the conventional Euclidean distance. This choice is driven by the nature of our hash encoding scheme, where token values act as identifiers without meaningful numerical relationships. Instead, our distance calculation incorporates two key factors:

(1) *Token frequency at each position:* For each token in the log, we evaluate its frequency of occurrence at the corresponding position across all logs in the cluster. A higher frequency indicates that the token is more representative of that position in the cluster. We denote the frequency of the token at position i in log L within cluster C as $f_i(L, C)$.
(2) *Position importance:* Positions with greater variability are more likely to contain variables and are assigned lower importance. We introduce a weight $w_i = \frac{1}{n_i - 1}$ for each position i, where n_i represents the distinct token count at position i within cluster C.

The positional similarity distance is then defined as:

$$d(L, C) = \frac{\sum_{i=1}^{m} w_i \cdot f_i(L, C)}{\sum_{i=1}^{m} w_i} \quad (2)$$

After computing the distances between each log and the clusters, each log is assigned to the cluster with the smallest distance (i.e., the highest positional similarity). This ensures that logs with similar structures are grouped together effectively.

After the initial assignment of logs to clusters, we iteratively refine the clustering results. In each iteration, as clusters contain new sets of logs, we recalculate the distances between each log and all clusters, then reassign each log to its nearest cluster. During this process, we monitor the saturation score of each cluster compared to its parent node (the set of all logs). If a cluster's saturation score shows no improvement, indicating that no additional token positions have been identified as either constants or variables within

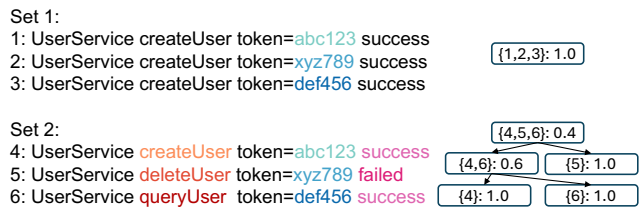

Figure 5: Illustration example of two log sets and the corresponding clustering trees (*log IDs: saturation* for each node)

this cluster, we introduce a new cluster. The centroid of this new cluster is initialized using the log that exhibits maximum distance from all existing cluster centroids. This strategic expansion of clusters is naturally bounded by the finite number of token positions in the logs. Once all possible token positions are classified (reaching a saturation score of 1), further splitting would not yield meaningful improvements. This approach ensures both adaptability to complex log patterns and compute efficiency by avoiding the creation of unnecessary hierarchical levels.

4.5 Calculation of Saturation

Saturation is used to evaluate how well the positions in a group of logs have been resolved into constants or variables, and controls the termination of hierarchical clustering. Unlike existing works [30], our saturation calculation considers both confirmed constants and likely variables. It permits faster clustering termination while avoiding unnecessary splits, which improves efficiency and accuracy.

If all logs in a group share the same token at a specific position, that position is definitively considered a constant. For example, in *Set 1* of Fig. 5, all positions except the token value are identical across logs, meaning these positions are constants. Meanwhile, the high variability of the token position strongly suggests that it is a variable, as further splitting based on token values would not generate meaningful templates. However, for *Set 2*, though the token position still has different values in every log, there exists value variability across other positions (e.g., the action and status fields). This broader variability indicates that the token might not always be a standalone variable but could be structurally correlated with other fields. In such cases, maintaining separate template for each log preserves important structural patterns.

Therefore, to compute the saturation score, we account for both constant and variable positions in three steps.

(1) *Proportion of constants* Let m be the total number of positions in the logs, and m_c the number of positions where all tokens are identical across logs. The proportion of constants is computed as $f_c = \frac{m_c}{m}$, representing how many positions are fully resolved.
(2) *Variability of unresolved positions* For each unresolved position, let n be the total number of logs and n_u the number of distinct tokens at that position. The scale factor for variability at position i is $f_v^{(i)} = \frac{\log(n_u)-1}{\log n}$, which grows with the number of distinct tokens. The minimum scale value across all unresolved positions is selected as the overall variability factor f_v, ensuring that positions with the highest variability dominate.
(3) *Confidence adjustment* To account for the influence of unresolved positions, we introduce a confidence factor $p_c = \frac{1}{2^{m-m_c-1}}$, which decreases as fewer positions are resolved. This factor ensures that unresolved positions contribute less to the final score when many constants are already identified.

Finally, the saturation score $s(C)$ is computed as:

$$s(C) = (f_v \cdot p_c + (1 - p_c)) \cdot f_c \tag{3}$$

This formula balances the proportion of constants with variability in unresolved positions, giving higher scores to groups that are well-resolved while penalizing groups with high variability.

Based on the saturation definition above, in Fig. 5, for Set 1, the saturation of all three logs is already 1 and thus, we avoid further meaningless splits. In Set 2, we gradually separate the three logs into different clusters, where the saturation of each new node increases compared to its parent until reaching 1 at the leaf nodes. Compared to prior methods, our approach provides finer control over the clustering process by dynamically considering both resolved constants and unresolved variability. This refinement leads to more accurate and meaningful log templates and enhances efficiency by avoiding meaningless splits.

4.6 Balanced grouping

When calculating the distance between a log and multiple clusters, it is common for the log to have the same distance to multiple clusters. In such cases, to ensure even cluster distribution, we randomly assign the log to one of these clusters with equal probability, rather than deterministically assigning it to the first cluster.

Balancing the distribution of logs across clusters helps minimize the depth of the resulting clustering tree. When users specify a saturation threshold at query time, the system identifies the coarsest template that satisfies the threshold by traversing the ancestor nodes of the most precise template (pre-computed during online matching). A shallower tree means fewer nodes need to be traversed, which improves query efficiency and reduces latency. It could also reduces the total number of iterations required by the clustering algorithm, thereby improving compute efficiency at training time.

4.7 Early Stop

In certain cases, the algorithm can immediately determine that each distinct log should form a separate cluster without proceeding with the full clustering process. Ending early in these situations could reduce the computational overhead. Early stop applies in the following scenarios:

(1) *Few logs:* If the number of logs is less than or equal to 2, each log naturally forms a separate cluster.
(2) *Single unresolved position:* If only one position remains unresolved (i.e., it cannot be classified as a constant or variable), further clustering is unnecessary since splitting based on a single position will not increase saturation.
(3) *Completely distinct unresolved positions:* If unresolved positions contain entirely different tokens in each log, these logs are inherently dissimilar and should belong to separate clusters.

4.8 Online Matching

In the online matching process, logs are directly matched to template texts instead of traversing the clustering tree by recalculating positional similarity distances. This approach significantly reduces

Table 1: Loghub and Loghub-2.0 dataset statistics

Dataset Name	Loghub			Loghub-2.0		
	#Logs	Size	#Templates	#Logs	Size	#Templates
HealthApp	2000	183.06 KB	75	212394	19.53 MB	156
OpenStack	2000	581.17 KB	43	207632	58.56 MB	48
OpenSSH	2000	219.94 KB	27	638947	67.27 MB	38
Proxifier	2000	231.41 KB	8	21320	2.40 MB	11
HPC	2000	147.63 KB	46	429988	31.10 MB	74
Zookeeper	2000	273.33 KB	50	74273	9.85 MB	89
Mac	2000	311.93 KB	341	100314	14.72 MB	626
Hadoop	2000	375.93 KB	114	179993	30.41 MB	236
Linux	2000	211.41 KB	118	23921	2.04 MB	338
Android	2000	272.54 KB	166	-	-	-
HDFS	2000	281.10 KB	14	11167740	1.46 GB	46
BGL	2000	309.72 KB	120	4631261	686.12 MB	320
Windows	2000	278.74 KB	50	-	-	-
Apache	2000	167.23 KB	6	51978	4.75 MB	29
Thunderbird	2000	317.57 KB	149	16601745	2.34 GB	1241
Spark	2000	191.67 KB	36	16075117	1.52 GB	236

storage requirements compared to recalculating distances at each tree node, which would require storing detailed token-level information (e.g., token frequencies or variability metrics) for every node. By storing only the final template texts, our method avoids this overhead, making it more efficient for cloud-based log parsing.

When a new log arrives, it is sequentially matched against all templates in descending order of saturation score, stopping as soon as a match is found. The matching process is position-based: the token in the log must either match the token in the template exactly or match a wildcard, which indicates a variable. Although this method does not guarantee that a log will map to the exact node it would have been assigned to during clustering, it achieves high accuracy. This is because the templates, generated through clustering, already capture the key structural patterns of logs, and the saturation score prioritizes templates that are both precise and general. It is also demonstrated by our experiment results in Section 5.4. In summary, this approach ensures accurate and efficient matching without recalculating distances or traversing the tree.

5 Experiment

In this section, we comprehensively evaluate the effectiveness and efficiency of *ByteBrain* on widely-used public datasets.

5.1 Experiment Setup

5.1.1 Dataset. We evaluate our method on two widely-used public datasets: LogHub [36] and LogHub-2.0 [15]. The original LogHub dataset [36], consisting of 16 diverse datasets from various sources such as distributed systems, operating systems, and software applications, has been widely adopted in numerous log parsing studies [8, 9, 21, 30, 33, 34]. However, its relatively small size (2,000 labeled logs per dataset) limits its validity. To address this, LogHub-2.0 extends LogHub with larger-scale labeled logs, some exceeding 50 million messages, and has gained traction as a benchmark for scalable log parsing [14]. The combination of these datasets allows us to evaluate our method comprehensively, assessing both parsing accuracy on diverse sources and efficiency on large-scale logs.

5.1.2 Baselines. We compare against a comprehensive set of baseline methods, covering diverse log parsing techniques. Syntax-based baselines include clustering-based approaches, such as IPLoM [22], LogCluster [19], and LenMa [26]; frequent pattern mining approaches, such as SLCT [29], LFA [25], LogMine [11], and SHISO [24]; heuristic rule-based approaches, such as AEL [16], Drain [12] and Spell [10]; search-based approach, including Logsig [27] and MoLFI [23]. Deep learning-based baselines include UniParser [21], Logram [9] and LogPPT [18]. LLM-based methods are represented by LILAC [7]. These baselines are implemented with either the open-source Logparser toolkit [37] or their official open-source code.

5.1.3 Evaluation Metrics. We adopt the following standard metrics to evaluate the performance of our method, consistent with prior work in log parsing [7, 12, 21, 30, 33].

- Grouping Accuracy (GA): The ratio of correctly grouped logs to total logs. A log is correctly grouped only when placed with all other logs sharing its ground-truth template. This strict metric prevents accuracy inflation from simple, frequent patterns.
- Throughput (logs/sec): Logs processed per second, calculated as the total log count divided by the combined time for model training and log matching.

5.2 Effectiveness Comparison

We rigorously evaluate the effectiveness of our method, *ByteBrain*, by comparing it with state-of-the-art log parsing approaches across two diverse datasets: 16 small labeled datasets from LogHub and 14 large-scale datasets from LogHub-2.0. The results, presented in Table 2 and Table 3, demonstrate *ByteBrain*'s superior performance across a wide spectrum of log types and volumes.

On the LogHub dataset, *ByteBrain* achieved an impressive average grouping accuracy of 0.98, surpassing most existing methods. It consistently ranked among the top performers across individual datasets, showcasing its versatility in handling various log patterns effectively. *ByteBrain*'s performance was particularly notable on complex datasets (e.g., Linux and Mac), underscoring its robustness and adaptability to diverse log structures.

The advantages of *ByteBrain* became even more pronounced when evaluated on the large-scale LogHub-2.0 datasets. With an average grouping accuracy of 0.90, *ByteBrain* significantly outperformed most baselines, many of which experienced substantial performance degradation when scaling to larger log volumes. Notably, *ByteBrain* maintained high accuracy across different log types, from system logs (e.g., HDFS, Spark) to application logs (e.g., Thunderbird, HealthApp), demonstrating its scalability and consistency in handling massive and diverse log data. While LILAC showed slightly higher accuracy, its poor efficiency (as detailed in Section 5.2) limits its practical applicability in real-world scenarios, especially for large-scale, real-time log parsing tasks.

It's worth noting that *ByteBrain*'s performance remained consistently high across datasets of varying sizes and complexities, despite not being specifically optimized for any particular log type or domain. This domain-agnostic effectiveness demonstrates *ByteBrain*'s inherent ability to handle diverse and unpredictable log patterns, a crucial attribute for a cloud-based log parsing service that must process logs from various applications and systems. Moreover, *ByteBrain* successfully completed parsing tasks on all datasets, unlike some competing methods that failed to finish on certain large-scale logs, further highlighting its robustness and reliability. In some datasets, such as Mac, our method shows slightly lower

Table 2: Group Accuracy Comparison on LogHub. The highest group accuracy for each dataset is highlighted in bold, and the second highest is underlined.

Method	Android	Apache	BGL	HDFS	HPC	Hadoop	HealthApp	Linux	Mac	OpenSSH	OpenStack	Proxifier	Spark	Thunderbird	Windows	Zookeeper	Average
AEL	0.68	**1.00**	0.76	**1.00**	0.90	0.54	0.57	0.67	0.76	0.54	0.76	0.52	0.91	0.94	0.69	0.92	0.76±0.17
Drain	0.91	**1.00**	0.96	**1.00**	0.89	0.95	0.78	0.69	0.79	0.79	0.73	0.53	0.92	0.96	**1.00**	0.97	0.87±0.14
IPLoM	0.71	**1.00**	0.94	**1.00**	0.82	0.95	0.82	0.67	0.67	0.80	0.87	0.52	0.92	0.66	0.57	0.96	0.80±0.15
LenMa	0.88	**1.00**	0.69	**1.00**	0.83	0.89	0.17	0.70	0.70	0.93	0.74	0.51	0.88	0.94	0.57	0.84	0.77±0.21
LFA	0.62	**1.00**	0.85	0.82	0.55	0.89	0.90	0.28	0.60	0.50	0.20	0.03	0.99	0.65	0.59	0.84	0.64±0.29
LogCluster	0.80	0.71	0.83	0.79	0.53	0.55	0.56	0.63	0.60	0.42	0.70	0.48	0.80	0.60	0.71	0.73	0.65±0.12
LogMine	0.50	**1.00**	0.72	0.78	0.69	0.85	0.87	0.61	0.88	0.43	0.74	0.52	0.58	0.92	0.99	0.69	0.74±0.18
Logram	0.85	0.70	0.74	0.98	0.96	0.97	0.97	0.46	0.67	0.85	0.55	0.95	0.90	0.76	0.96	0.96	0.83±0.16
LogSig	0.54	0.73	0.23	0.38	0.09	0.51	0.63	0.11	0.52	0.44	0.84	0.49	0.54	0.76	0.68	0.78	0.52±0.23
MoLFI	0.63	**1.00**	0.95	0.51	0.46	**1.00**	0.72	0.28	0.64	0.54	0.21	0.01	0.42	0.66	0.41	0.84	0.58±0.28
SHISO	0.58	**1.00**	0.71	0.33	0.40	**1.00**	0.87	0.67	0.59	0.62	0.72	0.52	0.91	0.58	0.70	0.66	0.68±0.19
SLCT	0.88	0.73	0.57	0.84	0.33	0.55	0.42	0.30	0.56	0.52	0.87	0.52	0.69	0.88	0.70	0.73	0.63±0.19
Spell	0.92	**1.00**	0.79	**1.00**	0.65	0.78	0.64	0.61	0.76	0.55	0.76	0.53	0.91	0.84	0.99	0.96	0.79±0.16
UniParser	**0.97**	**1.00**	**1.00**	**1.00**	0.97	**1.00**	**1.00**	0.88	**1.00**	**1.00**	**1.00**	0.98	**1.00**	**0.99**	**1.00**	**1.00**	**0.99±0.03**
LogPPT	0.89	**1.00**	0.95	**1.00**	0.94	0.99	**1.00**	0.93	0.78	0.63	0.99	**1.00**	**1.00**	0.68	0.99	0.99	0.92±0.12
LILAC	0.93	**1.00**	0.98	**1.00**	0.97	0.99	**1.00**	0.75	0.82	0.56	**1.00**	**1.00**	**1.00**	0.98	0.99	0.99	0.94±0.12
ByteBrain	0.94	**1.00**	0.95	0.98	**1.00**	**1.00**	0.96	**0.98**	0.90	0.99	**1.00**	0.99	**1.00**	0.96	**1.00**	0.97	0.98±0.03

Table 3: Group Accuracy Comparison on LogHub-2.0. The highest group accuracy for each dataset is highlighted in bold, and the second highest is underlined. Missing data indicates the corresponding method failing to finish.

Method	Apache	BGL	HDFS	HPC	Hadoop	HealthApp	Linux	Mac	OpenSSH	OpenStack	Proxifier	Spark	Thunderbird	Zookeeper	Average
AEL	**1.00**	**0.92**	**1.00**	0.75	0.82	0.73	0.92	0.80	0.71	0.74	0.97	\	0.79	**1.00**	0.86±0.11
Drain	**1.00**	**0.92**	**1.00**	0.79	**0.92**	0.86	0.69	0.76	0.71	0.75	0.69	0.89	**0.83**	0.99	0.84±0.11
IPLoM	0.99	0.90	0.96	0.79	**0.92**	0.98	0.81	0.64	0.41	0.38	0.80	0.72	0.72	0.99	0.79±0.20
LenMa	0.99	\	**1.00**	0.79	0.80	\	0.81	0.70	**0.75**	0.85	0.50	\	\	0.86	0.81±0.14
LFA	0.81	0.73	0.75	0.73	0.83	0.80	0.23	0.59	0.16	0.67	0.35	0.60	0.38	0.84	0.61±0.23
LogCluster	0.55	0.76	0.56	0.73	0.48	0.73	0.60	0.46	0.22	0.69	0.66	0.41	0.42	0.74	0.57±0.16
LogMine	**1.00**	0.64	\	\	0.83	\	0.74	0.85	\	\	0.50	\	\	0.70	0.75±0.16
Logram	0.30	\	\	0.64	0.19	0.23	0.13	0.36	0.22	0.53	0.03	\	\	0.75	0.34±0.23
LogSig	0.11	\	0.37	0.00	0.00	0.00	0.14	0.00	0.47	0.38	0.49	\	\	0.00	0.18±0.21
MoLFI	0.59	\	**1.00**	0.66	0.65	0.55	0.19	0.53	0.45	0.27	0.00	\	\	0.83	0.52±0.29
SHISO	0.57	0.60	**1.00**	0.08	0.72	0.08	0.07	0.61	0.40	0.81	0.69	\	\	0.82	0.54±0.31
SLCT	0.42	\	0.41	0.64	0.23	0.11	0.08	0.42	0.28	**1.00**	0.02	\	\	0.75	0.40±0.30
Spell	**1.00**	0.68	0.96	\	0.45	0.65	0.62	0.76	0.62	0.78	0.52	\	\	0.99	0.73±0.19
UniParser	0.29	0.55	**1.00**	0.79	0.72	0.45	0.26	0.89	0.50	**1.00**	0.51	0.85	0.44	**1.00**	0.66±0.26
LogPPT	0.79	0.31	0.69	0.78	0.53	0.84	0.20	0.54	0.28	0.53	0.51	0.45	0.42	0.97	0.56±0.23
LILAC	**1.00**	0.89	**1.00**	**0.87**	0.87	**1.00**	**0.97**	**0.90**	0.69	**1.00**	**1.00**	**1.00**	0.81	**1.00**	**0.93±0.10**
ByteBrain	0.99	0.91	**1.00**	0.80	**0.92**	0.96	0.81	0.81	0.63	0.99	0.98	0.97	0.78	0.97	0.90±0.11

accuracy. This is primarily because our approach is syntax-based, making it difficult to capture patterns in logs that require a holistic semantic understanding of structured text.

Overall, these comprehensive results solidify the effectiveness of our log parsing method.

5.3 Efficiency Comparison

We conducted the throughput experiments on a server equipped with an Intel(R) Xeon(R) Platinum 8336C CPU @ 2.30GHz and 128GB RAM. Our algorithm, including both training and matching, is implemented in Python, leveraging Just-In-Time (JIT) compilation for code acceleration and multi-threading parallel processing.

As shown in Fig. 6, our method significantly outperforms all baseline methods in terms of throughput across the majority of datasets. On average, *ByteBrain* achieves a remarkable throughput of 229 thousand logs per second, which is 1-3 orders of magnitude higher than most existing methods and 840.68% faster than the fastest baseline (LogCluster). This exceptional performance is particularly evident in large-scale datasets such as Thunderbird, where our method processes 519 thousand logs per second, far surpassing the next best performer.

Among the baseline methods, traditional approaches like AEL, Drain, and IPLoM show relatively consistent performance across datasets, with average throughputs ranging from 8,850 to 12,200 logs per second. However, more sophisticated methods such as UniParser and LogPPT, while potentially offering higher accuracy, suffer from significantly lower throughput, processing only 2,130 and 1,140 logs per second on average, respectively. Notably, some methods like LenMa, LogMine, and Logram failed to complete parsing on several datasets, indicating limitations in their efficiency or ability to handle diverse log formats.

To ensure a fair comparison, we also evaluated our method's throughput when using a single core (*ByteBrain* Sequential) and when Just-In-Time (JIT) compilation is disabled (*ByteBrain* w/o JIT, which also disables parallelization as multi-threading is not available without JIT). Even with a single core, our method maintains an impressive average throughput of 166,000 logs per second, significantly outperforming all baseline methods. The modest performance gain of *ByteBrain* over *ByteBrain* Sequential is expected and consistent with our findings that parallelism offers limited improvement on smaller datasets, as demonstrated in our parallelism scalability analysis (see Fig. 12). Furthermore, when JIT compilation is disabled, although the throughput decreases to 89,100 logs

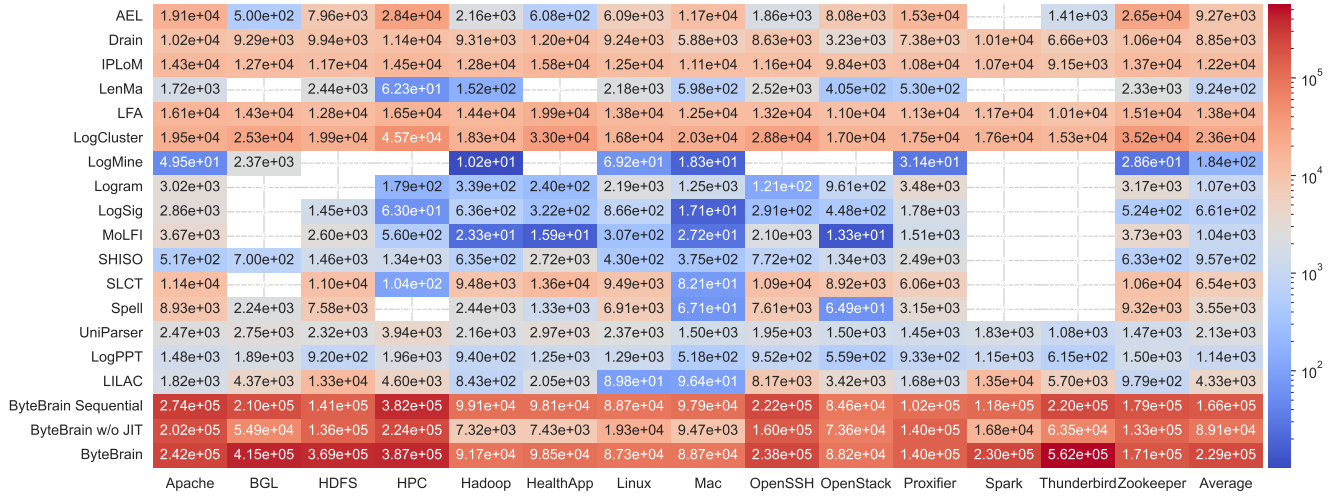

Figure 6: Throughput (#logs/second) comparison on LogHub-2.0. Missing data indicates failing to finish.

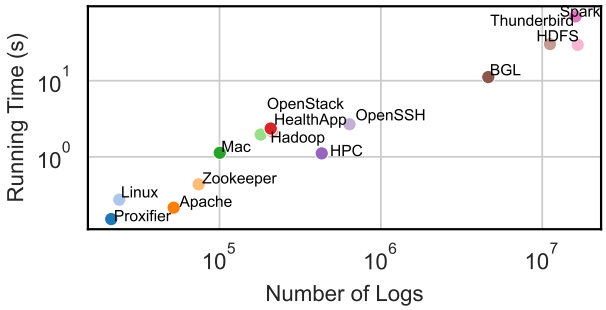

Figure 7: Our running time scales linearly with log size

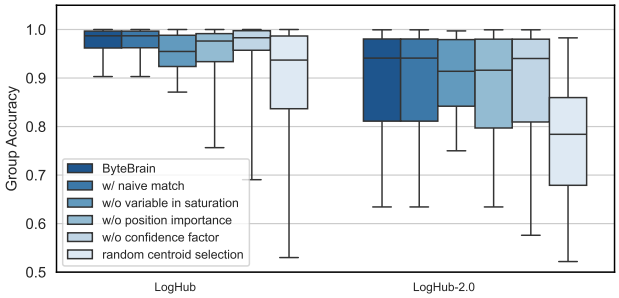

Figure 8: Our online match method maintains performance, while other techniques improve accuracy.

per second, it still surpasses baseline methods by at least an order of magnitude. These results demonstrate that the superior performance of our method stems from its algorithmic efficiency rather than merely from optimized implementation.

Fig. 7 provides further insight into *ByteBrain*'s efficiency by plotting the running time against the number of logs for different datasets. The graph shows a near-linear relationship between processing time and log volume, with most datasets clustered along a similar trajectory. This linear scaling demonstrates our method is able to efficiently handle increasing log volumes, which is a crucial feature for cloud-based log parsing services.

In summary, these efficiency comparisons highlight the superior performance of our method in processing large-scale log data. The high throughput and linear complexity make it particularly well-suited for cloud-based log parsing services, where handling massive volumes of diverse log data efficiently is crucial.

5.4 Ablation Study

We compare *ByteBrain* with the following variants with respect to either accuracy or efficiency to validate the effectiveness of our proposed techniques:

- *w/ naive match*: Use the templates assigned during clustering for training logs instead of the matching method in Section 4.8.
- *w/o position importance*: For positional similarity distance, set $w_i = 1$ in Eq. 2.
- *w/o variable in saturation*: For saturation, set $s(C) = f_c$ in Eq. 3.
- *w/o confidence factor*: For saturation, set $s(C) = f_v \cdot f_c$ in Eq. 3.
- *random centroid selection*: Randomly select initial centroids for new clusters instead of the K-Means++ strategy.
- *w/o ensure saturation increase*: Split each node into two clusters, even if their saturation scores do not increase.
- *w/o balanced group* and *w/o early stopping*
- *w/o deduplication & related techs*: Skip deduplication and dependent optimizations like balanced group and early stopping.

5.4.1 Text-based matching does not compromise accuracy. As shown in Fig. 8, directly using the templates assigned to training logs during clustering produces almost identical group accuracy compared to assigning templates by matching each log with the template texts after the clustering tree is built. Therefore, the online matching method proposed in Section 4.8 significantly reduces storage overhead with virtually no impact on matching performance.

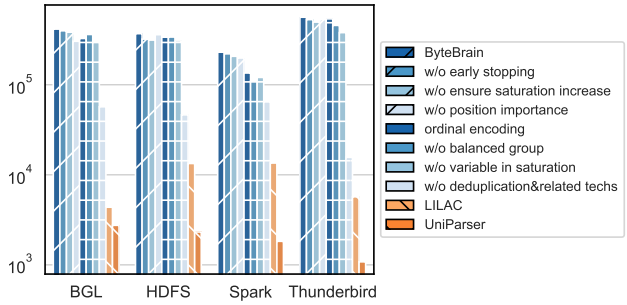

Figure 9: Our method boosts throughput and remains much faster than baselines without individual techniques.

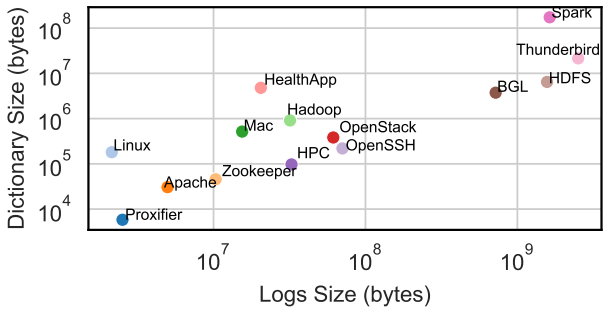

Figure 10: Dictionary size with ordinal encoding, demonstrating storage savings by hash encoding

5.4.2 Accuracy improvement brought by the proposed techniques. As shown in Fig. 8, removing variable positions from saturation calculation reduces accuracy, confirming its contribution. Interestingly, on LogHub-2.0, *w/o variable saturation* achieves higher minimum accuracy, suggesting our heuristic position estimation may occasionally misclassify positions on challenging datasets. Nevertheless, variable saturation consistently improves overall performance across all datasets.

Removing position importance from distance calculations decreases accuracy, confirming its value in capturing structural significance of log positions. This effect is more pronounced on the larger, more complex LogHub-2.0 datasets, underscoring these techniques' importance when handling diverse, voluminous log data.

Random centroid selection causes the most severe accuracy reduction, highlighting intelligent selection's critical role in achieving high parsing accuracy, especially for large-scale, diverse log datasets. The confidence factor removal from saturation calculation shows relatively minor impact across datasets, indicating its less significant contribution compared to other techniques.

This ablation study confirms each technique contributes to *ByteBrain*'s accuracy, with their importance amplified by increasing log data scale and complexity. Together, these techniques enable *ByteBrain* to maintain robust performance across diverse log types and volumes.

5.4.3 Efficiency improvement brought by the proposed techniques. As shown in Fig. 9, our proposed techniques deliver significant efficiency improvements across the four large-scale datasets (over 500MB).

The most striking impact comes from deduplication and its related techniques. Without these optimizations, the throughput drops dramatically, particularly evident in the Thunderbird dataset where performance decreases by about two orders of magnitude. This underscores the critical role of deduplication, balanced grouping, and early stopping in handling large-scale log data efficiently. Nevertheless, when compared to the best performing baselines like LILAC and UniParser, the throughput of each variant is consistently higher by one or two orders of magnitude across all datasets.

Variable saturation scoring is the second most important improvement, enabling faster convergence by reducing unnecessary splits on variable positions during training. Balanced grouping ranks third, preventing unbalanced clusters where a single node dominates, ensuring efficient processing across the clustering tree. Hash encoding also provides a notable speedup by enabling parallel token processing, unlike ordinal encoding, which requires sequential mapping. Other proposed techniques, such as position importance and ensuring saturation increase, also demonstrate noticeable improvements in throughput across different datasets, though to a lesser extent.

Importantly, while some techniques yield limited improvements on specific datasets, they consistently contribute positively across scenarios without any performance degradation. By combining all these techniques, our method achieves remarkable performance across all datasets.

5.4.4 Hash encoding reduces space consumption. We study the sizes of the dictionary files (mapping of tokens to encodings) generated by ordinal encoding on LogHub-2.0 datasets, which represent the space savings we achieve by using hash encoding. As shown in Fig. 10, as log size increases, the dictionary size required for ordinal encoding grows significantly, reaching hundreds of megabytes for large datasets like Thunderbird and Spark. Our hash encoding method eliminates the need for storing these large dictionaries entirely. This approach not only reduces storage requirements but also improves parsing efficiency by minimizing data transfer overhead. Consequently, it helps users minimize their operational costs in cloud environments, where storage incurs ongoing expenses. Moreover, the space savings become increasingly significant as the scale of log data grows, making our method particularly suitable for large-scale, cloud-based log parsing services.

5.5 Parameter Sensitivity

5.5.1 Saturation. As shown in Fig. 11, the group accuracy of our method remains relatively stable across a wide range of saturation thresholds, indicating that our method is not overly sensitive to this parameter. This robustness ensures consistent parsing results even when the threshold is not precisely tuned.

While our method maintains relatively stable performance across different saturation thresholds, the threshold still effectively controls template precision when varied across a wider range. This controllable behavior is desirable, as it allows users to adjust parsing precision according to their needs while maintaining robustness against small parameter perturbations.

To illustrate how the saturation threshold affects template generation, Table 4 shows templates produced at different threshold

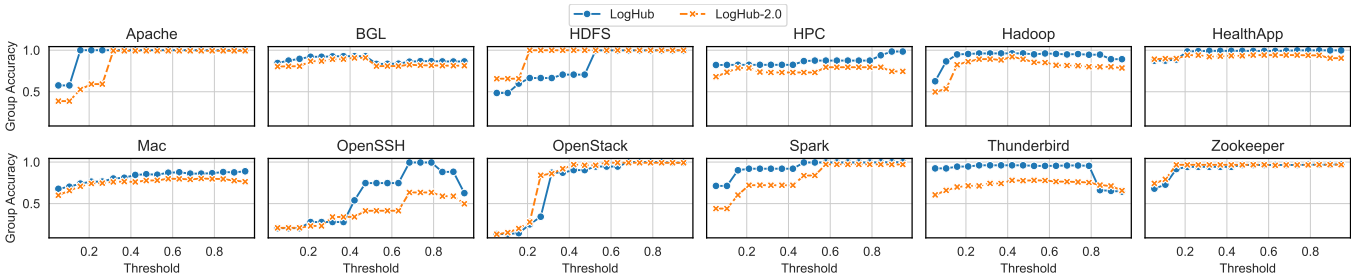

Figure 11: Group accuracy with different saturation threshold on LogHub and LogHub-2.0 (selected datasets due to space limit)

Table 4: Templates obtained by varying saturation thresholds showing adaptability

Saturation	Template
0.05	* lock * * * tag * name * ws * uid * pid *
0.78	release lock * flg * tag * name * ws * uid * pid *
	acquire lock * flags * tag * name * ws * uid * pid *
0.9	release lock * flg * tag * name android ws * uid * pid *
	release lock * flg * tag * name * ws null uid * pid *
	acquire lock * flags * tag * name android ws * uid * pid *
	acquire lock * flags * tag * name * ws null uid * pid *
0.95	release lock * flg * tag * name android ws * uid * pid *
	release lock * flg * tag * name * ws null uid * pid *
	release lock * flg * tag * name audioserver ws null uid * pid *
	release lock * flg * tag * name android ws null uid * pid *
	acquire lock * flags * tag * name android ws null uid * pid *
	acquire lock * flags * tag * name android ws * uid * pid *
	acquire lock * flags * tag * name audioserver ws null uid * pid *
	acquire lock * flags * tag * name * ws null uid * pid *

values for Android logs in LogHub. At a low threshold (0.05), the template is highly generalized with most fields marked as variables (*). As the threshold increases to 0.78, more structural elements like "release" and "acquire" are preserved. At higher thresholds (0.9 and 0.95), the templates become increasingly specific, distinguishing between similar terms like "flg" and "flags" and preserving system-specific values like "android" and "audioserver". This progression demonstrates how users can effectively control template granularity to support different analysis scenarios, from high-level pattern recognition to detailed debugging.

5.5.2 Parallelism. We investigated the impact of parallelism on throughput. In Fig. 12, as the degree of parallelism increases, we see a general trend of improved throughput across large-scale datasets, while smaller datasets like Linux and Proxifier show relatively modest gains with increased parallelism. This suggests that *ByteBrain*'s parallel processing capabilities are particularly beneficial for handling large-scale log data. Interestingly, we observe that the throughput improvement tends to plateau as parallelism increases beyond a certain point, especially for smaller datasets. This indicates that there's an optimal level of parallelism for each dataset size, beyond which additional parallel processing may not yield significant performance gains.

6 Industrial Evaluation

ByteBrain has been successfully deployed as part of Volcano Engine's Torch Log Service (TLS) and is now available to invited users. In production, we use a Go-based service to schedule training tasks, which are executed on separated Pods and utilize the same Python implementation as described in Section 5. For online matching, which must integrate with conventional text indexing systems, we reimplemented the matching module in C++ and Rust and embedded it directly into the log indexing pipeline. This strategic integration eliminates cross-server data transfer (between C++ indexing code and our log parsing module), substantially reducing I/O overhead and log parsing latency.

Table 5: Performance evaluation using actual production data from the TLS on Volcano Engine, demonstrating *ByteBrain*'s effectiveness in actual cloud environments.

Topic Scenario	Log Volume	Model Size	Training Time
Text stream processing	189 MB/s	3 MB	0.91s
Webserver access log	57.8 MBs	10 MB	7.98s
Webserver access log	47.7 MB/s	3 MB	1.02s
Go HTTP API server	3.51 MB/s	7 MB	1.65s
Go search server	2.46 MB/s	7 MB	4.64s

The system enables users to organize queried logs by their corresponding templates, providing a structured and intuitive view of complex log data. A distinctive feature is the real-time precision adjustment capability through an interactive slider in the web interface, allowing users to dynamically control template granularity based on specific analytical requirements. This interactive capability helps users identify patterns and anomalies more effectively by allowing them to switch between different levels of abstraction on demand. Based on the parsed log templates, users can save selected templates to a template library, which can then be used to configure alerts (e.g., sudden changes in template count or the appearance of new templates). Users can also compare the templates generated across different time periods to analyze changes in log patterns.

Table 5 presents performance metrics collected from diverse log topics in production environments, representing characteristic real-world deployment scenarios. As illustrated, the system processes exceptionally high log volumes, reaching up to 189 MB/s, which is equivalent to millions of logs per second. Despite this substantial throughput, our algorithm completes model training sessions within seconds, demonstrating remarkable computational efficiency. The system maintains an end-to-end visible latency of approximately 5-15 seconds per log (encompassing ingestion through completed indexing, enabling user queries), confirming that our log parsing method effectively keeps pace with real-time log generation rates. It is worth noting that this latency figure includes both log parsing and traditional text indexing operations. This level

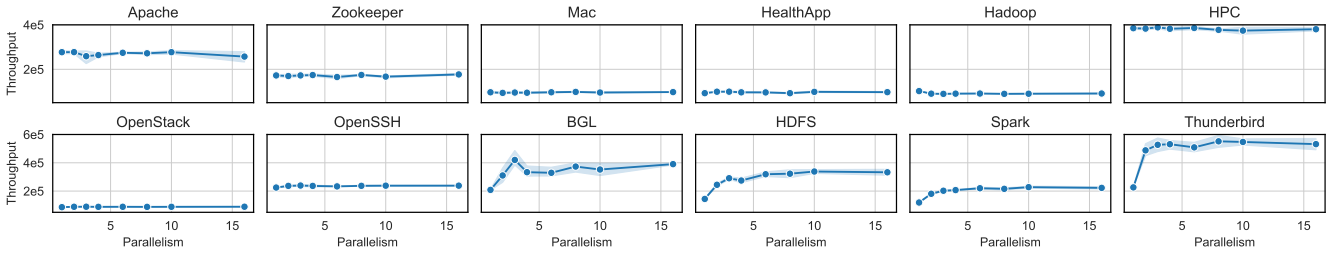

Figure 12: Throughput vs/ parallelism on LogHub-2.0 (selected datasets due to space limit). The datasets are sorted by size.

of responsiveness satisfies the requirements for most real-time application scenarios.

The results in Table 5 also highlight the storage efficiency of our method. The size of log parsing model for each topic is only a few megabytes, which is significantly smaller than the size of the corresponding raw log text. This minimal model size ensures that our approach introduces negligible storage overhead, making it highly cost-efficient for large-scale cloud applications.

Due to privacy constraints preventing direct access to user logs, we cannot determine optimal thresholds for calculating desired template counts and parsing accuracy for each topic. Nevertheless, we observe that the number of most precise templates (saturation ≥ 0.9) typically ranges between 1,000 and 10,000 across different deployment scenarios.

The successful integration of *ByteBrain* within Volcano Engine demonstrates its practical value in production environments, where it effectively combines high-throughput processing capabilities with flexible, user-centric template management. This operational validation complements our experimental findings and confirms *ByteBrain*'s suitability for enterprise-scale log parsing applications.

7 Discussion

In this section, we discuss several limitations of our approach and potential improvements that can be made in the future.

First, while our approach achieves excellent accuracy and efficiency, it inherently lacks semantic understanding of log content. Unlike human operators who naturally group logs based on their meanings, our syntax-based approach can only rely on structural similarities. Semantic parsers leveraging natural language processing techniques can interpret logs like humans do, intelligently grouping similar logs and separating different ones, though at higher computational costs. Our choice of a syntax-based approach prioritizes compute-efficiency and cost-efficiency, enabling efficient processing of massive log volumes while maintaining high accuracy. Looking forward, we may combine semantic-based understanding with syntax-based methods to achieve both efficiency and semantic understanding in future work, creating a hybrid solution that balances the strengths of both approaches.

The second limitation of our approach, similar to other syntax-based methods [12, 30], is the inability to directly parse logs of varying lengths into the same template. This limitation stems from syntax-based methods relying solely on comparing tokens at identical positions to determine log similarity, without understanding the semantic meaning. This becomes particularly challenging when logs contain variable-length elements, such as when a print statement outputs a dynamic list, resulting in semantically similar logs being parsed into different templates despite originating from the same log statement. In this paper, we deliberately choose not to implement dynamic matching solutions (e.g., using longest common subsequence to compare logs of different lengths) to address this challenge. The reason is that allowing wildcards to match variable numbers of tokens would require a search process during online matching to determine the optimal token spans for each wildcard. Such an approach would significantly increase the computational complexity of online matching, making it impractical in cloud environments where millions of logs need to be matched per second. Instead, we propose a simple yet effective optimization at the query result processing layer. For example, consider three templates generated from the statement `print(f'users={users}')` where the `users` list contains one, two, and three elements: `users *`, `users * *`, and `users * * *`. When processing query results before presentation, we merge consecutive wildcards in each template, resulting in `users *` for all three cases. We then group logs with the same merged template together in the response. This optimization balances user experience and system performance: users see one intuitive template `users *` that accommodates variable-length lists, while our underlying system maintains efficiency by using the original fixed-length templates during log parsing and matching.

8 Conclusion

This paper presents an adaptive and efficient log parsing approach, which is optimized as a cloud service, for processing diverse, massive logs from different tenants. Our method offers real-time adjustable parsing precision, incorporates efficiency-enhancing techniques (deduplication, balanced grouping, early termination), and minimizes storage overhead through hash encoding and text-based matching. Comprehensive evaluations on public datasets demonstrate state-of-the-art efficiency with comparable accuracy, while production deployment validates its practical effectiveness and efficiency. By integrating out-of-the-box log parsing and intelligent analysis capabilities into our log service, we enhances its overall intelligence and usability. Future work will extend the framework to handle more complex log patterns, including structural content and dynamic-length variables, enabling broader applications.

References

[1] [n. d.]. *Amazon CloudWatch*. https://aws.amazon.com/cn/cloudwatch/ Accessed: April 1, 2025.

[2] [n. d.]. *Cloud Monitoring as a Service | Datadog*. https://www.datadoghq.com/ Accessed: April 1, 2025.

[3] GitHub [n. d.]. *Elastic/Logstash: Logstash - Transport and Process Your Logs, Events, or Other Data*. GitHub. https://github.com/elastic/logstash Accessed: April 1, 2025.

[4] Splunk [n. d.]. *Splunk | The Key to Enterprise Resilience*. Splunk. https://www.splunk.com Accessed: April 1, 2025.

[5] David Arthur and Sergei Vassilvitskii. [n. d.]. K-Means++: The Advantages of Careful Seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (USA, 2007-01-07) *(SODA '07)*. Society for Industrial and Applied Mathematics, 1027–1035.

[6] An Ran Chen, Tse-Hsun Chen, and Shaowei Wang. 2022. Pathidea: Improving Information Retrieval-Based Bug Localization by Re-Constructing Execution Paths Using Logs. *IEEE Transactions on Software Engineering* 48, 8 (Aug. 2022), 2905–2919. https://doi.org/10.1109/tse.2021.3071473

[7] Xiaolei Chen, Peng Wang, Jia Chen, and Wei Wang. 2023. AS-Parser: Log Parsing Based on Adaptive Segmentation. *Proceedings of the ACM on Management of Data* 1, 4 (Dec. 2023), 1–26. https://doi.org/10.1145/3626719

[8] Guojun Chu, Jingyu Wang, Qi Qi, Haifeng Sun, Shimin Tao, and Jianxin Liao. 2021. Prefix-Graph: A Versatile Log Parsing Approach Merging Prefix Tree with Probabilistic Graph. In *2021 IEEE 37th International Conference on Data Engineering (ICDE) (ICDE 2021)*. IEEE, Chania, Greece, 2411–2422. https://doi.org/10.1109/icde51399.2021.00274

[9] Hetong Dai, Heng Li, Che Shao Chen, Weiyi Shang, and Tse-Hsun Chen. 2020. Logram: Efficient Log Parsing Using n-Gram Dictionaries. *IEEE Transactions on Software Engineering* 48, 3 (Jan. 2020), 1–1. https://doi.org/10.1109/tse.2020.3007554 arXiv:2001.03038 [cs]

[10] Min Du and Feifei Li. 2016. Spell: Streaming Parsing of System Event Logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. 859–864. https://doi.org/10.1109/ICDM.2016.0103

[11] 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 '16)*. Association for Computing Machinery, New York, NY, USA, 1573–1582. https://doi.org/10.1145/2983323.2983358

[12] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. 2017. Drain: An Online Log Parsing Approach with Fixed Depth Tree. In *2017 IEEE International Conference on Web Services (ICWS) (ICWS Computer Society 2017)*. IEEE, Honolulu, HI, USA, 33–40. https://doi.org/10.1109/icws.2017.13

[13] 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) (ICSE 2023)*. IEEE, Melbourne, Australia, 881–893. https://doi.org/10.1109/icse48619.2023.00082

[14] 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. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 137–160. https://doi.org/10.1145/3643733 arXiv:2310.01796 [cs]

[15] 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. https://doi.org/10.48550/ARXIV.2308.10828 arXiv:2308.10828 [cs]

[16] 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 (QSIC 2008)*. IEEE, Oxford, United Kingdom, 181–186. https://doi.org/10.1109/qsic.2008.50

[17] 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 '22)*. Association for Computing Machinery, New York, NY, USA, 1095–1106. https://doi.org/10.1145/3510003.3510101

[18] 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) (ICSE 2023)*. IEEE, Melbourne, Australia, 2438–2449. https://doi.org/10.1109/icse48619.2023.00204 arXiv:2302.07435 [cs]

[19] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log Clustering Based Problem Identification for Online Service Systems. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. ACM, Austin Texas, 102–111. https://doi.org/10.1145/2889160.2889232

[20] 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. (Sept. 2019).

[21] Yudong Liu, Xu Zhang, Shilin He, Hongyu Zhang, Liqun Li, Yu Kang, Yong Xu, Minghua Ma, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. 2022. UniParser: A Unified Log Parser for Heterogeneous Log Data. In *Proceedings of the ACM Web Conference 2022 (WWW '22)*. ACM, Virtual Event, Lyon France, 1893–1901. https://doi.org/10.1145/3485447.3511993 arXiv:2202.06569 [cs]

[22] Adetokunbo A.O. 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 (KDD09)*. ACM, Paris France, 1255–1264. https://doi.org/10.1145/1557019.1557154

[23] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. 2018. A Search-Based Approach for Accurate Identification of Log Message Formats. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. 167–16710.

[24] Masayoshi Mizutani. 2013. Incremental Mining of System Log Format. In *2013 IEEE International Conference on Services Computing*. 595–602. https://doi.org/10.1109/SCC.2013.73

[25] Meiyappan Nagappan and Mladen A. Vouk. 2010. Abstracting Log Lines to Log Event Types for Mining Software System Logs. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 114–117. https://doi.org/10.1109/MSR.2010.5463281

[26] Keiichi Shima. 2016. Length Matters: Clustering System Log Messages Using Length of Words. arXiv:1611.03213 [cs]

[27] 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 '11)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2063576.2063690

[28] Shimin Tao, Weibin Meng, Yimeng Cheng, Yichen Zhu, Ying Liu, Chunning Du, Tao Han, Yongpeng Zhao, Xiangguang Wang, and Hao Yang. 2022. LogStamp: Automatic Online Log Parsing Based on Sequence Labelling. *ACM SIGMETRICS Performance Evaluation Review* 49, 4 (June 2022), 93–98. https://doi.org/10.1145/3543146.3543168

[29] R. Vaarandi. 2003. A Data Clustering Algorithm for Mining Patterns from Event Logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003) (IEEE Cat. No.03EX764)*. 119–126. https://doi.org/10.1109/IPOM.2003.1251233

[30] Xuheng Wang, Xu Zhang, Liqun Li, Shilin He, Hongyu Zhang, Yudong Liu, Lingling Zheng, Yu Kang, Qingwei Lin, Yingnong Dang, Saravanakumar Rajmohan, and Dongmei Zhang. 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 (ESEC/FSE '22)*. ACM, Singapore, Singapore, 1198–1208. https://doi.org/10.1145/3540250.3549176

[31] 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 (ICSE '24)*. ACM, Lisbon Portugal, 199:1–199:12. https://doi.org/10.1145/3597503.3639155

[32] Kundi Yao, Mohammed Sayagh, Weiyi Shang, and Ahmed E. Hassan. 2022. Improving State-of-the-Art Compression Techniques for Log Management Tools. *IEEE Transactions on Software Engineering* 48, 8 (Aug. 2022), 2748–2760. https://doi.org/10.1109/TSE.2021.3069958

[33] Siyu Yu, Pinjia He, Ningjiang Chen, and Yifan Wu. 2023. Brain: Log Parsing With Bidirectional Parallel Tree. *IEEE Transactions on Services Computing* 16, 5 (Sept. 2023), 3224–3237. https://doi.org/10.1109/tsc.2023.3270566

[34] Tianzhu Zhang, Han Qiu, Gabriele Castellano, Myriana Rifai, Chung Shue Chen, and Fabio Pianese. 2023. System Log Parsing: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 35, 8 (2023), 1–20. https://doi.org/10.1109/tkde.2022.3222417 arXiv:2212.14277 [cs]

[35] Nengwen Zhao, Honglin Wang, Zeyan Li, Xiao Peng, Gang Wang, Zhu Pan, Yong Wu, Zhen Feng, Xidao Wen, Wenchi Zhang, Kaixin Sui, and Dan Pei. 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 (ESEC/FSE '21)*. ACM, Athens Greece, 1404–1415. https://doi.org/10.1145/3468264.3473933

[36] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R. Lyu. 2023. Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE) (ISSRE 2023)*. IEEE, Florence, Italy, 355–366. https://doi.org/10.1109/issre59848.2023.00071 arXiv:2008.06448 [cs]

[37] 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) (ICSE 2019)*. IEEE, Montreal, QC, Canada, 121–130. https://doi.org/10.1109/icse-seip.2019.00021