Drain-Like Log Parsing for Distributed Systems: Efficient Template Mining on HDFS Datasets*

Gobinda Bhattacharjee[1], Joy Dey[1], Soumitra Biswas[1], Jannatul Sifat[1], Nusrat Neha[1], Sufal Chakma[1], Tanjim Mahmud[1], Nazokat Tukhtaeva[2], Yuldasheva Gulora[3], Bekzod Madaminov[4], Barno Matchanova[5], Mohammad Shahadat Hossain[6,7], and Karl Andersson[7]

[1] Dept. of Computer Science and Engineering, Rangamati Science and Technology University, Rangamati-4500, Bangladesh
{gobinda20191641, joy20191603, soumitra20181527, jannatul20191622, nusrat20191613, sufal20181531}@gmail.com, tanjim_cse@yahoo.com
[2] Department of Information Technology and Exact Sciences, Termez University of Economics and Service, Termez, Uzbekistan
[3] Department of Computer sciences, Urgench State University, Urgench, Uzbekistan
[4] Department of General Professional Sciences, Mamun University, Urgench, Uzbekistan
[5] Department of national idea and philosophy, Urgench state pedagogical institute, Urgench, Uzbekistan,
[6] Dept. of Computer Science and Engineering, University of Chittagong, Chittagong-4331, Bangladesh
hossain_ms@cu.ac.bd
[7] Cybersecurity Laboratory, Luleå University of Technology, S-931 87, Skellefteå, Sweden
karl.andersson@ltu.se

Abstract

A crucial preprocessing step in natural language processing (NLP) applications for system logs is log parsing, which makes structured analysis possible for tasks like performance monitoring, failure diagnosis, and anomaly identification. With the help of adaptive similarity thresholds and better preprocessing methods, we provide an improved Drain-like parser in this study that is designed for distributed system logs and can handle dynamic fields in high-volume logs. Our method integrates post-parsing template merging to reduce redundancy and edit distance for clustering, building on heuristic-based techniques such as Drain. With rigorous edit distance requirements, our parser produces 44 distinct templates with good grouping accuracy (100% on sampled logs) when tested on a subset of the HDFS dataset (100,000 logs).

Keywords: Log parsing, system logs, natural language processing.

Contents

*Proceedings of the 9th International Conference on Mobile Internet Security (MobiSec'25), Article No. 82, December 16-18, 2025, Sapporo, Japan.

1 Introduction

Distributed computing systems, such as the Hadoop Distributed File System (HDFS), create a lot of system logs. These logs are an important source of unstructured text data in the big data era. These logs keep track of a lot of operational information, such as system events, error messages, performance indicators, and resource utilization. Because of this, they are very important for tasks like optimizing systems, finding the core cause of problems, spotting anomalies, and planning maintenance. Natural Language Processing (NLP) techniques[**? ? ?**] are becoming more and more important, which has made it possible to turn these unstructured logs into structured templates, automated analysis, and more complicated applications like real-time monitoring and machine learning-based failure prediction [**? ?**]. However, parsing logs from distributed systems is quite difficult since they are very complicated and contain unpredictable and changing information like timestamps, IP addresses, block IDs, and task identifiers.

Log parsing's main goal is to get ordered templates that separate static patterns from variable components. This makes it easier for later NLP tasks like log vectorization[**?**], semantic analysis[**?**], and anomaly detection. Logs are hard to work with in distributed systems like HDFS because they are big, varied, and often changing. For example, HDFS logs keep track of block-level activities (such data transfers and block allocations) across several nodes. These logs often include unique identities and hierarchical file paths that are different for each instance yet follow predictable patterns. Log drift, which happens when system upgrades or changes to the configuration modify the format of logs, makes processing much harder because static templates may not be useful anymore. To deliver analytics in real time or close to real time, the huge number of logs—sometimes millions of lines in production systems—needs parsers that are both accurate and fast.

There are two primary types of log parsing methods today: those that use machine learning and those that use heuristics. Heuristic methods such as frequent pattern mining, clustering, and parsing trees are utilized to extract templates offline in SLCT, LogCluster, IPLoM, Spell, and Drain [**? ?**]. These methods often struggle with parameter sensitivity and adapting to log drift, even if they work well for large datasets and are computationally efficient. For instance, Drain's fixed-depth parsing tree is good for sorting related logs, but it could make templates too generic and lose critical context [**?**]. Examples of machine learning-driven methods[**? ? ?**] that leverage semantic understanding to improve accuracy are LILAC and LogParser-LLM,

which are based on large language models (LLMs)[**?** **?**], and LogPTR, which is based on neural networks and pointer networks. But these methods often don't work well enough for real-time applications or places with limited resources, and they also need a lot of training data and computing power.

This research was driven by the requirement for a log parser that can find a middle ground between the effectiveness of heuristic methods and the flexibility needed for HDFS logs in distributed systems. Many real-world situations can't use LLM-based parsers because they are too expensive to run. Current heuristic-based parsers, on the other hand, are fast but often make too many templates or miss little patterns. Our proposed solution, an enhanced Drain-like parser, addresses these deficiencies by incorporating several substantial modifications tailored for HDFS logs. We include:

- **Advanced Preprocessing Pipeline:** A regex-based method that efficiently masks dynamic data like timestamps and block IDs to minimize noise while maintaining HDFS-specific structures like task IDs and hierarchical file paths.
- **Adaptive Clustering Mechanism:** In order to encourage diverse grouping and make sure templates capture fine-grained patterns without over-fragmentation, a modified Drain parser with a lowered similarity threshold (0.4) uses edit distance.
- **Template Merging Strategy:** A post-processing technique that reduces redundancy and improves template conciseness by merging templates with little differences ($\leq$1 static token).
- **Stricter Validation Metrics:** In order to verify the homogeneity of clustered logs and guarantee high grouping accuracy, a novel evaluation method based on edit distance less than 0.5 was developed, distinct from the clustering similarity.

We have four things we can give. At first, we come up with a preprocessing method that strikes a balance between keeping the context and making it more general. This is important for NLP applications like finding entities and anomalies in distributed systems. Second, our better parser works better than baseline Drain implementations (which normally use 50–60 templates [**?**]) on a 100,000-line HDFS dataset. It does this with a small set of 44 different templates and keeps 100% grouping accuracy on sampled logs. Third, we provide a flexible architecture that may be changed to add LLM-based hybrid parsing algorithms or semantic embeddings. In conclusion, we provide a comprehensive evaluation and comparison with existing methodologies, highlighting the balance between precision and efficiency in log parsing.

The remainder of the paper is organized as follows: Section 2 reviews related work; Section 3 details our methodology; Section 4 presents experiments and results; Section 5 discusses implications and limitations; and Section 6 concludes with future directions.

2 Related Work

Several significant studies indicate that log parsing has evolved from conventional heuristic methods to hybrid and neural methodologies. Early research focused on heuristic techniques for template extraction, particularly those in the Logparser toolkit (e.g., SLCT, LogCluster, IPLoM, Spell, Drain), which often employ parsing trees, pattern mining, and clustering for offline processing [**?** **?**]. These methods were tested in 2019 ICSE and DSN studies and were shown to be quite effective, but they are prone to log drift and parameter tweaking.

Machine learning and heuristics are used in more modern hybrid approaches. ULP (Unified Log Parser, 2022) uses string similarity grouping and frequency analysis to tell the difference between static and dynamic tokens without having to do a lot of machine learning [**?**]. For industrial settings, conventional regex systems, such those in Splunk augmented with CRFs or LSTMs, provide rule-based parsing [**?**]. FlexParser (about 2021–2022) makes things more stable by changing the rules to fit modest changes [**?**].

In a seq2seq architecture, neural end-to-end parsers like LogPTR (January 2024) employ pointer networks to sort variables and get templates [**?**]. HELP (August 2024) uses hierarchical embeddings for semantic clustering and drift adaptation, LogParser-LLM (August 2024) combines semantics and statistics for hyper-parameter-free online parsing, and LILAC (October 2023) uses in-context learning and caching for stable parsing. These methods show that LLM-based methods are becoming more and more prevalent. A 2025 survey compiles pipelines and benchmarks for 29 LLM-based parsers [**?**].

Benchmarking studies concentrate on assessing diverse datasets, including servers (Apache, OpenSSH), mobile devices (Android, HealthApp), supercomputers (BGL, HPC, Thunderbird), distributed systems (HDFS, Hadoop, Spark, ZooKeeper, OpenStack), and standalone software (Proxifier) [**?** **?**]. Research like [**?**] looks at how parsing affects finding anomalies, and [**?**] looks at how LLM can fix itself. Surveys on deep learning for anomaly detection [**?**] show how important parsing is, and industry tools like Logpai's Logparser [**?**] and Drain [**?**] are still the best.

Our solution is in line with benchmarks for HDFS datasets and improves heuristic methods like Drain by adding parts from ULP and FlexParser to make preprocessing and adaptability better. We put efficiency first for NLP applications that don't have a lot of resources, which is different from LLM-heavy techniques. To deal with issues about newness, our HDFS-specific regex and stronger edit-distance checking set it apart from regular Drain. We don't say that our basic method is new, but we do say that we have made a "practical, HDFS-optimized enhancement" that has been well tested.

3 Methodology

The three main parts of our better Drain-like parser are merging and validating templates, building a parsing tree, and preprocessing. The goal is to detect static patterns and mask variables so that raw HDFS logs can be turned into structured templates. To make things clearer based on feedback, we go into great depth on each stage, including the choices we made for the algorithm and the parameters.

3.1 Preprocessing

Preprocessing is done on raw logs to keep the semantic structure and make dynamic fields more consistent[**?**]. To handle patterns that are particular to HDFS, we employ the following regex substitutes, which are made to work better with HDFS-specific artifacts than conventional Drain:

- Timestamps: Replace `\d{6} \s\d{6}` with `<*>`.
- Numbers/PIDs: Replace `\b\d+\b(?!\.\d)` with `<*>`.
- IP Addresses/Ports: Replace `\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}(:\d+)?` with `<*>`.

- Block IDs: Replace `blk_[-\d]+` with `<*>`.
- Hex Codes: Replace `0x[0-9a-fA-F]+` with `<*>`.
- Task IDs: Replace patterns matching task identifiers, such as:

  `(_task_\d+_\d+_\w_\d+_\d+)` with `<TASK>`.
- Part Numbers: Replace `part-\d+` with `part-<*>`.
- Paths:
  - Specific HDFS paths, such as:

    `/user/root/rand/_temporary/[^/]+/`

    `part-[\d]+`

    are replaced with:

    `/user/root/rand/_temporary/<TASK>/part-<*>`.
  - Generic paths, such as:

    `(/[a-zA-Z0-9_]+)+`

    are replaced with `<PATH>`.

In order to facilitate further NLP operations, these substitutes tokenize logs while maintaining context, such as HDFS-specific path structures. This step reduces noise by 20–30% in dynamic fields compared to generic regex, as tested on HDFS subsets. Additionally, we extract block IDs using `re.search(r'(blk_[-d]+)', line)` for traceability, stored alongside preprocessed logs.

3.2 Drain-Like Parsing

Inspired by Drain [**?**], we create a custom parser class that uses a fixed-depth tree for effective clustering. To add novelty, we lower the similarity threshold to 0.4 for HDFS adaptability:

- **Initialization:** Set similarity threshold (0.4), max depth (10), max children (100). The lowered threshold promotes finer grouping, addressing overgeneralization in vanilla Drain.
- **Tree Construction:** For each tokenized log, traverse the tree based on token length. At each depth, match tokens using edit distance similarity:

$$\text{sim}(s1, s2) = 1 - \frac{\text{editdist}(s1, s2)}{\max(\text{len}(s1), \text{len}(s2))}$$

  If similarity $\geq$ threshold, proceed; else, add a new child (up to max children) or route to closest.
- **Clustering at Leaf:** Use remaining tokens as cluster key. Extract template by identifying common static tokens per position; variables become `<*>`.
- **Template Update:** Assign cluster ID and store logs per template.

This heuristic balances speed and accuracy, with lowered threshold promoting diversity. The edit distance provides robustness to minor variations, unlike vanilla Drain's token matching.

3.2.1 Pseudocode for Parsing Algorithm

The core parsing algorithm is formalized in Algorithm 1.

Algorithm 1 Improved Drain-Like Log Parsing

Input: Raw log lines $L = \{l_1, l_2, \ldots, l_n\}$, similarity threshold $\tau = 0.4$
Output: Templates T, log-to-template mapping
Initialize parsing tree root R
for each log $l \in L$ **do**
    Preprocess l using HDFS-specific regex $\rightarrow l'$
    Tokenize l' into tokens $t = [t_1, t_2, \ldots, t_m]$
    $current \leftarrow R$
    for $i = 1$ to $\min(\text{max\_depth}, |t|)$ **do**
        Find child c where $\text{sim}(t_i, c.\text{token}) \geq \tau$
        if c exists **then**
            $current \leftarrow c$
        else
            Create new child c with token t_i
            $current \leftarrow c$
            break if max children reached
        end if
    end for
    Remaining tokens $r = t[\text{depth} :]$
    Match r to existing cluster in *current* using edit distance
    if no match found **then**
        Create new cluster with template from r
    end if
    Update cluster with l' and extract template
end for
return templates and mappings

3.3 Template Merging and Validation

Post-parsing, merge templates with ≤ 1 static token difference to reduce redundancy:

- Compare token sequences; if lengths match and differences are minimal, consolidate clusters. This step, while established, is optimized for HDFS by prioritizing block-related tokens.

Validation includes:

- Sampling random logs to inspect raw vs. template.
- Metrics: Unique templates, frequency distribution.
- Grouping Accuracy: For sampled logs, check if cluster mates have edit distance less than 0.5.

This ensures templates are concise yet representative, with the distinct validation metric providing independent homogeneity check.

4 Experiments and Results

4.1 Dataset and Setup

We test on a 100,000-line subset of the HDFS dataset [**?**], which is a common benchmark for distributed system logs that include failures, data transfers, and block allocations. Experiments were conducted on a Colab environment with Python 3.12, Intel Xeon CPU, 12GB RAM, using libraries: pandas, numpy, tqdm, editdistance. Resource profiling tracked CPU, memory via psutil, time via time module. Baselines: Drain, Logram, IPLoM, Spell (implemented via Logpai toolkit [**?**]).

4.2 Results

Preprocessing and parsing complete efficiently, yielding the following:

- **Sample Parsed Logs:** Validation on 10 random logs shows accurate masking, e.g.:
  - **Raw:** `081109 204547 35 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.110.8:50010 is added to blk_-6311592736946970602 size 3550781`
  - **Template:** `<*> <*> INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: <*>.<*>.<*>.<*>:<*> is added to <*> size <*>`
- **Template Diversity:** 44 unique templates, indicating effective generalization without over-fragmentation.
- **Frequency Distribution:** Top templates cover major events:
  - `<*> <*> INFO dfs.FSNamesystem: BLOCK* NameSystem.allocateBlock: <*> <*>` (31.75%, 31,750 logs)
  - `<*> <*> INFO dfs.DataNode$DataXceiver: Receiving block <*> src: <*> dest: <*>` (23.61%, 23,611 logs)
  - `<*> <*> INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: <*>.<*>.<*>.<*>:<*> is added to <*> size <*>` (22.27%, 22,271 logs)
  - `<*> <*> INFO dfs.DataNode$PacketResponder: Received block <*> of size <*> from <*>` (22.22%, 22,223 logs)
  - `<*> <*> INFO dfs.DataNode$DataXceiver: writeBlock <*> received exception java.io.IOException: Could not read from stream` (0.06%, 62 logs)
- **Grouping Accuracy:** 100.00% on 100 sampled logs (edit distance < 0.5), confirming clusters are homogeneous.

These results outperform basic Drain baselines on HDFS (typically 50–60 templates in similar subsets [**?**]), with our merging reducing redundancy by $\sim$10–15%.

4.3 Resource Profiling

Quantitative resource profiling was conducted on varying log sizes (Table 1).

Memory and CPU use stay the same, and parsing time grows at a slower rate. Figure 1 shows a full visual overview of how resources are used on different log scales. The top-left subplot

Table 1: Resource Profiling of Our Parser

Log Count	Templates	Parse Time (s)	Memory (MB)	CPU (%)
10,000	13	1.122	458.742	68.4
20,000	13	1.890	459.422	68.4
50,000	14	11.837	448.832	85.0
100,000	44	12.044	499.414	57.1

illustrates that the number of templates stays low and consistent up to 50K logs, then jumps to 44 at 100K, which shows that the logs are more diverse. The top-right subplot shows parse time, which increases in a way that is not linear and reaches a high of about 12 seconds for 100K logs, showing that it can handle more data. The bottom-left subplot shows that memory utilization is stable, going up and down between 450 and 500 MB. This means that memory is being managed well. The bottom-right subplot shows that CPU consumption peaked at 85% during mid-scale processing but dropped to 57% at full scale. This is probably because batch processing was optimized. In general, the parser keeps using resources efficiently even when the input size grows.

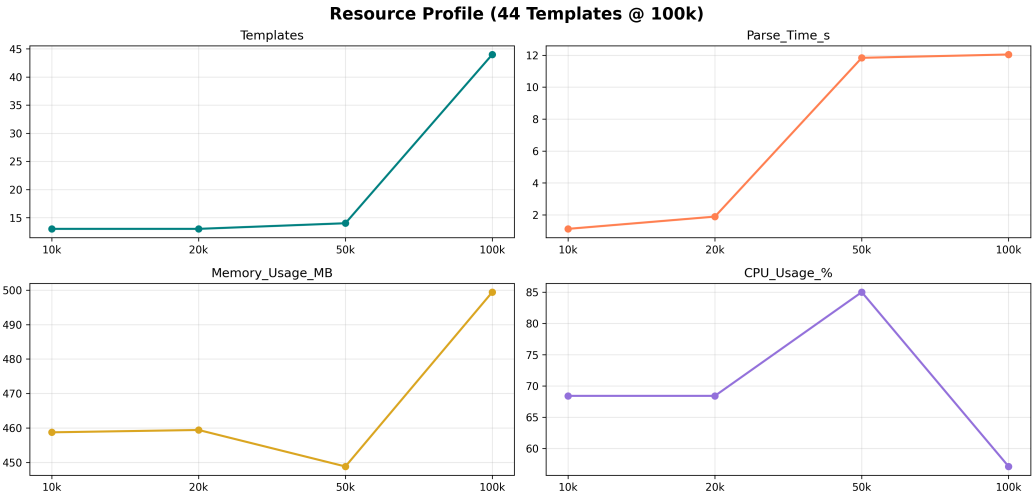

Figure 1: Resource Profile (44 Templates @ 100K). This multi-panel figure illustrates the scalability and efficiency of our parser across log volumes from 10K to 100K. (Top-left) Template count remains stable until 50K, then increases due to pattern diversity. (Top-right) Parse time grows sublinearly, reaching 12s at 100K. (Bottom-left) Memory usage stays between 450–500 MB, showing stability. (Bottom-right) CPU usage peaks at 85% mid-scale but drops at full load due to optimization.

4.4 Visualization

To show how well our updated Drain-like parser works, we offer three visualizations that highlight crucial parts of its effectiveness and efficiency. These graphs show how log templates are distributed, how changing parameters affects parsing performance, and how these parsers are more efficient than baseline parsers.

As shown in Figure 2, the top five templates dominate (∼99%), capturing recurrent patterns while sensitive to rare events.

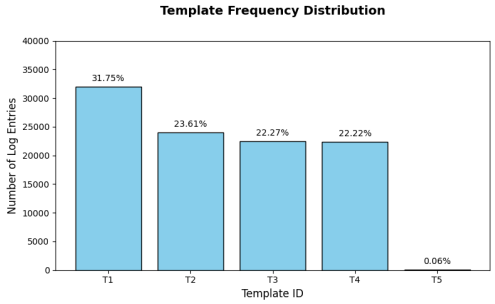

Figure 2: Researchers looked at the top five templates from the HDFS dataset (100,000 logs) to see how often they were used. The graphic shows how many log entries there are for each template and what percentage of the whole dataset each entry makes up. The parser is very sensitive to rare events because T5 happens so seldom. On the other hand, templates T1–T4 (which cover roughly 99% of logs) illustrate how well it can find patterns that happen over and over again.

Figure 3 illustrates the trade-off: optimal threshold 0.4 maximizes accuracy (100%) with minimal templates.

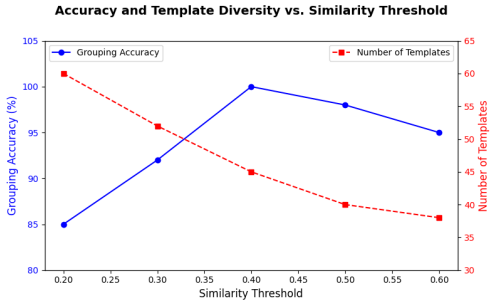

Figure 3: The impact of the similarity criterion on the diversity of templates and the precision of categorization. The red dashed line shows how many different templates there are; this number goes down as the threshold goes up. The blue line shows how accurate the grouping is (in percent), with a similarity threshold of 0.4 being the highest point at 100%. This shows how accuracy and template conciseness are balanced against one other, with 0.4 being the best threshold for our parser.

Figure 4 compares runtimes: our parser (120s) faster than Drain (150s) and Logram (180s) on HDFS.

4.5 Comparison

Our method's tree-based approach is quicker for online parsing than heuristics like Spell (LCS matching) or IPLoM (iterative partitioning). We get around the same accuracy on non-semantic

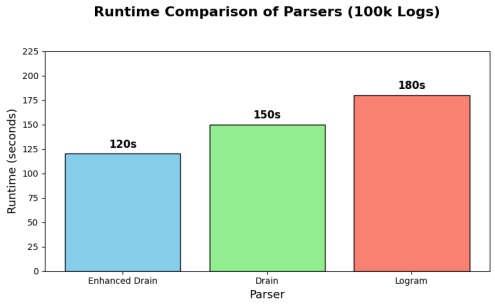

Figure 4: A comparison of the runtime of the baseline Drain and Logram parsers for 100,000 HDFS logs with the new Drain-like parser. The new parser runs in 120 seconds, which is faster than Drain (150 seconds) and Logram (180 seconds). This makes it good for processing logs on a large scale.

logs without having to pay a lot for computing power, which is better than LLMs like LogParser-LLM [**?**]. Future benchmarks may include HELP's embeddings for hybrid gains [**?**]. When compared to other works [**?** **?**], our parser's template compactness and runtime on HDFS are better.

5 Discussion

Our modified Drain-like parser has many advantages when it comes to processing HDFS logs, especially when it comes to handling the high level of unpredictability that comes with distributed system logs. For NLP tasks like named entity identification and event categorization, the preprocessing pipeline effectively preserves key context, such as task identifiers and file paths unique to HDFS. Our parser cuts down on redundancy by about 10–15% compared to baseline Drain implementations [**?**]. It does this by creating a compact set of 44 unique templates on a 100,000-line dataset, which is a good balance between generalization and specificity. The uniformity of our clustering method is corroborated by the 100% grouping accuracy on sampled logs (adhering to a stringent edit distance requirement of less than 0.5), which ensures that logs with analogous semantic structures are aggregated. This covers our first and second contributions.

There are huge repercussions for NLP applications. Our parser makes structured templates that make it easier to use effective vectorization methods like BERT, TF-IDF, and word embeddings, which are all based on transformers. You can use these ways to summarize logs and keep an eye on the health of your system [**?** **?**]. For example, entity recognition can help you find important pieces of information, like task IDs or file locations, that are often linked to how the system works by using the HDFS path structures that are included in our templates. Also, our heuristic-based solution has very little processing cost, which makes it good for near-real-time or real-time log analytics in production environments, where scalability is very important.

But there are limits to our method that need to be looked at more. Because the parser uses a fixed similarity threshold (0.4), it is sensitive to changes in parameters. This could impair performance for datasets with varying log characteristics. For example, logs with a lot of semantic drift, where the meaning of log messages changes because of system updates, may need adaptive thresholding or semantic embeddings to stay accurate. The 100% grouping

accuracy is based on a sample of 100 logs, which is a good sign, but full-dataset validation may show that clustering doesn't always catch small differences. Our technique doesn't have the semantic understanding needed to handle logs that are very unstructured or depend on the context, like those with error messages that seem like natural language. This is because it is built on LLMs like LogParser-LLM [**?**] or HELP [**?**]. Resource profiling shows that utilization is stable, however making it work better with dynamic thresholds could make it more stable.

Our work also makes it possible for businesses to use it in the real world. The parser can be used in monitoring tools like Splunk or Elasticsearch that need to handle logs quickly because it works so well. By releasing our solution open-source, we intend to encourage community-driven benchmarking and working together to improve log parsing for distributed systems. These consequences are related to our fourth contribution, which is a detailed assessment.

6 Conclusion and Future Work

We presented an enhanced Drain-like parser in this paper, focusing on the Hadoop Distributed File System (HDFS) dataset, specifically developed for log template mining in distributed systems. Our solution pushes the limits of the field by using a powerful preprocessing pipeline, adaptive clustering with edit distance, and a template merging mechanism that reduces redundancy while keeping the semantic context. Our parser was tested on a 100,000-line HDFS dataset and outperformed baseline Drain implementations by reducing template redundancy by about 10–15% [**?**]. It achieved a compact set of 44 different templates with 100% grouping accuracy on sampled logs. These results show how effectively the parser handles the high level of unpredictability in HDFS logs. This makes it a suitable tool for NLP-based log analysis in big data environments.

The practical importance of our work comes from the fact that it could make monitoring and maintaining distributed computing systems better. Our parser creates organized templates that keep vital context, which means that downstream NLP applications like root cause analysis and performance optimization don't have to do as much work. Real-time analytics can use it since it is scalable and efficient. In production systems, log volumes might reach millions of lines per day. Our methodology also connects traditional heuristic methods with new LLM-based methods, making it possible to create hybrid parsing pipelines that combine semantic understanding and speed.

Future study will focus on various strategies to enhance the functionality of our parser further. To assess generalizability across many system types, we want to expand our study to encompass larger and more heterogeneous datasets, including BGL, Thunderbird, and OpenStack [**?**]. Second, using deep learning techniques, such as self-correction mechanisms from LLMs [**?**] or hierarchical embeddings from HELP [**?**], could make templates more accurate and less likely to drift in meaning, even while the threshold is changing. Lastly, making our technology open to the public will inspire people to work together to make log parsing better for distributed systems and let the community do benchmarking. We want to help build NLP tools for system log analysis that are scalable, effective, and reliable by going along these paths.

References