SPINE: A Scalable Log Parser with Feedback Guidance

Xuheng Wang  
Tsinghua University  
Beijing, China

Xu Zhang  
Liqun Li  
Microsoft Research  
Beijing, China

Shilin He  
Microsoft Research  
Beijing, China

Hongyu Zhang  
The University of Newcastle  
NSW, Australia

Yudong Liu  
Microsoft Research  
Beijing, China

Lingling Zheng  
Microsoft Azure  
Redmond, USA

Yu Kang  
Qingwei Lin  
Microsoft Research  
Beijing, China

Yingnong Dang  
Microsoft Azure  
Redmond, USA

Saravanakumar Rajmohan  
Microsoft 365  
Redmond, USA

Dongmei Zhang  
Microsoft Research  
Beijing, China

ABSTRACT

Log parsing, which extracts log templates and parameters, is a critical prerequisite step for automated log analysis techniques. Though existing log parsers have achieved promising accuracy on public log datasets, they still face many challenges when applied in the industry. Through studying the characteristics of real-world log data and analyzing the limitations of existing log parsers, we identify two problems. Firstly, it is non-trivial to scale a log parser to a vast number of logs, especially in real-world scenarios where the log data is extremely imbalanced. Secondly, existing log parsers overlook the importance of user feedback, which is imperative for parser fine-tuning under the continuous evolution of log data. To overcome the challenges, we propose SPINE, which is a highly scalable log parser with user feedback guidance. Based on our log parser equipped with initial grouping and progressive clustering, we propose a novel log data scheduling algorithm to improve the efficiency of parallelization under the large-scale imbalanced log data. Besides, we introduce user feedback to make the parser fast adapt to the evolving logs. We evaluated SPINE on 16 public log datasets. SPINE achieves more than 0.90 parsing accuracy on average with the highest parsing efficiency, which outperforms the state-of-the-art log parsers. We also evaluated SPINE in the production environment of Microsoft, in which SPINE can parse 30 million logs in less than 8 minutes under 16 executors, achieving near real-time performance. In addition, our evaluations show that SPINE can consistently achieve good accuracy under log evolution with a moderate number of user feedback.

CCS CONCEPTS

• Software and its engineering → Maintaining software.

KEYWORDS

Log parsing, Log Data Analysis, Feedback Guidance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ESEC/FSE ’22, November 14–18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9413-0/22/11...$15.00
https://doi.org/10.1145/3540250.3549176

1 INTRODUCTION

Recent years have witnessed the surging popularity of cloud services, serving millions of customers on a 24x7 basis. Logs, which record important system events and runtime status, provide first-hand information to developers and operators. However, the rapid increase of log data volume and complicated dependency among different log events pose a great challenge to manual log analysis. It thus necessitates research efforts in automated log analysis [13], such as log-based anomaly detection [8, 15, 23, 36], failure diagnosis [1, 14, 21], failure prediction [35], log compression [22, 33], etc. We have found a critical prerequisite step that most automated log analysis approaches depend on: log parsing, which transforms semi-structured console logs into a structured format for downstream tasks to analyze.

Formally, log parsing is defined as extracting log templates and log parameters from raw log messages [38]. As shown in Fig.1, logs are generated from logging statements in programs. A log message usually contains a fixed header including information such as timestamp, verbosity level and component name. The log message body typically consists of two parts: 1) Template – static keywords describing system events, which are explicitly written in logging statements; 2) Parameters – dynamic variables that vary during run time. As the example in Fig. 1 shows, the log header can be distinguished easily through regular expressions due to its relatively static pattern. The body of a log message consists of two parts: the template "Running task "<" in stage "<" (TID "<")", where the "<"s are the placeholders; the log parameters "1.0", "0.0" and "0", which correspond to the placeholders in the template.

A straightforward idea for log parsing is to match log messages with the logging statements in the corresponding source code [34]. However, the diverse logging formats, tools, and programming languages make it not as trivial as it seems. More than that, source code is not always accessible, especially for third-party or proprietary libraries [38]. To overcome this problem, log parsers that solely depend on generated log data have started to thrive in recent years,
such as Drain [12], Logram [4], AEL [18], and LenMa [29]. The core idea behind these approaches is to extract the common parts among log messages as templates and the dynamic parts as parameters.

Although the proposed log parsers have achieved promising accuracy on public log datasets [38], they still face many challenges in real-world applications. Firstly, most existing log parsers can only run in single-thread mode. However, real-world log data is well recognized by its huge data volume (e.g., 200 million logs are generated per hour in a service of Microsoft according to our empirical study), which is thus beyond the processing power of any single core or node, especially for those real-time log analysis scenarios. Log parsing seems to be a task that can be easily parallelized at the first glance. However, the intrinsic imbalance nature of real-world log data, as discussed in our empirical study (Sec. 2.1), can significantly hinder the parallelization efficiency. Secondly, log parsers will be continuously updated as the system that generates logs evolves (Sec. 2.2) [36]. The model parameters also need to be continuously adjusted to adapt to new-coming logs, and otherwise the parsing accuracy would degrade gradually. Unfortunately, model tuning for existing log parsers is both sophisticated and labor-intensive, which requires tedious hyper-parameters adjustment based on a large validation set with manually labeled logs. In summary, these findings motivate us to design a new log parser that is able to scale efficiently across multiple executors, and meanwhile keep user intervention lightweight.

In this paper, we propose a highly-scalable log parser with feedback guidance, namely SPINE. We propose a novel data scheduling algorithm for parsing parallelization, considering the huge log data volume and especially the imbalance characteristic. We design a feedback schema based on pairwise-labeling. Specifically, SPINE selectively picks pairs of logs to query user feedback that can maximize the accuracy gain. We conducted extensive experiments on 16 public datasets from Loghub [16] to evaluate the effectiveness and efficiency of our proposed method. Without user feedback enabled, SPINE already achieves over 0.9 average parsing accuracy, which significantly outperforms the state-of-the-art log parsers. After enabling only a dozen of feedback iterations, the accuracy can be further boosted by up to 30% for those datasets with relatively low accuracy. With regards to efficiency, the single-executor SPINE achieves a higher throughput 50,000 logs per second compared with other log parsers. The throughput can reach about 200,000 logs per second after parallelization. We successfully integrated SPINE in the log processing pipeline for Service X, which is a global-scale service. Our evaluation results confirmed that SPINE can fulfill the requirement for downstream near real-time log analysis tasks. In addition, the parsing accuracy gradual decline caused by logs evolution can be remedied through periodical users’ feedback.

To summarize, our main contributions are as follows:

- We propose an effective and efficient log parser named SPINE, which incorporates users’ feedback for model tuning and a novel data scheduling strategy for parallelization acceleration.
- We evaluated SPINE on 16 public log datasets. The experimental results show that its effectiveness and efficiency outperform other state-of-the-art log parsers, especially after enabling the feedback guidance mechanism and parallelization acceleration.
- Through the evaluation on the production environment, the practicality of SPINE is also confirmed. SPINE achieves the near real-time parsing efficiency under large-scale imbalanced logs, and leverages periodical feedback guidance to make up accuracy drop under evolving logs.

2 EMPIRICAL STUDY AND MOTIVATION

We performed an empirical study to understand the characteristics of log data in the industrial environment. We collected logs from a large-scale production cloud service, namely Service X. Service X is one of many cloud services in Microsoft, which possesses millions of nodes and serves worldwide customers. Another purpose of our empirical study is to understand the limitations of existing log parsers, which motivates our work.

2.1 Huge Volume and Data Imbalance

Tens of billions of logs are generated from the cloud platform per day in the production environment [21]. Only for the single log source in our study, there are about 5 billion log messages generated every day on average, which means about 200 million logs per hour. This huge volume of log data puts tremendous stress on the log parser and the subsequent log analysis tasks. In addition, we found that log data is usually characterized by extreme imbalance. We sampled 0.01% of log data in a 24-hour period. Then we applied a state-of-the-art log parser Drain [12] to get initial parsing results, which are further calibrated manually. Fig. 2 shows the histogram of the number of logs with different templates. The X-axis denotes the template ID and the Y-axis denotes the number of logs (logarithmic scale) corresponding to the template. We can see that most of logs are concentrated on the top few log templates. In fact, the top 10 log templates account for ~ 74.9% of total logs. There is a long tail in log templates distribution as shown in Fig. 2.

A log parser must be able to process massive and imbalanced log data efficiently. Even though the existing parsers have been optimized at the algorithm design stage, a single thread or node is inadequate to fulfill the performance requirement facing an extremely large volume of log data. Therefore, there is an essential need for a log parser that allows the parsing process to scale horizontally across many executors. A naive idea is to randomly split log data into data chunks with equal sizes. Then, we can distribute
the data chunks across multiple executors running an existing log parser. This however will lead to inconsistent parsing results at different executors, i.e., logs with the same template get parsed differently. Allocating logs to executors according to their prefix token hash would mitigate the inconsistency problem. It is because log messages with the same prefix tokens are likely with the same log template so that they would be processed in the same executor, as will be discussed in Sec. 3.6. However, due to the imbalance characteristic in log data, the execution time of a parallel task is determined by the longest sub-task which, in this situation, processes the largest subset of logs with the same prefix tokens. In this work, we first split logs into log groups based on their prefix tokens in Sec. 3.3. Then, we propose a novel data scheduling algorithm to optimize the performance of parsing log data in parallel. More details will be elaborated in Sec. 3.6.

2.2 Evolving Log Data

The evolving problem in log data has been studied in [36], which has a significant influence on downstream tasks like log-based anomaly detection. To understand its impact on the log parsing task, we collected logs ranging from January 2022 to March 2022 (8 weeks) from Service X. We obtain the log templates in the same way as described above. Then, we count the number of newly emerging log templates as time going. The result is shown in Fig. 3. We can see that the number of log templates continuously increases compared with that of the first week due to the Continue Integration/Delivery paradigm [2, 17].

The ever-changing log data motivates retraining the log parser periodically in order to maintain consistently good accuracy. Existing log parsers typically adopt unsupervised methods such as clustering [32], frequent pattern mining [4], longest common subsequence extraction [7] to identify the common parts as the templates. The model parameters (e.g., the number of clusters or the frequency threshold) need to be continuously adjusted, which, if otherwise missed, would lead to an accuracy drop. However, the cost of parameter tuning, which usually requires adequate labeled logs as the validation set and the deep understanding of adopted parsing methods, is either ignored or overlooked before. In this work, we propose a lightweight user feedback schema that does not require user to have deep knowledge of the parsing model. The objective is to minimize the number of feedback queries while maximizing the accuracy gain. More details are discussed in Sec. 3.7.

3 APPROACH

In this section, we will introduce the overall workflow of SPINE and describe its major components. Before that, we summarize the basic requirements for an ideal log parser (or log parsing system) that can be widely-adopted in practice.

- **Accurate**: parsing accuracy is always the first priority for a log parser, as evaluated in many existing studies. It has been confirmed that the parsing accuracy would greatly influence the effectiveness of downstream log analysis applications.
- **Scalable**: in practice, when parsing logs with large data volume, the efficiency of the log parser is very crucial. Standalone log parsers like most existing ones do not flexibly scale up by design and cannot satisfy industrial needs.
- **Evolvable**: the log parser should be evolvable such that it can be updated according to continuous new-coming logs and allows end-users to fine-tune the changing parser with lightweight feedback, no need for the deep understanding of adopted parsing technologies.

3.1 Overview

Our log parsing framework, namely SPINE, is shown in Fig. 4. SPINE has two phases: offline training (red arrow) and online parsing (green arrow). In the offline training phase, SPINE trains a model based on the prior collected training data. Then, the log parser powered by the learned model will be used in the online parsing phase. In the online phase, we assume that the newly generated log data is buffered into batches, and the log data is thus parsed on a batch basis.

SPINE consists of four components, i.e., pre-processing, initial grouping, progressive clustering, and online parsing. We perform token splitting and log cleaning for pre-processing. Then, initial grouping aims to fast split log messages into coarse-grained disjoint log groups. For each log group, we apply a progressive clustering algorithm [3, 37] to further partition similar logs into different log clusters. Logs in one log cluster are likely coming from the same logging statement, and therefore, the common tokens are extracted as the template. During the online parsing phase, SPINE directly applies the learned model (i.e., the clusters and their extracted
templates) to incoming batches of logs. The newly arrived logs will first be pre-processed and assigned into log groups as we do in the training phase. Then we try to put them into existing log clusters based on the similarity between these log messages and all clusters’ templates. We will describe the details of the components through Sec. 3.4 – 3.5.

Recall that one of our design requirements is that SPINE can be scaled up to multiple executors for extremely large-scale log data. A particular challenge with industrial log data is its extreme imbalance, as we have discussed in Sec. 2.1. Therefore, we design a scheduling algorithm to balance the workload at different parser executors, which aims to minimize the execution time. The details of our scheduling algorithm are discussed in Sec. 3.6. For data evolving discussed in Sec. 2.2, SPINE incorporates user feedback to maintain consistent parsing accuracy. Continuously tuning parser parameters under log evolution is a tedious and labor-intensive task. It requires a good knowledge about the parsing model and plenty of labeled logs as a validation set. We thus design a user-friendly and lightweight interaction process that aims to minimize the number of user queries while maximizing the accuracy gain. The details of our feedback mechanism can be found in Sec. 3.7.

3.2 Pre-Processing

Recent work [12, 38] has pointed out that pre-processing of raw log data greatly affects the parsing results. In SPINE, we first split each log message into a series of tokens. The tokenization relies on not only the widely adopted delimiters such as the white-space, but also some special symbols, such as “,” and brackets. We do not naively apply the delimiters. Instead, we consider the common context patterns by using some simple regular expressions. For instance, the “,” in “uid:12345” is clearly a delimiter, while it should not be treated as a delimiter in “Starts at 2022-03-04T12:24:11Z”. As for log cleaning, some commonly-used parameters, such as IP addresses, GUIDs, or file paths, can be masked in advance. Moreover, SPINE also allows users to specify a list of regular expressions according to the characteristics of their own log data.

3.3 Initial Grouping

After pre-processing, initial grouping is used to group similar logs in a coarse-grained fashion. Some existing work [9, 10, 30] directly applies a clustering algorithm to the entire log dataset, which may be infeasible because the large data volume can easily overwhelm general clustering methods. Inspired by Drain [12], the computational cost of the subsequent clustering can be significantly reduced after initial grouping. In our implementation, we incorporate a similar initial grouping based on the prefix tokens. Specifically, log messages with identical first $k$ tokens belong to the same log group. The underlying assumption here is that log messages with the same prefix tokens are likely to share the same log template [12]. More than that, we consider other heuristics that can help refine the grouping results. We can also consider other log properties such as log length, component name and verbosity level. Logs with the same property value would be further categorized into the same log group. It is worth noting that the generated log groups are mutually exclusive with each other. Therefore, they can be independently allocated on multiple executors to parallelly run. More details will be described in Sec. 3.6.

3.4 Progressive Clustering

So far, log messages have been roughly grouped into mutually exclusive log groups. Logs in one log group likely belong to more than one log template. We need to further divide logs into more fine-grained log clusters to extract their log templates. Considering the enormously large volume of log data, it is typically unacceptable to apply sophisticated clustering algorithms. As a result, we borrow the idea from an efficient and progressive clustering approach [3, 37] that iteratively partitions log messages into binary subgroups. Without restrictions, the partitioning can ultimately divide a group of logs into clusters where each cluster contains only a single log. Therefore, knowing when to stop is critical. In the following of this section, we will introduce the progressive clustering algorithm, and
then our proposed stop criterion based on token saturation will be also presented.

3.4.1 Progressive Clustering. In each log group, the log messages are first transformed into one-hot encoding vectors. After that, we leverage a meta clustering method (e.g., K-Means [27] or Gaussian Mixture Clustering [5]) on the log vectors to cluster them into two subgroups. The same process continues on each subgroup iteratively until a stop criterion is satisfied. Finally, we can obtain a binary clustering tree [37] and we treat the leaf nodes as the logs, as shown in Fig.5. The time complexity of progressive clustering approximately $O(p+n)$ [27], where $n$ is the number of logs and $p$ denotes the number of partitions and $p \ll n$ in general. The high efficiency makes it suitable for our situation with large data volume.

3.4.2 Stop Criterion: Token Saturation. At each iteration, the only decision we need to make is whether to stop dividing one node into two child nodes. The stop criterion plays a key role in the clustering tree growth. SPINE adopts the token saturation as the stop criterion. We first define the saturated tokens as the tokens that appear in all log messages of a group. As the example in Fig.5 shows, the tokens in red color are saturated tokens in the corresponding tree nodes. The saturation of a log message $l$ is defined as the ratio of saturated tokens among all tokens in this log, i.e., $S_l = \frac{|S_l|}{|l|}$, where $S_l$ denotes the saturated tokens. Given $S_l$ for any single log, we can then calculate the average log token saturation $\bar{S}$ for any node on the clustering tree. We use the average token saturation to guide the progressive clustering process. The average token saturation $\bar{S}$ increases monotonously with the depth of the tree nodes. Without restriction, the leaf nodes would only contain identical logs (i.e., $\bar{S} = 1$), leading to over-clustered logs. Conversely, a cluster of logs generated by different logging statements results in a lower average token saturation value because it is unlikely that different log templates share many common tokens. We use a predefined token saturation threshold $T$ to cluster logs under a proper granularity. If $\bar{S} \geq T$, this leaf node would no longer be partitioned anymore.

Nevertheless, it is not practically feasible to choose an optimal threshold for an ever-changing model under evolving log data. We notice that this is a common problem, which is overlooked by previous work. Existing log parsers [4, 12, 18, 29] are typically adjusted based on sophisticated hyper-parameters. The parameter tuning requires a deep understanding of the underlying parsing mechanisms and heavy labeling costs for a validation dataset, which are not practical with continuously evolving logs. Therefore, we are required to incorporate lightweight user feedback to refine the model from time to time. More details will be introduced in Sec. 3.7.

3.4.3 Template Extraction. After clustering, the logs in a leaf node are likely to share the same log template. We assign a unique cluster ID to each leaf node. The saturated tokens can then be directly extracted as the log template. Other tokens are treated as parameters.

3.5 Online Parsing

In the online parsing phase, logs are processed on a batch basis. Each newly arrived log within the current batch will first be preprocessed and assigned into an initial group as discussed in Sec. 3.2-3.3. After that, we try to assign it to an existing log cluster based on a similarity score. Recall that each log cluster has a unique extracted template. The similarity score is defined as the ratio of the number of common tokens over the number of template tokens, i.e., $\text{sim}(l, t) = \frac{|l \cap t|}{|t|}$ where $l$ and $t$ represent the token set of a log and a template, respectively. We calculate the similarity scores between the log and the templates of all clusters inside the log group. The log cluster with the highest similarity score, denoted as $\text{sim}_{\text{max}}$, is selected and this log will be assigned to it. The template of a log cluster can change after adding new logs. The saturated tokens may become unsaturated as they may not appear in the newly added logs. In addition, SPINE also clusters all unmatched logs to form new log clusters at the end of each log batch. We mark a log as unmatched if there is no cluster with a similarity score higher than a loose threshold (e.g., 20% in our implementation). As a result, the model gets updated after parsing each batch of logs.

3.6 Data Scheduling for Parallelization

In previous sections, we have elaborated the building blocks of SPINE for offline training and online parsing. As discussed in Sec. 2.1, scalability is a critical property facing the vast amounts of logs. In this section, we will first discuss the parallelization challenge under imbalanced logs and then propose a log data scheduling algorithm to overcome the challenge.

3.6.1 Challenge. Log groups generated by the initial grouping (Sec. 3.3) are disjoint. As a result, logs in different groups can be processed in a completely parallel manner. However, due to the imbalance issue as introduced in Sec. 2.1, the size of one log group can be several orders of magnitude larger than another log group. The execution time of clustering is proportional to the log volume.

Figure 5: Progressive clustering with token saturation based stop criterion. Red color tokens denote the saturated tokens. The blue dashed boxes denote the feedback query process.
Consequently, the parsing time of some groups could be significantly longer than other groups. The parsing time for a batch of logs is thus determined by the largest log group, which implies a sub-optimal performance under log imbalance.

3.6.2 Log Data Scheduling. The challenge motivates us to design a scheduling algorithm to distribute log parsing workload more evenly across multiple executors towards optimal performance. We assume that \( m \) log groups \( g_i \in G \) are distributed to \( n \) executors \( e_i \in E \). The goal is to divide the log parsing workload as evenly as possible. Ideally, each executor would process an equal number of logs, denoted as \( \text{avg} = \frac{\sum g_i}{n} \), where \( |g_i| \) is the size of log group \( g_i \). To achieve this goal, for those log groups whose sizes are above average, we tend to split them into smaller subsets with approximately the average number of logs. In contrast, for the log groups with sizes below average, we pack them together so that the size of the resulting supersets approximate the average. The latter is essentially a classic optimal job scheduling problem [24] and a greedy bin-packing algorithm BestFit [6] can be applied. The basic idea of the BestFit bin-packing algorithm is iteratively assigning the largest log group in the candidate list to the executor with the lightest workload.

The pseudo-code is presented in Algorithm 1. We aim to allocate \( m \) log groups \( G \) to \( n \) executors \( E \). We first sort all log groups according to their sizes in descending order (line 1). Then, we split the log groups whose sizes are above the average. We do the splitting one by one in a for loop. The average number of logs, denoted as \( \text{avg} \), is calculated in line 3, which is based on the number of logs to be scheduled \( l_{left} \) and the number of unused executors \( |E_{left}| \). For each log group, whose size exceeds \( \text{avg} \), we need to split it uniformly into \( h \) subsets and assign them to the same number of executors (line 5 - line 7). After that, we update the executors \( E_{left} \), the left log groups \( G_{left} \) (line 8), and recalculate \( \text{avg} \) (line 9). The for loop breaks after all above-the-\( \text{avg} \) log groups are scheduled (line 10 ~ line 11). We directly invoke the BestFit [6] algorithm to pack all subsequent log groups into \( |E_{left}| \) supersets and assign them to the rest of executors (line 12 ~ line 13).

In Algorithm 1, logs belonging to one log group might be parsed on different executors when the group size is above average. Therefore, the models for the same log group are updated separately on different executors as discussed in Sec. 3.5. To merge two different models generated from both executors, if a log cluster (with a unique ID) exhibits in both models, we merge the two clusters into one cluster. The log template of the resulting cluster is the intersection of the two templates mentioned above. It is possible that some logs are unable to be assigned to any existing log clusters, especially due to log evolution shown in Fig. 3. At the end of each batch, SPINE collects all unmatched logs from different executors and performs the progressive clustering (details explained in Sec. 3.4) to form new log clusters. Finally, the updated model is broadcast to all executors again before processing the next batch of logs.

### 3.7 Incorporating User Feedback

As introduced in Sec. 2.2, user feedback is desired for maintaining consistent accuracy under evolving log data. In this work, we propose a novel feedback scheme based on the progressive clustering algorithm. Our top considerations are user-friendliness and efficiency. Users should be able to provide their feedback without understanding the model details. Moreover, the accuracy of the log parser can be significantly improved with as little feedback as possible.

3.7.1 Feedback Query. The model construction is essentially the process of building the binary clustering tree for each log group (see Sec. 3.4). Each leaf node represents a group of logs sharing the same template. As a result, the key factor affecting the accuracy of the model is whether a leaf node should be split into two children. In Sec. 3.4.2, we introduced a static saturation threshold as the stop criterion, which is inadequate for real-world situations because the evolving logs would make the initial model parameters not applicable anymore and cause the incorrect parsing results. We thus introduce a user feedback scheme to enhance the node splitting.

The feedback process, which involves multiple rounds of queries, is triggered during offline model training or after parsing each log batch. In each round, SPINE recommends a pair of logs from the same log cluster (i.e., a tree leaf node as shown in Fig. 5) to users under certain conditions. The user gives feedback about whether this pair of logs share the same template. The user feedback helps SPINE to determine whether to split the corresponding cluster into two child clusters. Then, SPINE selects the next log cluster that requires human intervention to make the splitting decision. This feedback process continues until the interaction rounds have reached the upper bound specified by users or there are no more log clusters worthy of querying user feedback. Fig. 5 shows an example of a feedback query, where one leaf node splits into two tentative clusters. SPINE selects two representative logs from each tentative cluster to query user feedback. The user clicks a button to answer “Yes” or “No”. In this example, the two logs are with different templates, one about “Running task...” and the other about “Finished task...”. So the feedback is “No” in this case. With the user feedback, SPINE...
We conducted experiments based on 16 public benchmark datasets. We calculate the saturation gain for each leaf node. A high saturation gain implies that splitting the current leaf nodes has a high potential to improve the accuracy of the model. As a result, we tend to query user feedback for leaf nodes with high saturation gain values.

4 EXPERIMENTS

We conducted extensive experiments on public log datasets to answer the following research questions:

- **RQ1**: How effective and efficient is SPINE without feedback guidance and parallelization acceleration?
- **RQ2**: How effective is SPINE with feedback guidance?
- **RQ3**: How efficient is SPINE with parallelization acceleration?

4.1 Implementation and Environment

In our experiments, we set up a Virtual Machine (VM) with 64 Intel(R) Xeon(R) Platinum 8370C CPU @ 2.80GHz processors and 256GB RAM. The operating system is Ubuntu-20.04. We implemented SPINE in Python 3.8. We implemented our four versions of our proposed log parser, i.e., 1) **SPINE-base**: the basic version of SPINE without parallelization and feedback guidance for RQ1; 2) **SPINE-feedback**: SPINE-base with feedback guidance enabled for RQ2; 3) **SPINE-parallel**: a parallel version of SPINE equipped with our proposed log data scheduling for RQ3; 4) **SPINE-complete**: the complete version of SPINE for production evaluation in Sec.5.

4.2 Datasets and Metrics

We conducted experiments based on 16 public benchmark datasets from LogPai [19], which consists of various logs from different systems spanning across distributed systems, supercomputers, operating systems, mobile systems, server applications, and standalone software. In each log dataset, every log message is labeled with a ground truth log template and a corresponding unique template ID.

In our evaluation, we follow a widely adopted metric, namely Parsing Accuracy [4, 20, 38], to measure the effectiveness of various log parsers. Parsing accuracy is defined as the ratio of correctly parsed log messages over the total number of log messages. Besides, we measure the execution time in seconds and the throughput (the processed number of logs per second) when we compare SPINE with other parsers in terms of efficiency.

4.3 RQ1: How Effective and Efficient Is SPINE without Feedback Guidance and Parallelization Acceleration (SPINE-base)?

For RQ1, we evaluated SPINE-base from parsing accuracy and execution time, respectively. We compared SPINE-base with the 5 state-of-the-art log parsers on the 16 public log datasets.

4.3.1 Parsing Accuracy. The experimental results are shown in Table 1. We listed the average parsing accuracy of each parsing method and summarized the best accuracy for every log dataset. In particular, we mark accuracy values greater than 0.90 in boldface and the best accuracy is also highlighted with an “∗” for each dataset. From the table, we can see that SPINE significantly outperforms AEL [18], LenMa [29], Spell [7], and IPLoM [25]. Among the baselines, Drain [12] achieves the best performance. However, SPINE achieves better results than Drain in 8 datasets and achieves comparable accuracy in 5 datasets. We note that the parsing accuracy is relatively low for all log parsers under certain log datasets, such as Linux, Mac, and OpenStack. The reason is that there are a considerable number of logs generated from similar templates in the above three datasets (e.g., “Took <*> seconds to destroy the instance on the hypervisor.” and “<*> seconds to spawn the instance on the hypervisor.” in OpenStack log dataset). The model hyper-parameters, e.g., the saturation threshold $T$ in SPINE-base, should be adjusted aggressively in order to correctly distinguish them. However, it leads to the incorrect parsing results for other templates, as they will be over-clustered. It shows that manually tuning the parsing models is not systematic and can result in poor parsing accuracy. This motivates us to incorporate user feedback as discussed in Sec. 3.7.

4.3.2 Execution Time. To measure the efficiency of our proposed method, we compared the execution time of SPINE with 5 baseline log parsers on three large log datasets, namely HDFS, BGL, and Spark. We measured the online parsing time of each parser under the different scales of log data, which are randomly sampled from the original datasets. As described in our experiment settings, we used only a single executor for the sake of fairness. In order for robustness, we run each log parser for five times and calculated the average execution time in Fig. 6.

Lenma and AEL failed to finish parsing on Spark dataset (about 25 million of logs) within our time limitation (6 hours), and neither did Lenma on the 4.5-million BGL dataset, so their corresponding results are not shown in the figure. From the figure, we can conclude that SPINE-base outperforms all existing log parsers, especially when the data size is large. The execution time increases linearly with the size of the logs on these datasets. For example in HDFS log dataset, SPINE-base only spent ~4 minutes to parse 10 million logs. The high performance of SPINE-base is due to two reasons.
4.4 RQ2: How Effective Is SPINE with Feedback Guidance (SPINE-feedback)?

We applied SPINE-feedback on the three log datasets with unsatisfactory parsing accuracy under all log parsers, i.e., Linux, Mac, and OpenStack. We first trained an initial parsing model for each dataset. Then, we queried user feedback for the log clusters with the highest saturation gain scores. For this experiment, we used simulation instead of real user intervention since we know the ground truth for public log datasets. We thus can directly check whether the recommended pairs of logs belong to the same template. We also implemented two baseline recommendation strategies to show the effectiveness of our proposed feedback mechanism. The first baseline is a random strategy (denoted as Random), where pairs of logs are sampled from random log clusters to recommend to the user. In the second baseline (denoted as LNS), we consider the leaf node size. We prefer to select log pairs from the larger log clusters.

4.5 RQ3: How Efficient Is SPINE with Parallelization Acceleration (SPINE-parallel)?

We evaluated the effectiveness of our proposed log data scheduling method on the three largest datasets, i.e., HDFS, BGL, and Spark. In order to simulate the log volume in the real-world, we expanded all three datasets to 30 million with uniform replication. We run SPINE-parallel with different numbers of executors and recorded the throughput (the processed number of logs per second) for each dataset. We ran five times under each setting and took the average as the final results, which are shown in Fig. 8. We can see that our proposed log scheduling method can significantly improve the parsing efficiency. For example, SPINE-parallel can achieve about 5× throughput improvement for the BGL dataset with 16 executors, about 225,000 logs per second. Moreover, we compared SPINE-parallel with a naive BestFit scheduling scheme where large log groups would not be split into smaller subsets. Our proposed log data scheduling algorithm achieves a higher speed-up ratio with more executors (16 or 32). When the number of executors is
fewer than that of log groups, our proposed method degrades to
the BestFit strategy because all log groups can be directly packed
into a few supersets with similar sizes. Therefore, there is no need
to split large log groups for re-balance. From the figure, we also
can see the two scheduling schemes achieve similar performance
under 4 or 8 executors.

5 INDUSTRIAL EVALUATION

We have successfully integrated SPINE into the log processing
pipeline for Service X. SPINE has been running for 3 months. We
carried out an online evaluation in the production pipeline, aiming
to validate its practicability in the industry.

Logs in Service X evolve as time goes, which has been discussed
in Sec.2.2. We activated the feedback process every month. The
recommended pairs of logs are presented to the engineers from
Service X via a dashboard. The engineers clicked a “Yes” or “No”
checkbox to determine whether the given pair of logs belong to
the same template. In each feedback process, we only query user
feedback for 20 pairs of logs, which is a negligible workload. In order
to interpret the parsing accuracy, we randomly sampled 2000 logs on
every Monday of the past 12 weeks and manually labeled the ground
We have identified the following major threats to validity:

- Data quality: in this paper, we used public log datasets for evaluation. The ground truth templates of all log messages are provided within the datasets. Although these datasets are the standard benchmarks used by many related work [4, 12, 38], they may also contain a small proportion of mistakes.
- Correctness of feedback: in SPINE, we rely on users to provide correct feedback about whether a given pair of logs belong to the same template. We assume that the feedback information is always correct. However, in reality, the quality of feedback may vary, i.e., users may provide wrong feedback, which can affect the parsing accuracy.
- Representativeness of production log data: in Sec. 5, we evaluated the parsing accuracy of SPINE based on a small set of sampled production log data, which may not be representative. This is because it is almost impossible for us to manually label the full amount of logs. Therefore, the sampled logs may not cover all log templates in production.

7 RELATED WORK

Log parsing has become an active research topic in the last few years [12, 25, 30, 38]. Some existing log parsers extract the frequent part from logs as the log template. The infrequent part can be regarded as the log parameters. Some approaches, such as SLCT [31], identified the frequent log tokens across the entire log data as the template. Similar to this method, LFA [28] considered the token frequency distribution. LogCluster [32] and Logram [4] further used the token position and n-gram information, respectively, not merely relying on the single log token.

There is another body of clustering-based log parsers proposed in recent years, including LIKE [9], LogSig [30], LogMine [10], SHISO [26], and LenMa [29]. Specifically, these approaches leveraged various clustering algorithms to group similar logs and treated the logs under the same cluster belonging to the same template. It is worth noting that SHISO and LenMa are both online log parsing methods, which means they are capable of parsing log messages one by one, not need to wait for a batch of logs accumulated [38].

Besides the above-mentioned parsers, AEL [18] clustered logs into multiple groups by the occurrences between constant tokens and variable tokens. IPLoM [25] proposed an iterative partitioning strategy. It partitions log messages into groups by message length, token position and mapping relation. Spell [7] utilized the longest common subsequence algorithm to parse logs. Drain [12] built a fixed-depth tree structure to divide logs into different groups according to the log length and head tokens.

Some existing log parsers, such as Drain [7, 12, 18, 25, 29], have shown their effectiveness and high efficiency on public benchmark datasets. However, as stated in Sec. 2, we still encountered some challenges when applying these methods to real-world scenarios. Firstly, most of these existing work cannot be paralleled in a straightforward way. It is because these algorithms are not initially designed for scalability. POP [11], as the distributed enhanced version of Drain, tried to make up for this deficiency. It transferred Drain to Spark platform but did not optimize the log parsing algorithm itself to adapt to the imbalanced log data. In addition, most of log parsers did not involve human in the loop. Engineers often identify some evolving logs are not parsed correctly but they can hardly teach the parsing model to rectify these mistakes.

8 CONCLUSION

In this paper, we propose a highly scalable log parser with feedback guidance, namely SPINE. We design a new log parsing model equipped with initial grouping and progressive clustering, based on which a novel log data scheduling algorithm is proposed to improve the parallelization efficiency under the large-scale imbalanced log data. Besides, SPINE incorporates user feedback to achieve constantly good parsing accuracy for evolving log data. We evaluated SPINE on 16 public log datasets. The experiment results show that the proposed approach achieves over 0.90 parsing accuracy on average with high parsing efficiency, outperforming the state-of-the-art log parsers. We deployed SPINE in the production environment of Microsoft. SPINE achieves near real-time performance under large-scale imbalanced log data, and can also maintain a stable accuracy during log evolution with a little user feedback.
REFERENCES


