

Article

Polo: Adaptive Trie-Based Log Parser for Anomaly Detection

Yuezhou Zhou and Yuxin Su * 

School of Software Engineering, Sun Yat-sen University, Zhuhai 528406, China; zhouyhz27@mail2.sysu.edu.cn

* Correspondence: suyx35@mail.sysu.edu.cn

Abstract: Automated log parsing is essential for many log-mining applications, as logs provide a vast range of information on events and variations within an operating system or software at runtime. Over the years, various methods have been proposed for log parsing. With improved log-parsing methods, log-mining applications can gain deeper insights into system behaviors and identify anomalies or failures promptly. However, current log parsers still face limitations, such as insufficient parsing of log templates and a lack of parallelism, as well as inaccurate log template parsing. To overcome these limitations, we have designed Polo, a parser that leverages a prefix forest composed of ternary search trees to mine templates from logs. We then conducted extensive experiments to evaluate the accuracy of Polo on nine representative system logs, achieving an average accuracy of 0.987. It is 9.93% to 40.95% faster than the state-of-the-art parsing methods. Furthermore, we evaluated our approach on a downstream log analysis task, specifically anomaly detection. The experimental results demonstrated that, in terms of F1-score, our parser outperformed Deeplog, LogAnomaly, CNN, and LogRobust by 11.5%, 4%, 1%, and 19.1%, respectively, exhibiting a promising recall score of 0.971. These results indicate the effectiveness of Polo for anomaly detection.

Keywords: log analysis; parsing; trie; log anomaly; software engineering

MSC: 68N30



Citation: Zhou, Y.; Su, Y. Polo: Adaptive Trie-Based Log Parser for Anomaly Detection. *Mathematics* **2023**, *11*, 4797. <https://doi.org/10.3390/math11234797>

Academic Editor: Cheng-Chi Lee

Received: 14 October 2023

Revised: 22 November 2023

Accepted: 23 November 2023

Published: 28 November 2023



Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Online computer systems are susceptible to various malicious attacks in cyberspace. The timely detection of anomalous events from these systems is a crucial first step in protecting them from attacks or malfunctions [1]. Software-intensive systems often record runtime information by printing console logs. A large and complex system can generate a massive amount of logs, which can be invaluable for troubleshooting purposes [2]. System logs, which capture detailed information about computational events generated by computer systems, play a vital role in modern anomaly detection task.

Software-intensive systems often generate console logs to record runtime information [3]. These logs are crucial for troubleshooting purposes in large and complex systems. Log-based anomaly detection aims to identify abnormal system behaviors by analyzing the run-time information recorded in logs [4]. It is different from computer vision [5–7] and digital time series [8]. While several approaches have been proposed for this purpose, they all share a common initial step called log parsing. This step is essential because automated log analysis requires structured input logs, while original logs consist of semi-structured text generated by logging statements [9]. Log parsing involves recognizing different fields (e.g., verbosity levels, timestamps) and extracting the message content, which is represented as structured event templates with corresponding parameters. By performing log parsing, the original logs can be transformed into structured input logs suitable for efficient anomaly detection in software-intensive systems [10].

However, log data is typically in an unstructured format, which makes it challenging to understand the system's internal state and perform effective monitoring, administration, and troubleshooting [11]. Log parsing is a critical pre-processing step that bridges

this gap by converting the original unstructured log messages into structured event templates [12,13]. Log parsing methods extract log structures by analyzing log data and identifying unique features. AEL [14] divides log messages into groups by comparing constant and variable tags. IPLoM [15] partitions log messages iteratively based on message length, token position, and mapping relationships. Spell [16] parses system logs using the longest common subsequence method. Drain [17] represents log messages with a fixed-depth tree structure and efficiently extracts common templates. SemParser [18] extracts semantics from log messages according to the contextual knowledge base of the log statements.

Unfortunately, most existing log-parsing methods cannot be parallelized to achieve faster parsing speeds, and the extracted templates are not precise enough, which hinders the accuracy of downstream tasks such as anomaly detection. We conduct an extensive study to investigate the performance of Polo on several system logs from two perspectives: The experimental results demonstrate that our approach can generate more accurate log templates, which achieve an average accuracy of 0.987. Furthermore, we evaluated our approach on a downstream log analysis task, specifically anomaly detection. The experimental results demonstrated that, in terms of F1-score, our parser outperformed Deeplog, LogAnomaly, CNN, and LogRobust by 11.5%, 4%, 1%, and 19.1%, respectively, exhibiting a promising recall score of 0.971. These powerful results reveal the superiority of Polo and emphasize the importance of speed and accuracy in log analytics, especially when the software systems we handle are more complicated than ever before.

Polo is instrumental in cloud system monitoring, involving offline training and online serving. Online, Kafka facilitates real-time log analytics, with Polo conducting anomaly detection using Apache Flink. Visualization via Prometheus aids engineers in confirming anomalies or addressing false positives. In the offline stage, Polo undergoes (re)training using raw logs stored in Apache HDFS, triggered by engineers based on monitoring panel observations.

In summary, the contribution of this paper is threefold:

- To the best of our knowledge, Polo is the first parser capable of parallelizing log parsing and generating more accurate log templates specifically designed for anomaly detection.
- We evaluated the accuracy of Polo on nine system logs, and the results demonstrated that our framework is capable of effectively summarizing templates from logs.
- We also applied Polo to the task of anomaly detection, and the promising results revealed that this parsing method is more suitable for the field of log anomaly detection.

2. Related Work

2.1. Log Parsing

As shown in Figure 1, log parsing refers to the process of extracting templates from raw log data. Log parsing errors can lead to extra log events and wrong log templates [19]. When it comes to log parsing, there are several methods available to make sense of the data. One popular approach is frequent pattern mining [10], which involves identifying patterns that occur frequently in the log files. This can provide valuable insights into system behavior and help identify potential performance issues.

Another approach is using heuristics, which relies on expert domain knowledge to understand the meaning of different log entries based on context and other factors. This approach can quickly identify problem areas but may introduce errors and lacks accuracy at times.

Clustering is another method used to analyze log files. It groups related log entries based on their similarity, helping to uncover patterns and trends that may not be immediately apparent otherwise.

Each method has its strengths and weaknesses, and the best approach depends on the specific project goals. Frequent pattern-based methods extract common patterns from log data and treat them as fixed templates. These methods often use offline techniques

involving traversing the log data multiple times, building frequent item sets, grouping log messages into clusters, and extracting templates from each cluster.

Raw Logs

```
081111 072949 23644 INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block
blk_1508705816376906895 terminating

081111 072956 23082 INFO dfs.DataNode$PacketResponder: Received block blk_-6441360440068187612 of
size 67108864 from /10.250.10.100

081111 073005 23505 INFO dfs.DataNode$PacketResponder: PacketResponder 1 for block
blk_2128110944185709079 terminating
```

Log Templates

```
PacketResponder <*> for block <*> terminating
Received block <*> of size <*> from <*>
```

Figure 1. Example of Log Parsing.

Heuristics-based methods observe log data and extract unique features to process the data further. For example, AEL [14] divides log messages into groups by comparing constant and variable tags, while IPLoM [15] uses an iterative partitioning strategy based on message length, token position, and mapping relationships. Drain [17] represents log messages using a fixed-depth tree structure and efficiently extracts common templates.

Heuristics-based approaches leverage the characteristics of logs and have demonstrated superior performance in terms of accuracy and time efficiency compared to other techniques [10]. Our proposed solution extends the Drain method with a modified three-tree structure to improve template extraction accuracy.

2.2. Anomaly Detection

Anomaly detection in log data involves identifying abnormal patterns that deviate from expected behaviors in software systems. Existing methods for anomaly detection based on log data can be classified into three categories: graph model-based [20–22], probability analysis-based [23], and machine learning-based detection methods [24].

Deep learning models, especially RNN-based models, have been extensively used for log-based anomaly detection due to their effectiveness in capturing intricate patterns and dependencies within log sequences.

2.2.1. Log Partition

Since logs are text messages, they need to be transformed into numerical features for machine learning algorithms to understand. To achieve this, log messages are first represented as log templates using log parsers. Then, logs are grouped into different sets, where each set represents a log sequence, often utilizing timestamps and identifiers (e.g., task/job/session IDs) of the logs. Log partition employs three types of window [25]:

- **Fixed window:** Fixed window with pre-defined partition sizes that represent time intervals for splitting logs ordered by timestamps. In this case, there is no overlap between consecutive fixed partitions.
- **Sliding window:** Sliding window has two parameters, namely window size and stride. The stride indicates the distance the time window slides forward along the time axis to generate log partitions. Typically, the stride is smaller than the partition size, leading to overlapping between different sliding windows. Therefore, the sliding window strategy generates more log sequences compared to the fixed partition strategy, depending on the partition size and stride.
- **Session window:** Unlike fixed/sliding windows, session windows group logs based on identifiers. Identifiers are used to group logs into the same execution path. For example, HDFS logs use block_id to record the execution path.

2.2.2. Feature Extraction

Extracting log features is the foundation of anomaly detection. Generally, researchers select features from system logs, including log templates, event occurrence counts, event indices, and log variables, and encode them using one-hot encoding or other weighted methods. Additionally, an increasing number of works are beginning to apply natural language processing (NLP) methods to log pre-processing, where each element in the sequence can simply be an index of the log event or a more complex feature, such as a log embedding vector. The aim is to learn the semantics of logs to make more intelligent decisions. Specifically, the words in log events are first represented by word vectors learned using word2vec algorithms, such as FastText [26] and GloVe [27]. DeepLog [28] assigns each log event with an index, and then generates a sequential vector for each log window. Quantitative vectors are similar to log count vectors, which are used to hold the occurrence of each log event in a log window. LogAnomaly [29] leverages both sequential and quantitative vectors to detect anomalies different from them, and semantic vectors are acquired from language models to represent the semantic meaning of log events. Each log window is converted into a set of semantic vectors for the detection models. For instance, LogRobust [30] adopts a pre-trained FastText model to compute the semantic vectors of log events.

2.2.3. Deep-Learning Model

After log partition and feature extraction, logs are represented in the different formats required by deep-learning models. These methods are briefly described as follows:

- **DeepLog:** Du et al. [28] introduced DeepLog, which is the pioneering work that utilizes LSTM for log anomaly detection. Notably, it learns log patterns based on the sequential relationships among log events, where each log message is represented by the index of its corresponding log event. Additionally, DeepLog is the first approach to detect anomalies in a forecasting-based manner, a technique that has been widely adopted in subsequent studies.
- **LogAnomaly:** Meng et al. [29] proposed LogAnomaly, which uses log count vectors as inputs to train an LSTM model. They also demonstrated that existing word2vec models did not distinguish well between synonyms and antonyms. Therefore, they trained a word embedding model to explicitly consider the information of synonyms and antonyms. Like DeepLog, a forecasting-based detection model is designed to predict the next log event, and if the examined log event violates the prediction results, it will be marked as an anomaly.
- **LogRobust:** Zhang et al. [30] observed that many existing studies on log anomaly detection do not achieve the promised performance in practical scenarios. Specifically, most of these studies make a closed-world assumption, assuming that (1) the log data remains stable over time and (2) the training and testing data share the same set of distinct log events. However, log data often contain previously unseen instances due to changes in logging statements and noise in log processing. To address the issue of log instability, they proposed LogRobust, which leverages off-the-shelf word vectors to extract the semantic information of log events. This approach is one of the earliest studies to consider the semantics of logs, as also done by Meng et al.
- **CNN.** Lu et al. [31] were the first to investigate the feasibility of using CNN for log-based anomaly detection. Their approach involved constructing log event sequences through identifier-based partitioning, with padding or truncation applied to ensure consistent sequence lengths. To enable convolution calculations, which require a two-dimensional feature input, the authors introduced an embedding method called logkey2vec. They began by creating a trainable matrix with dimensions equal to the number of distinct log events $\times$ embedded size (a tunable hyperparameter). Next, they applied different convolutional layers with various shape settings, concatenating their outputs and passing them through a fully-connected layer to generate the prediction result.

3. Design of Polo

3.1. Overview

The process of Polo has four main steps to finish, which include the following: (1) Preprocessing; (2) Search template by descending the TrieTree; (3) Node Update; and (4) Template Generation.

The process of log parsing involves matching log data with different ternary search trees. The trie-forest locks onto a ternary search tree based on the starting word of the log. As seen in Algorithm 1, the ternary search tree then moves to the corresponding node based on the field information in the log data. When a leaf node is reached, it represents a template. If the searched leaf node does not match, the ternary search tree creates the corresponding node and adds it to the ternary search tree.

Algorithm 1 Parsing log message

```

1:  $message \leftarrow preprocess(logmessage)$ 
2: Initialize the root node:  $curNode \leftarrow forest.root$ 
3:  $template \leftarrow searchtemplate(curnode, message)$ 
4: if  $Similarity(message, template) > \delta$  then
5:   return  $curnode$ 
6: if  $Similarity(message, template) < \delta$  then
7:   Node Update( $message, curnode$ )

```

3.2. Trie-Forest

3.2.1. Step 1: Preprocessing

According to our previous empirical study on existing log-parsing methods, preprocessing log messages can improve parsing accuracy [32]. Users can provide regular expressions (regexes) representing commonly used variables (e.g., IP address, file paths, URLs). Before any parsing operation, Polo will run all those regexes and remove the tokens matched from the raw log message by this regular expression. This step is simple, but it is the best way to use domain knowledge to improve the quality of parsing.

3.2.2. Step 2: Search Template by Descending the TrieTree

In Algorithm 2, we extract log templates by constructing a trie-forest, which takes into account both log structure and parallel processing to aid in further log anomaly detection. By grouping logs with similar structures, efficient parallel processing can be achieved in large log datasets.

Algorithm 2 Search Template

INPUT: message, curnode

OUTPUT: template

```

1:  $node \leftarrow curnode$ 
2: for word in message do
3:   if word < node.label then
4:      $node \leftarrow node.leftnode$ 
5:   else if word > node.label then
6:      $node \leftarrow node.rightnode$ 
7:   else if word = node.label then
8:      $node \leftarrow node.middlenode$ 
9: if node.isLeaf then
10:   $template \leftarrow node.template$ 
11:   $sim \leftarrow Similarity(message, template)$ 
12:  if  $sim > \sigma$  then
13:    return  $template$ 
14:  if  $sim < \sigma$  then
15:    Node Update( $message, curnode$ )

```

The process begins by examining the root node of the trie-forest. Starting from this root node, Polo evaluates the first word of the input message to determine which ternary search tree to search. If the first word matches the label of the root node in a ternary search tree, Polo traverses that specific tree.

For each word in the log message, Polo compares it with the label of the current node. If there is a match, it moves to the middle child; if the word is greater, it checks the right child, and if it is smaller, it checks the left child. This process continues until a leaf node is reached. The log parser then employs the similarity function defined by Formula (1) to find the most similar template in the group.

Formula (1) [17] calculates the similarity between log A and template B. If the calculated similarity exceeds a predefined threshold, the parser interprets the log data as an instance matching the template. It then identifies variable parts in the log data. Providing a more detailed and clearer explanation would involve specifying the exact details of Formula (1) and the threshold value used in this process.

$$\text{Similarity}(A, B) = \frac{\sum \varphi(a_i, b_i)}{|A| + |B|} \quad (1)$$

where a_i and b_i represent the i -th token of the log A and log B ; $|A|$ and $|B|$ are the log message lengths of log A and log B.

3.2.3. Step 3: Node Updating

The detailed process is described in Algorithm 3. When the leaf nodes from the search do not match, the parser constructs the corresponding node and adds it to the ternary search tree. Specifically, Polo compares the labels in the same position as the log message and log event. If the labels match, there is no modification to that label position. The parse tree is then updated with the new log group.

Algorithm 3 Adding new log to the TrieTree

INPUT: message, curnode

OUTPUT: curnode

```

1: node ← rootnode
2: for word in message do
3:   if word < node.label then
4:     ADD(node.leftnode, word)
5:     node ← node.leftnode
6:   else if word > node.label then
7:     ADD(node.rightnode, word)
8:     node ← node.rightnode
9:   else
10:    ADD(node.middlenode, word)
11:    node ← node.middlenode
12: return node

```

Starting from the root node, each character of the message is examined and compared with the current node's label. If the character is smaller than the label, the left-child path is followed. If the current node lacks a left child, a new node is created and attached as the left child. If a left child already exists, the insertion process continues in this subtree, recursively examining its sub-nodes until a suitable position to insert the next character is found.

Similarly, if the word is larger than the label, Polo follows the right-child path. If the current node lacks a right child, a new node is created and attached as the right subtree. If a right child already exists, the insertion process continues in this child by recursively examining its sub-nodes until a suitable position to insert the next character is identified.

3.2.4. Step 4: Template Generation

By repeatedly applying the Step 2 and Step 3 procedures for each log message, the trie-forest gradually evolves, providing an efficient and structured representation of log messages. Each word in the message can correspond to a node in the tree, which is labeled as either a constant or a variable. Subsequently, this enables efficient log parsing and analysis, allowing researchers to uncover valuable insights within large-scale log datasets.

3.3. Advantages

The high accuracy of Polo enables more thorough log parsing, leading to improved log anomaly detection. By accurately parsing logs, similar log events can be identified and grouped. This grouping helps in accurately detecting anomalous logs during the anomaly detection process.

Here are the advantages of using Polo compared to other state-of-the-art log-parsing methods:

1. Space efficiency: Polo reduces the number of nodes that have to be stored compared to binary trees. This results in better space utilization. In scenarios where a large amount of data needs to be stored, the ternary search tree is more effective in memory usage.
2. Time efficiency: One advantage of adopting a trie-forest is that it enables us to divide the logs into multiple partitions, parse each partition independently, and generate a ternary search tree for each partition. This helps to avoid being affected by irrelevant partition noise during parsing, and also ensures the possibility of parallelism, greatly improving the efficiency of log parsing.

The time complexity of the insertion operation is $O(\log n)$, where n is the number of nodes in the tree. This is particularly advantageous for complex log messages. When a large string or number of strings are to be searched, Polo usually outperforms the prefix tree as it minimizes unnecessary comparisons. When looking for the prefix of a string, the search can be carried out on all three directions of the tree in parallel, instead of checking all the child nodes. Therefore, searching strings in a ternary search tree is more efficient than in a prefix tree.

In summary, replacing a prefix tree with Polo can lead to improved search efficiency and better space utilization, making it an effective data structure for large-scale string manipulation.

4. Evaluation

4.1. Experimental Setup

4.1.1. Datasets

He et al. released LogHub [33], a repository of system log files for research purposes, which has been used by many log-related studies. We report results evaluated on nine popular datasets ranging from distributed, operating, and mobile systems. Details are shown in Table 1, where # Dataset, # Templates, # Log Messages.

Table 1. Details of the loghub datasets.

Dataset	Templates	Log Messages
HDFS	14	2000
Hadoop	114	2000
ZooKeeper	50	2000
OpenStack	43	2000
Mac	341	2000
HPC	46	2000
Proxifier	8	2000
OpenSSH	27	2000
HealthApp	75	2000

We evaluate the anomaly detection performance on four datasets, Details are shown in Table 2:

- HDFS [34] dataset, which includes log messages by running map-reduce tasks on more than 200 nodes.
- BlueGene/L Supercomputer System (BGL) [35]. The BGL dataset is collected from a BlueGene/L supercomputer system at Lawrence Livermore National Labs (LLNL). Logs contain alert and non-alert messages identified by alert category tags. The alert messages are considered anomalous. The BGL dataset consists of 4,747,963 messages, of which 348,460 are anomalous.
- The Spirit dataset is an aggregation of system log data from the Spirit supercomputing system at Sandia National Labs [35]. In previous studies, it has been used for log analytics, such as fault detection [36]. There are more than 172 million log messages labeled as anomalous on the Spirit dataset. In this paper, we use a small set containing the first 5 million log lines of the original Spirit dataset, which contains 764,500 abnormal log messages (15.29%).
- The Thunderbird dataset is an open dataset of logs collected from a Thunderbird supercomputer at Sandia National Labs (SNL) [35]. The log data contains normal and abnormal messages, which are manually identified. In prior work, it has been used to evaluate the performance of compressors on large-scale cluster log data [37]. We leverage 6,307,012 continuous log lines for computation-time purposes, which contain 391,723 abnormal log messages (6.21%).

Table 2. Details of the anomaly detection datasets.

Dataset	Log Messages	Anomalies
HDFS	11,172,157	284,818
BGL	4,747,963	348,460
Thunderbird	6,307,012	391,723
Spirit	5,000,000	764,890

4.1.2. Evaluation Metrics

We use three metrics (i.e., Precision, Recall, and F-measure) to measure the effectiveness of Polo's offline log parsing. Precision is the ratio of generated log templates that are the same as the ground truth; Recall is the ratio of ground truth templates that are correctly figured out; F1-score is the harmonic average of the two indexes [28]:

$$Precision = \frac{TP}{TP + FP} \quad (2)$$

$$Recall = \frac{TP}{TP + FN} \quad (3)$$

$$F - measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (4)$$

TP (True Positive) refers to the real case, that is, the real situation is positive and the predicted situation is also positive.

FP (False Positive) refers to false positive example, that is, the real situation is negative and the predicted situation is positive.

FN (False Negative) refers to a false negative example, that is, the real situation is positive and the predicted situation is negative.

4.1.3. Baselines

In our experiments, we decouple the anomaly detection framework into two parts: first, log parsing transfers raw messages into structured log templates associated with

key parameters, and then the extracted features are fed to deep learning models to analyze template sequences in a session. A dependable processor should perform well as a foundational processor for log analysis, regardless of the downstream detection model used. In our experiments, we compare the performance of different baseline parsers under various anomaly detection techniques.

For each dataset, we first execute the log-parsing techniques to generate log-parsing results and compute their accuracy. We then execute the anomaly detection techniques on each of the log-parsing results and compute their accuracy in terms of precision, recall, and F1 score.

For all datasets, we first sort logs in chronological order and apply partition to generate log sequences, which will then be shuffled. Note we do not shuffle the input windows, W , generated from log sequences. Next, we utilize the first 80% of data for model training and the remaining 20% for testing. For log partition, we apply identifier-based partitioning to HDFS and fixed partitioning with one hour of partition size to BGL, Spirit, and Thunderbird.

For a fair comparison, all experiments are conducted on a machine with 1 NVIDIA 3090 GPU (24 GB of RAM), 20 Intel(R) Xeon(R) Gold 6148 CPU @ 2.40 GHz, and 289,256 GB of RAM. The parameters of all methods are fine-tuned to achieve the best results.

4.2. Experimental Results

4.2.1. Parsing Accuracy of Polo

Accurate log parsing is crucial for downstream applications, and our method is evaluated on nine datasets. Details are shown in Table 3. Notably, it achieves the highest accuracy on five datasets, showcasing its effectiveness. The HealthApp dataset, with 75 templates, highlights Polo's impressive adaptability to handle a larger template count, contributing to exceptional performance. Polo's capability to manage complexity from increased templates positively impacts accuracy, particularly evident in the HealthApp dataset.

Table 3. Accuracy of each log-parsing approach on different datasets.

Method	HDFS	Hadoop	Zookeeper	HPC	Thunderbird	Andriod	HealthApp	OpenSSH	Mac
Polo	0.999 *	0.952	0.999 *	0.993 *	0.998 *	0.995	1 *	0.968	0.983
Spell	0.932	0.995 *	0.996	0.970	0.967	0.998 *	0.389	0.906	0.983
Drain	0.997	0.945	0.978	0.988	0.995	0.822	0.997	0.998 *	0.997 *
AEL	0.877	0.950	0.999	0.981	0.993	0.829	0.997	0.937	0.952
IPLoM	0.851	0.947	0.895	0.913	0.946	0.888	0.765	0.988	0.927

The highest accuracy achieved among all methods on each dataset is marked with an asterisk (*).

4.2.2. Robustness of Log Parsers

Figure 2 displays boxplots comparing the accuracy of different methods, with Polo standing out for its smallest interquartile range and highest median accuracy, and the circles represent individual data points that are below the lower whisker and are considered outliers or extreme values. This indicates Polo consistently outperforms baseline methods across datasets. Additionally, Polo surpasses Drain, the state-of-the-art parser, due to its robustness and unique ternary tree structure. Polo's tree structure allows it to effectively parse diverse log data, leading to improved performance and stability compared to Drain.

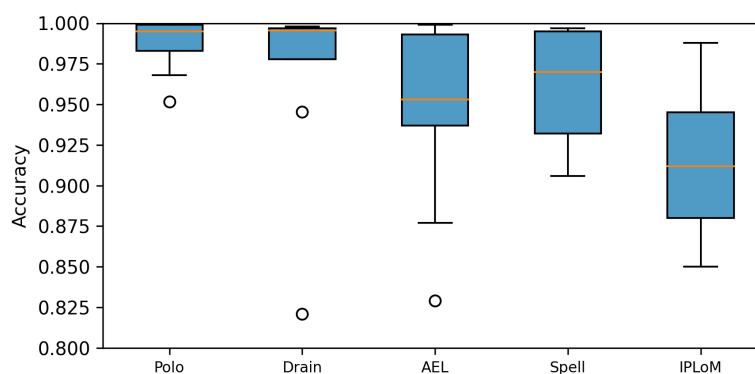


Figure 2. Compression ratio on different datasets.

4.2.3. Efficiency of Log Parsers

Log parsing is a preliminary step to many log mining tasks. Therefore, parsing time can be a bottleneck for those downstream applications.

The experimental results are presented in Table 4. Polo demonstrates better efficiency as it can process all datasets within 20 min. It is 9.93% to 40.95% faster than the four state-of-the-art parsing methods. Spell and Drain can complete the parsing within an hour for Thunderbird, HDFS, and BGL datasets, but they take more than eight hours for Spirit. In contrast, Polo typically finishes within twenty minutes. Our method, Polo, outperforms competitors in log-parsing complexity. Unlike Drain, it avoids the ternary tree explosion issue by using a forest structure, making it faster. Several factors contribute to these results. Firstly, Spell is based on the Longest Common Subsequence algorithm, which has a complexity of $O(|ti| \times m)$, where $|ti|$ represents the number of tokens to compare with the template. AEL struggles with log parsing, performing poorly compared to others. Faster log parsing has real-world benefits, enhancing system reliability and enabling quicker anomaly detection, crucial in time-sensitive environments. Our approach achieves efficiency through optimized data structures, parallel processing, and algorithmic enhancements, including a three-tiered tree structure for organized log data representation. These elements expedite parsing, providing timely analysis for proactive system maintenance and optimization.

Table 4. Time of the parsers running the log dataset.

Parsers	Polo	Drain	Spell	AEL	IPLoM
Thunderbird	0:18:13	0:29:08	0:39:07	2:21:29	0:16:00
HDFS	0:33:52	0:45:17	0:45:38	1:23:14	0:37:36
BGL	0:10:39	0:19:22	0:42:02	4:42:04	0:15:50
Spirit	0:13:34	8:05:36	16:25:09	15:09:41	0:18:13

The bold font represents the shortest time among all methods.

4.3. Effectiveness of Log Anomaly

In Table 5, we can observe that the performance of the studied models varies significantly with different log parsers. LogRobust achieves an F-measure of 0.838 and a Recall of 1.0 with the Polo parser on the BGL dataset. When experimenting with Spell, these values drop to 0.458 and 0.638, respectively, although Drain is one of the most accurate log parsers according to a recent benchmark study [32]. The results also show that LogRobust and CNN can handle log-parsing noise better than other models. This is probably because of their use of semantic vectors. Moreover, we have found that the errors generated by different log parsers have a distinct impact on the detection models. The results on the BGL and Spirit datasets show that Polo outperforms Drain, AEL, Spell, and IPLoM. This is because Polo can control the generation of additional log events, which benefits the

performance of DeepLog and LogAnomaly (they use prediction methods to forecast the next log event).

As shown in Table 5, compared to the state-of-the-art log parser, Polo has higher precision. The accuracy of log templates is crucial for the anomaly detection model. All parsers have lower recall on the BGL dataset. However, Polo outperforms most parsers on the same dataset, indicating that the ternary tree structure can efficiently handle the semantic information of log templates and is highly effective in dealing with complex log patterns. It can adapt well to downstream tasks such as anomaly detection.

Table 5. Impact of the different log-parsing results on anomaly detection accuracy.

Model		BGL					Spirit				
		Polo	Drain	Spell	AEL	IPLoM	Polo	Drain	Spell	AEL	IPLoM
DeepLog	Precision	0.607	0.248	0.231	0.243	0.233	0.609	0.414	0.602	0.337	0.607
	Recall	1.0	0.958	0.731	1.0	0.985	1.0	0.997	1.0	0.995	1.0
	F1 score	0.461	0.485	0.422	0.418	0.375	0.867	0.584	0.751	0.503	0.755
LogAnomaly	Precision	0.548	0.313	0.271	0.539	0.173	0.609	0.335	0.612	0.333	0.607
	Recall	1.0	1.0	1.0	1.0	0.992	1.0	0.995	1.0	0.891	1.0
	F1 score	0.438	0.44	0.442	0.49	0.294	0.867	0.501	0.759	0.485	0.755
LogRobust	Precision	0.896	0.358	0.317	0.661	0.726	0.949	0.544	0.947	0.335	0.507
	Recall	1.0	0.638	0.778	0.767	0.916	1.0	0.997	1.0	0.995	1.0
	F1 score	0.838	0.458	0.204	0.418	0.833	0.882	0.374	0.973	0.501	0.57
CNN	Precision	1.0	0.993	0.994	0.988	0.953	0.999	0.933	1.0	0.999	0.986
	Recall	0.970	0.942	0.942	0.942	0.767	1.0	0.998	0.995	0.982	1.0
	F1 score	0.985	0.944	0.967	0.964	0.85	0.997	0.964	0.997	0.99	0.993
Model		HDFS					Thunderbird				
		Polo	Drain	Spell	AEL	IPLoM	Polo	Drain	Spell	AEL	IPLoM
DeepLog	Precision	0.97	0.871	0.871	0.871	0.867	0.991	0.921	0.951	0.821	0.832
	Recall	1.0	0.995	0.995	0.994	1.0	0.636	0.667	0.535	0.583	0.411
	F1 score	0.983	0.929	0.929	0.928	0.89	0.299	0.229	0.235	0.224	0.191
LogAnomaly	Precision	0.971	0.871	0.871	0.871	0.867	0.211	0.209	0.234	0.348	0.217
	Recall	1.0	0.995	0.995	1.0	1.0	0.993	0.993	0.984	0.993	0.991
	F1 score	0.958	0.929	0.929	0.928	0.928	0.348	0.344	0.378	0.211	0.378
LogRobust	Precision	0.937	0.871	0.8707	0.871	0.866	0.931	0.972	0.95	0.921	0.834
	Recall	1.0	0.994	0.9945	0.994	1.0	0.969	0.484	0.614	0.445	0.991
	F1 score	0.972	0.928	0.929	0.928	0.889	0.493	0.453	0.434	0.39	0.378
CNN	Precision	0.882	0.882	0.911	0.887	0.953	0.957	0.957	0.995	0.951	0.991
	Recall	0.995	0.996	0.995	0.985	0.981	0.975	0.992	0.948	0.978	0.964
	F1 score	0.995	0.934	0.951	0.933	0.967	0.966	0.96	0.971	0.964	0.977

The bold font represents the highest accuracy achieved among all methods on each dataset.

The experimental results are shown in Table 6, where each row represents the performance with the selected parser and several model architectures. The last row reveals how much Polo increases the score compared to the best baseline results. It is noteworthy that Polo outperforms four baselines by a wide margin, regardless of the analytical techniques. We can observe that our parser surpasses Deeplog, LogAnomaly, CNN, and LogRobust by 11.5%, 4%, 1%, and 19.1%, respectively, in terms of F1-score. In LogAnomaly, Polo's lower accuracy (0.585 in Table 6) is due to the dataset's specific characteristics and methodology. Using FT-Tree for template approximation becomes less accurate with a high number of templates. The reduced frequency of matches in handling numerous templates leads to decreased accuracy in predictions, impacting Polo's performance in LogAnomaly. Overall, Polo demonstrates a promising Recall score of 0.971, indicating the effectiveness of Polo for anomaly detection.

Table 6. Average experiment results for anomaly detection.

Model						
Deeplog				LogAnomaly		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score
Drain	0.613	0.904	0.557	0.432	0.996	0.554
AEL	0.568	0.893	0.518	0.523	0.971	0.532
IPLoM	0.635	0.849	0.553	0.466	0.996	0.589
Spell	0.664	0.815	0.584	0.497	0.995	0.627
Polo	0.795	0.909	0.653	0.585	0.998	0.653
$\Delta\%$	+11.9%				+4%	
LogRobust				CNN		
	Precision	Recall	F1-score	Precision	Recall	F1-score
Drain	0.686	0.992	0.554	0.941	0.982	0.951
AEL	0.697	0.800	0.529	0.956	0.972	0.963
IPLoM	0.733	0.977	0.668	0.928	0.928	0.947
Spell	0.771	0.846	0.635	0.975	0.970	0.971
Polo	0.898	0.992	0.807	0.960	0.985	0.986
$\Delta\%$	+20.1%				+1%	

The bold font represents the highest accuracy achieved among all methods on each dataset.

4.4. Industrial Deployment

The application of Polo in cloud system monitoring can be divided into two stages: offline training and online serving. In the online stage, we utilize Kafka as a streaming channel for online log analytics. Data producers, representing different services generating raw log data at runtime, correspond to individual Kafka topics for data streaming. Our model acts as the data consumer, performing anomaly detection for each service. We employ Apache Flink for distributed log pre-processing and anomaly detection, allowing for high-performance and low-latency processing of streaming data. The detection results are visualized on a monitoring panel through Prometheus, enabling engineers to confirm true anomalies or flag false positives with simple clicks. In the offline stage, the model is (re)trained. Raw logs are initially archived and maintained in Apache HDFS, where they can be retrieved for model (re)training and evaluation. Manual threshold setting is required for alerting anomalies. Engineers can trigger model retraining if they observe performance degradation on the monitoring panel.

5. Conclusions

In this paper, we address two key limitations in current log parsers: insufficient log template parsing and a lack of parallelism. Our solution, Polo, utilizes a prefix forest of ternary trees to mine templates, achieving an average accuracy of 0.987 across nine system logs. Additionally, in downstream log analysis tasks such as anomaly detection, Polo significantly outperforms state-of-the-art parsers, highlighting the importance of accurate template parsing.

Polo's log-parsing and anomaly detection techniques have broad applications:

- IT Operations and Infrastructure Management: Polo is valuable for monitoring logs in large-scale IT environments, detecting anomalies in components such as servers, networks, and storage to prevent system failures or performance issues.
- Application Performance Monitoring: Polo can monitor logs from various applications (web, mobile, desktop) to identify performance issues, errors, or exceptions. This enables timely interventions and optimization of application performance.

- Cybersecurity and Intrusion Detection: Polo's techniques are effective in identifying suspicious log patterns or anomalies indicative of cyber threats, including unauthorized access attempts, unusual network traffic, or abnormal user behaviors.

Author Contributions: Conceptualization, Y.Z. and Y.S.; methodology, Y.Z. and Y.S.; software, Y.Z. and Y.S.; validation, Y.Z. and Y.S.; formal analysis, Y.Z.; investigation, Y.Z.; resources, Y.Z.; data curation, Y.Z.; writing—original draft preparation, Y.Z.; writing—review and editing, Y.Z.; visualization, Y.Z.; supervision, Y.S.; project administration, Y.S.; funding acquisition, Y.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflict of interest.

References

- Chen, M.; Zheng, A.X.; Lloyd, J.; Jordan, M.I.; Brewer, E. Failure diagnosis using decision trees. In Proceedings of the International Conference on Autonomic Computing, New York, NY, USA, 17–18 May 2004; pp. 36–43.
- Mi, H.; Wang, H.; Zhou, Y.; Lyu, M.R.; Cai, H. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Trans. Parallel Distrib. Syst.* **2013**, *24*, 1245–1255. [\[CrossRef\]](#)
- Bu, J.; Liu, Y.; Zhang, S.; Meng, W.; Liu, Q.; Zhu, X.; Pei, D. Rapid deployment of anomaly detection models for large number of emerging kpi streams. *J. Abbr.* **2008**, *10*, 142–149.
- El-Masri, D.; Petrillo, F.; Guéhéneuc, Y.-G.; Hamou-Lhadj, A.; Bouziane, A. A systematic literature review on automated log abstraction techniques. *Inf. Softw. Technol.* **2020**, *122*, 106276. [\[CrossRef\]](#)
- Zhang, J.; Xie, Z.; Sun, J.; Zou, X.; Wang, J. A cascaded R-CNN with multiscale attention and imbalanced samples for traffic sign detection. *IEEE Access* **2020**, *8*, 29742–29754. [\[CrossRef\]](#)
- Chen, Y.; Wang, J.; Xia, R.; Zhang, Q.; Cao, Z.; Yang, K. RETRACTED ARTICLE: The visual object tracking algorithm research based on adaptive combination kernel. *J. Ambient Intell. Humaniz. Comput.* **2019**, *10*, 4855–4867. [\[CrossRef\]](#)
- Zhang, J.; Wang, W.; Lu, C.; Wang, J.; Sangaiah, A.K. Lightweight deep network for traffic sign classification. *Ann. Telecommun.* **2020**, *75*, 369–379. [\[CrossRef\]](#)
- Xie, K.; Li, X.; Wang, X.; Xie, G.; Wen, J.; Cao, J.; Zhang, D. Fast tensor factorization for accurate internet anomaly detection. *IEEE/ACM Trans. Netw.* **2017**, *25*, 3794–3807. [\[CrossRef\]](#)
- He, S.; He, P.; Chen, Z.; Yang, T.; Su, Y.; Lyu, M.R. A survey on automated log analysis for reliability engineering. *ACM Comput. Surv.* **2021**, *54*, 1–37. [\[CrossRef\]](#)
- Zhu, J.; He, S.; Liu, J.; He, P.; Xie, Q.; Zheng, Z.; Lyu, M.R. Tools and benchmarks for automated log parsing. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Montreal, QC, Canada, 27 May 2019; pp. 121–130.
- He, S.; Lin, Q.; Lou, J.-G.; Zhang, H.; Lyu, M.R.; Zhang, D. Identifying impactful service system problems via log analysis. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista, FL, USA, 4–9 November 2018; pp. 60–70.
- Khatuya, S.; Ganguly, N.; Basak, J.; Bharde, M.; Mitra, B. Adele: Anomaly detection from event log empiricism. In Proceedings of the IEEE INFOCOM 2018-IEEE Conference on Computer Communications, Honolulu, HI, USA, 16–19 April 2018; pp. 2114–2122.
- Ma, M.; Zhang, S.; Pei, D.; Huang, X.; Dai, H. Robust and rapid adaption for concept drift in software system anomaly detection. In Proceedings of the 2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE), Memphis, TN, USA, 15–18 October 2018; pp. 13–24.
- Jiang, Z.M.; Hassan, A.E.; Flora, P.; Hamann, G. Abstracting execution logs to execution events for enterprise applications (short paper). In Proceedings of the 2008 The Eighth International Conference on Quality Software, Oxford, UK, 12–13 August 2008; pp. 181–186.
- Makanju, A.A.; Zincir-Heywood, A.N.; Milios, E.E. Clustering event logs using iterative partitioning. In Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, NY, USA, 28 June–1 July 2009; pp. 1255–1264.
- Du, M.; Li, F. Spell: Streaming parsing of system event logs. In Proceedings of the 2016 IEEE 16th International Conference on Data Mining (ICDM), Barcelona, Spain, 12–15 December 2016.
- He, P.; Zhu, J.; Zheng, Z.; Lyu, M.R. Drain: An online log parsing approach with fixed depth tree. In Proceedings of the 2017 IEEE International Conference on Web Services (ICWS), Honolulu, HI, USA, 25–30 June 2017; pp. 33–40.
- Huo, Y.; Su, Y.; Lee, C.; Lyu, M.R. SemParser: A Semantic Parser for Log Analytics. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 14–20 May 2023; pp. 881–893.

19. Le, V.; Zhang, H. Log-based anomaly detection without log parsing. In Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne Australia, 15–19 November 2021; pp. 492–504.
20. Wang, H.; Zhou, C.; Wu, J.; Dang, W.; Zhu, X.; Wang, J. Deep structure learning for fraud detection. In Proceedings of the 2018 IEEE International Conference on Data Mining (ICDM), Singapore, 17–20 November 2018; pp. 567–576.
21. Jia, T.; Chen, P.; Yang, L.; Li, Y.; Meng, F.; Xu, J. An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services. In Proceedings of the 2017 IEEE International Conference on Web Services (ICWS), Honolulu, HI, USA, 25–30 June 2017; pp. 25–32.
22. Yu, W.; Cheng, W.; Aggarwal, C.C.; Zhang, K.; Chen, H.; Wang, W. NetWalk: A Flexible Deep Embedding Approach for Anomaly Detection in Dynamic Networks. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, London, UK, 19–23 August 2018; pp. 2672–2681.
23. Xia, B.; Bai, Y.; Yin, J.; Li, Y.; Xu, J. LogGAN: A Log-Level Generative Adversarial Network for Anomaly Detection Using Permutation Event Modeling. *Inf. Syst. Front.* **2021**, *23*, 285–298. [[CrossRef](#)]
24. Oprea, A.; Li, Z.; Yen, T.-F.; Chin, S.H.; Alrwais, S. Detection of Early-Stage Enterprise Infection by Mining Large-Scale Log Data. In Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Rio de Janeiro, Brazil, 22–25 June 2015; p. 45.
25. He, S.; Zhu, J.; He, P.; Lyu, M.R. Experience report: System log analysis for anomaly detection. In Proceedings of the 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), Ottawa, ON, Canada, 23–27 October 2016; pp. 207–218.
26. Joulin, A.; Grave, E.; Bojanowski, P.; Douze, M.; Jégou, H.; Mikolov, T. Fasttext. zip: Compressing text classification models. *arXiv* **2016**, arXiv:1612.03651.
27. Pennington, J.; Socher, R.; Manning, C.D. Glove: Global vectors for word representation. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar, 25–29 October 2014; pp. 1532–1543.
28. Du, M.; Li, F.; Zheng, G.; Srikumar, V. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, New York, NY, USA, 30 October–3 November 2017; pp. 1285–1298.
29. Meng, W.; Liu, Y.; Zhu, Y.; Zhang, S.; Pei, D.; Liu, Y.; Chen, Y.; Zhang, R.; Tao, S.; Sun, P.; et al. Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. *IJCAI* **2019**, *19*, 4739–4745.
30. Zhang, X.; Xu, Y.; Lin, Q.; Qiao, B.; Zhang, H.; Dang, Y.; Xie, C.; Yang, X.; Cheng, Q.; Li, Z.; et al. Robust log-based anomaly detection on unstable log data. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26–30 August 2019; pp. 807–817.
31. Lu, S.; Wei, X.; Li, Y.; Wang, L. Detecting anomaly in big data system logs using convolutional neural network. In Proceedings of the 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), Athens, Greece, 12–15 August 2018; pp. 151–158.
32. He, P.; Zhu, J.; He, S.; Li, J.; Lyu, M.R. An evaluation study on log parsing and its use in log mining. In Proceedings of the 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Toulouse, France, 28 June–1 July 2016; pp. 654–661.
33. He, S.; Zhu, J.; He, P.; Lyu, M.R. Loghub: A large collection of system log datasets towards automated log analytics. *arXiv* **2020**, arXiv:2008.06448.
34. Xu, W.; Huang, L.; Fox, A.; Patterson, D.; Jordan, M.I. Detecting large-scale system problems by mining console logs. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating systems principles, Big Sky, MT, USA, 11–14 October 2009; pp. 117–132.
35. Oliner, A.; Stearley, J. What supercomputers say: A study of five system logs. In Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07), Edinburgh, UK, 25–28 June 2007; pp. 575–584.
36. Stearley, J.; Oliner, A.J. Bad words: Finding faults in spirit's syslogs. In Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID), Lyon, France, 19–22 May 2008; pp. 765–770.
37. Balakrishnan, R.; Sahoo, R.K. Lossless compression for large scale cluster logs. In Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium, Rhodes, Greece, 25–29 April 2006; p. 7.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.