applied sciences

MDPI

Article

Efficient Online Log Parsing with Log Punctuations Signature

Shijie Zhang and Gang Wu *

School of Software, Shanghai Jiao Tong University, Shanghai 200240, China; flamingoo@sjtu.edu.cn
* Correspondence: dr.wugang@sjtu.edu.cn

Abstract: Logs, recording the system runtime information, are frequently used to ensure software system reliability. As the first and foremost step of typical log analysis, many data-driven methods have been proposed for automated log parsing. Most existing log parsers work offline, requiring a time-consuming training progress and retraining as the system upgrades. Meanwhile, the state of the art online log parsers are tree-based, which still have defects in robustness and efficiency. To overcome such limitations, we abandon the tree structure and propose a hash-like method. In this paper, we propose LogPunk, an efficient online log parsing method. The core of LogPunk is a novel log signature method based on log punctuations and length features. According to the signature, we can quickly find a small set of candidate templates. Further, the most suitable template is returned by traversing the candidate set with our log similarity function. We evaluated LogPunk on 16 public datasets from the LogHub comparing with five other log parsers. LogPunk achieves the best parsing accuracy of 91.9%. Evaluation results also demonstrate its superiority in terms of robustness and efficiency.

Keywords: log parsing; log signature; punctuations; online algorithm

check for updates

Citation: Zhang, S.; Wu, G. Efficient Online Log Parsing with Log Punctuations Signature. *Appl. Sci.* **2021**, *11*, 11974. https://doi.org/10.3390/app112411974

Academic Editor: Agostino Forestiero

Received: 17 October 2021
Accepted: 12 December 2021
Published: 16 December 2021

1. Introduction

Logging is the practice of recording events that provides information about the system running status and execution paths. Earlier, system operators could understand runtime behaviors and diagnose failures by manually analyzing logs [1,2]. A modern system service are often composed of several basic services [3]. Moreover, modern system clusters usually contain hundreds of nodes, some of which are even geographically distributed [4]. In this context, with the increasing scale and complexity of modern software systems, the volume of logs explodes [5]. It leads to the emergence of automated log analysis approaches. These automated approaches bring more tools (e.g., anomaly detection [6,7], failure prediction [8,9], and failure diagnosis [10,11]) to ensure system reliability, which is an indispensable step towards AIOps (Artificial Intelligence for IT Operations).

Logs are unstructured text printed by logging statements in the source code. As shown in Figure 1, a logging statement is specified by log level, static string, and dynamic variables. As the variable value changes at runtime, a logging statement can produce different log messages. Typically, a log message may contain a timestamp, log level, logger name, and raw message content. Different log messages from the same logging statement have the same log template (event type).

Most data mining models used in log analysis require structured input. Therefore raw logs cannot be used as input directly [12–14]. To conquer the unstructured nature of raw logs, we need log parsing to convert the unstructured logs into a structured format before analysis [15]. The goal of log parsing is to extract the static template, dynamic variables, and the header information (timestamp, log level, logger name) from the raw log message to a structured format . Such structured data can be fed into downstream log analysis models.

Logging statement	`Log.d(tag, String.format("getRecentTasks: num=%d,flags=%s,totalTasks=%d", num, flags, totalTasks))`
Unstructured raw log	`03-17 16:13:47.518  1702  8671 D ActivityManager: getRecentTasks: num=`*`20`*`,flags=`*`0x1`*`,totalTasks=`*`46`*
Structured parsed log	**Timestamp:** `03-17 16:13:47.518`; **Pid:** `1702` **Tid:** `8671`; **Level:** `D`; **Tag:** `ActivityManager` **Static template:** `getRecentTasks: num=<*>,flags=<*>,totalTasks=<*>` **Dynamic variables:** *`20, 0x1, 46`*

Figure 1. An illustrative example of log parsing.

Regular expressions are always a choice for log parsing [15], but it is only practicable for a small number of log templates. Continuous human efforts are needed to develop and maintain the regular expressions, which are labor-intensive and error-prone [16,17]. As modern software has many log templates and constantly evolves [18], developing and maintaining such regular expressions could be a nightmare.

To alleviate the pain of human efforts, researchers have proposed many automated data-driven approaches. Earlier works leverage data mining approaches such as frequent pattern mining [19,20], clustering [21,22], and iterative partitioning [23] to extract the common part of log messages under the same cluster as the log template. However, all these approaches are offline, which requires a time-consuming training process and can not deal with the template changes caused by software updates. In contrast, online log parsers parse logs in a streaming fashion and do not require offline training. Therefore, what modern systems need is online log parsing, which is only studied in a few preliminary works [17,24–26].

Spell [26] and Drain [17] are state-of-the-art online log parsers. Spell measures the distance between log messages through longest common subsequence (LCS), and uses prefix tree to optimize the processing time of each log message close to linear. Drain also adopts a tree structure. Unlike Spell, it has more heuristics and strong assumptions about the length and preceding tokens of log messages.

In practice, we find that the tree structure has some limitations in robustness and efficiency. Although, existing online parsers achieve good parsing accuracy on specific datasets. Their parsing accuracy fluctuates across different datasets, which means that they are not robust (e.g., the average accuracy of Spell [26] is less than 80%). In addition, with the rapid growth of log volume and the increasing demand for low latency log analysis, efficiency becomes an essential concern of log analysis [18,27]. However, the previous benchmark [15] shows that Spell [26] and Drain [17] are not efficient enough. In this work, we abandon the tree structure and propose a hash-like method, which really improves robustness and efficiency.

Unlike previous work, we focus on meaningless punctuations in log messages rather than meaningful words. Because we believe that words processing is the reason why the previous approaches are inefficient. Our intuition is that punctuation marks of log messages from the same template tend to be the same, which means simple punctuations imply template information. There are fixed types of punctuations, and they are easy to process.

This paper proposes LogPunk, a robust and efficient log parser based on our novel log signature method. LogPunk is designed as a general-purpose online log parsing method, which is system and log type agnostic. We use log punctuations and length information (cf. Section 2) to generate log signatures. Each log signature corresponds to a signature group. Obviously, log messages with the same event type will have the same log signature and get into the same group. However, a signature group may also contain log messages from different event types with the same signature. We call this *signature collision* easy to solve by further calculating the similarity between the log message and event templates. Fortunately, we found in our experiments that each signature group only corresponds to one or two event templates in most cases.

We evaluate LogPunk on 16 datasets from the LogHub [28] comparing with five other log parsers. Experiments demonstrate that LogPunk is efficient but without loss of accuracy. LogPunk achieves the highest accuracy on 14 datasets and the best average PA of 0.919. More importantly, LogPunk performs consistently across different datasets, which means it is robust.

In summary, our paper makes the following contributions:

- We propose a novel and efficient log signature method based on log punctuations and length information applied for log parsing;
- We present an online log parser based on our log signature method, named LogPunk, which is better than the previous log parsers in robustness and efficiency;
- We conduct extensive experiments on 16 datasets and comparing LogPunk with five other log parsers. The results show that LogPunk is accurate, robust, and efficient.

The paper is organized as follows. Section 2 introduces the background of log parsing. Section 3 describes our log signature method and the implementation of LogPunk. Section 4 shows the results of evaluating LogPunk on the LogHub datasets. Section 5 compares LogPunk with tree-based methods and discusses the validity. Section 6 introduces the related work of three categories of data-driven log parsing approaches. Finally, Section 7 concludes the paper.

2. Problem Description

As shown in Figure 1, the goal of log parsing is to extract the static template, dynamic variables, and header information from a raw log message. As the header information usually follows a fixed format in the same system, regular expressions are commonly used to extract the header information and the log content.

Log content is the central processing object of log parsing, composed of static templates and dynamic variables. It can be defined as a tuple of $EV = \{(e_i, v_i) : e \in E,\ i = 1, 2, \cdots\}$, where E is the set of all log templates, $k = |E|$ is the number of all distinct templates and v_i is a list of variables.

Formally, given a set of log messages $L = \{l_1, l_2, \cdots, l_m\}$ that is produced by k logging statements (log templates) from the source code, where the k is unknown. A log parser is to parse L to get all k log templates.

After extracting the log content from the raw log, it is common to process the content with string splitting. By splitting the content, we get a split list, and each element in the list is called a token. Log message length is defined as the number of tokens in the split list. Formally, each log message consists of a bounded list of tokens, $\boldsymbol{t_i} = \{t_j : t \in T, j = 1, 2, \cdots, n\}$, where T is a set of all tokens, j is the token index within the split list and $n = |\boldsymbol{t_i}|$ is the number of tokens (message length).

The token is usually the smallest granularity to determine the template and variable part. A token is either part of the static templates or dynamic variables. Therefore, how to split the content has a significant influence on the parsing results. Different delimiters result in different split token lists. Previous studies often use spaces as delimiters.

Online algorithms usually have two core steps: first, find the candidate template set; and then traverse the candidate set to find the most suitable template.

Quickly finding candidate sets is critical in the first step. Inspired by the prefix tree data structure, Spell [26] and Drain [17] adopt a tree structure to find candidate sets to filter out most irrelevant templates. Such a tree structure is usually very complex. Some tree nodes have hundreds of children, and the tree needs frequent maintenance. In addition, the tree-based method does not guarantee that the returned template is the longest common subsequence. For example, the log message $l = DAPBC$ (each letter represents a token, and the space between tokens is omitted) and the template set $s = \{DA, ABC\}$, the prefix tree returns DA instead of ABC. Since the second step will traverse the candidate set, the size of the candidate set also matters.

In the previous benchmark [15], Spell and Drain failed to reach the first level efficiency, which means there is still room for improvement. Our goal is to find a more efficient method to obtain smaller candidate sets.

3. Methodology

As mentioned above, our approach is online, and it can process logs in a streaming way. As shown in Figure 2, when a new raw log message arrives, LogPunk will first preprocess it with some simple regular expressions to extract the content and split its content into a token list. Then, a log signature will be generated for this log message based on the token list. We can quickly locate the signature group that contains a list of possible templates according to the signature. The most suitable template will be returned by searching the signature group with our specially designed log similarity function. If no such template is found, this log message will be appended as a new template itself.

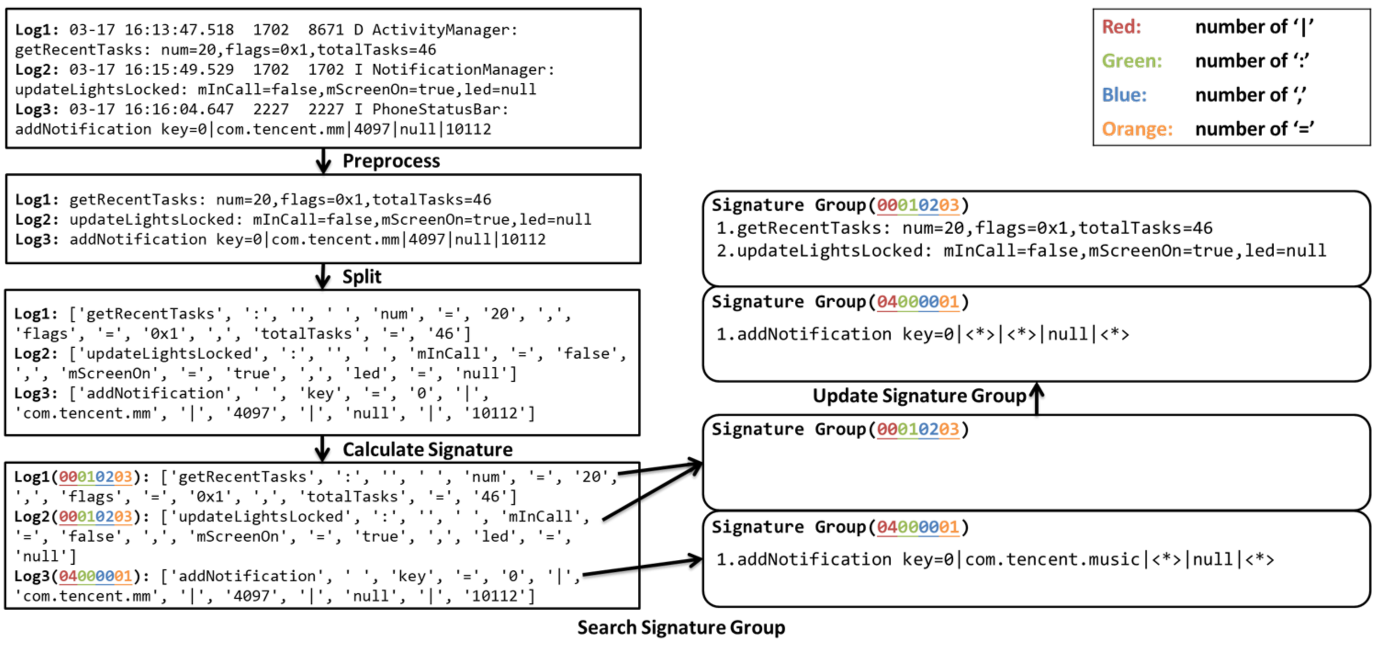
```
Log1: 03-17 16:13:47.518  1702  8671 D ActivityManager:
getRecentTasks: num=20,flags=0x1,totalTasks=46
Log2: 03-17 16:15:49.529  1702  1702 I NotificationManager:
updateLightsLocked: mInCall=false,mScreenOn=true,led=null
Log3: 03-17 16:16:04.647  2227  2227 I PhoneStatusBar:
addNotification key=0|com.tencent.mm|4097|null|10112
```
```
Log1: getRecentTasks: num=20,flags=0x1,totalTasks=46
Log2: updateLightsLocked: mInCall=false,mScreenOn=true,led=null
Log3: addNotification key=0|com.tencent.mm|4097|null|10112
```
```
Log1: ['getRecentTasks', ':', '', ' ', 'num', '=', '20', ',',
'flags', '=', '0x1', ',', 'totalTasks', '=', '46']
Log2: ['updateLightsLocked', ':', '', ' ', 'mInCall', '=', 'false',
',', 'mScreenOn', '=', 'true', ',', 'led', '=', 'null']
Log3: ['addNotification', ' ', 'key', '=', '0', '|',
'com.tencent.mm', '|', '4097', '|', 'null', '|', '10112']
```
```
Log1(00010203): ['getRecentTasks', ':', '', ' ', 'num', '=', '20',
',', 'flags', '=', '0x1', ',', 'totalTasks', '=', '46']
Log2(00010203): ['updateLightsLocked', ':', '', ' ', 'mInCall',
'=', 'false', ',', 'mScreenOn', '=', 'true', ',', 'led', '=',
'null']
Log3(04000001): ['addNotification', ' ', 'key', '=', '0', '|',
'com.tencent.mm', '|', '4097', '|', 'null', '|', '10112']
```
```
Signature Group(00010203)
 1.getRecentTasks: num=20,flags=0x1,totalTasks=46
 2.updateLightsLocked: mInCall=false,mScreenOn=true,led=null
Signature Group(04000001)
 1.addNotification key=0|<*>|<*>|null|<*>
```
```
Signature Group(00010203)
Signature Group(04000001)
 1.addNotification key=0|com.tencent.music|<*>|null|<*>
```

Figure 2. Basic workflow of our method.

3.1. Step 1 Preprocess and Split

This step extracts a list of tokens from each log message through three sub-steps: (1) log content extraction; (2) common variables substitution; (3) log content split.

First, a pre-defined regular expression is always used to extract the log content and the header information (e.g., timestamp, log level, and logger name) [15]. Since the header information often follows a fixed format in the same software system, it is convenient to extract it directly. Therefore, the log content is what we are concerned about during the log parsing.

After getting the log content, some simple user-defined regular expressions replace common variables (e.g., IP address, URL, and file path) with a special token "<*>". Moreover, the LogPai benchmark (cf. Section 4.1) has already defined such regular expressions. To avoid biased comparison, we apply these regular expressions to all log parsers in our experiments.

Finally, we split the log content with delimiters. Instead of using spaces as delimiters in prior works, we use more delimiters, such as commas (","), semicolons (";"), colons (":") and equal signs ("="). Because variable and constant parts are not always separated by spaces. In addition, we keep all the delimiters in the token list after splitting. Take log message *log1* in Figure 2 for example, the token list of *"getRecentTasks: num=20,flags=0x1,totalTasks=46"* is *['getRecentTasks', ':', ' ', 'num', '=', '20', ',', 'flags', '=', '0x1', ',', 'totalTasks', '=', '46']*.

3.2. Step 2 Generate Log Signature

In this step, we generate a log signature for each log message (Algorithm 1). To assign the same signature for log messages with the same templates, we must find their common points. Log messages with the same event type have words in common. Many prior studies have emerged based on this observation [17,26]. As mentioned above, we find that log messages with the same event type have the same punctuation marks. We use such punctuation information to generate the log signature.

Algorithm 1: Log signature

```
Input: The log content token_list after splitting.
Output: A number representing the log signature.
init PUNCTUATION_TABLE = {'|', '"', '(', '*', ';', ',', '=', ':', ' '};
init freq_dict = dictionary with default value of 0;
init first_non_digital_token = None;
init signature = 0;
for token in token_list do
    // count punctuation characters
    for char in token do
        if char in PUNCTUATION_TABLE then
            $freq\_dict[ch] += 1$;
        end
    end
    // find the first non digital token
    if first_non_digital_token is None and token contains no digital character then
        $first\_non\_digital\_token = token$;
    end
end
for char, freq in freq_dict do
    // assume the maximum punctuation count is less than 100
    $signature = signature * 100 + freq$;
end
// assume the maximum token length is less than 100
$signature = signature * 100 + len(first\_non\_digital\_token)$;
return signature
```

Specifically, we put all punctuations used to calculate the signature in a table called the *punctuation table*. For each character in the split token list, we count it in a frequency table if it appears in the punctuation table. For the convenience of the subsequent calculation, we convert the frequency table into a number like a radix conversion. For each punctuation and its frequency, we multiply by the radix and add the current frequency. For example, in Figure 2, there is zero vertical bar ("|"), one colon (":"), two semicolons (","), and three equal signs ("=") in the log message *log1*. So, the signature calculated is 00010203 if we take ten as the radix. In the implementation, we take 100 as the radix, assuming that the frequency of any punctuation will not exceed 100. We can quickly locate the signature group according to the number in the subsequent step by converting. This number is returned as the signature. Algorithm 1 shows the detail of how to calculate the log signature.

In practice, not all punctuations can be used to calculate the signature. For example, a logging statement may generate variables like "+1" and "−1" simultaneously. In this case, "+" and "−" should not be used to calculate the signature. By eliminating the punctuations appearing in variables, we get a stable punctuation table, and it performs well on 16 datasets evaluated. The punctuations we selected are vertical bar ("|"), double quotation ("""), parenthesis ("("), asterisk ("*"), semicolon (";"), comma (","), equal sign ("="), colon (":"), and space (" ").

Ideally, a log signature should correspond to only one event type. In this case, the log signature can be used to identify the event type uniquely. However, log messages from different event types may get the same log signature, called signature collisions. For example, log message *log1* and *log2* get the same signature (00010203), but they are from the different templates in Figure 2. The average number of templates corresponding to the same signature is called the collision index. By calculating the collision index, we can measure the severity of the collision. If we only use the punctuation table to calculate the signature, the collision index on evaluated datasets is 1.76. To optimize the collision index, inspired by previous work [17], we use the length information of the first non-digital token (tokens without digital characters). Finally, we get an average collision index of 1.25 on all the 2000 log messages subsets (cf. Section 4.1) from the 16 evaluated datasets.

3.3. Step 3 Search Signature Group

In this step, we search the signature group to find the most suitable template. The signature group maintains templates with the same signature in a list. The collision index is also the average number of templates in each signature group. Each signature corresponds to a signature group, and the corresponding relationship is recorded in a hash table. By looking at the hash table, we can quickly find the signature group. Due to the small number of templates in each signature group, we find the most suitable template by calculating the log similarity one by one.

In practice, we traverse the template list in the signature group to find the template with the largest similarity compared with the current log message. If the similarity is greater than a given *similarity threshold*, the template index will be returned. Like prior work, we consider that log messages with the same event type have the same length (cf. Section 2). So, if the length is not the same, the similarity value is zero. The similarity between the log message and the log template is defined as the number of identical tokens divided by the total number.

In addition, we found that some log templates are very similar because they have the same prefix tokens, and only the last few tokens are different. For example, BGL E99 *"program interrupt: fp cr field ....<*>"* and E100 *"program interrupt: fp cr update.............<*>"*. We apply a soft prefix tokens matching method before calculating the log similarity to deal with this case. To be specific, we compare the first N non-punctuation tokens between the template and the message. If tokens in the same position do not contain digital characters and are not the same, we think the template does not match. The user specifies the variable N as a hyperparameter called *prefix threshold*. The complete process of log similarity calculation is shown in Algorithm 2.

As each signature group contains an independent subset of the whole messages, existing online log parsing methods can also be applied to search templates inside each signature group. Take Drain for example, the core data structure of Drain is a fixed-depth tree. If we want to utilize Drain to search signature groups, we build such a tree structure and adopt all Drain steps inside each signature group. In our experiment, we introduce our log signature method to Spell and Drain in this way.

Algorithm 2: Log similarity

```
Input: The split template_token_list; the split content_token_list; prefix threshold pt.
Output: A number ranges [0, 1] representing the log similarity.
$m, n = len(template\_token\_list), len(content\_token\_list)$;
// the length of token list should be the same
if $m$ != $n$ then
    return 0
end
$count, total = 0, 0$;
for t1, t2 in zip(template_token_list, content_token_list) do
    if t1 in PUNCTUATION_TABLE or t2 in PUNCTUATION_TABLE then
        // punctuations should be the same
        if t1 != t2 then
            return 0
        end
    else
        if t1 == t2 then
            $count += 1$;
        end
        // soft prefix tokens matching
        if not (t1 contains digital character or t2 contains digital character) then
            if pt > 0 then
                return 0
            end
        end
        $total += 1$;
        $pt -= 1$;
    end
end
return $count/total$
```

3.4. Step 4 Update Signature Group

In the previous step, the most suitable template index is returned. If the index is valid, we will update the template by replacing different tokens in the same position with a special token "<*>". If the index is -1, it means that no suitable template is found. We will append this log message to the template list as a new template and return its index. Finally, the log *event id* is calculated as a *(log signature, template index)* tuple.

4. Evaluation

In this section, we evaluate LogPunk on 16 benchmark datasets from the LogHub [28] from three aspects: accuracy, robustness, and efficiency.

Accuracy. Accurate log parsers can correctly identify the static template and dynamic variables in the log content;

Robustness. Robust log parsers should perform consistently across different datasets, so they can be applied to more environments;

Efficiency. As log parsing is the first step of log analysis, inefficient log parsing cannot meet the real-time requirements.

We compare LogPunk with five previous state-of-the-art log parsers, including Drain [17], Spell [26], AEL [29], LenMa [25], and IPLoM [23]. All of them have been included in the LogPai benchmark [15]. As mentioned above, we also introduce our log signature method to Spell and Drain (denote as Spell+ and Drain+, respectively), which means that we utilize Spell and Drain to search signature groups. All experiments were conducted on a Linux machine with an 8-core Intel(R) Core(TM) i7-7700HQ CPU @ 2.80 GHz, 16 GB RAM, running 64-bit Ubuntu 18.04.5 LTS.

4.1. LogHub Dataset and Accuracy Metrics

Our benchmark datasets come from the LogHub data repository [28]. LogHub maintains a collection of logs from 16 different systems spanning distributed systems, supercomputers, operating systems, mobile systems, server applications, and stand-alone software. Many prior log parsing research [15–18,26] evaluated their approaches on these logs.

As illustrated in Table 1, LogHub contains 440 million log messages which amount to 77 GB. We can see that there are great differences in the number of templates in different datasets. The Android dataset has a maximum of 76,923 templates, while the Proxifier has a minimum of 9 templates. The average length of log messages is basically in the tens or twenties, but the maximum log message length of some datasets can reach hundreds. In terms of the log content, there is little difference among datasets, which is mostly readable free text.

Benefiting from the large size and diversity of LogHub datasets, we can measure the accuracy of log parsers and test their robustness and efficiency. LogHub picked up a subset of 2000 log messages from each dataset and manually labeled the event templates as the ground truth to ensure a consistent benchmark environment. In Table 1, "#Templates(2k)" indicates the number of event templates in the 2000 log subsets. Such manually labeled data are used to evaluate the accuracy and robustness of our log parser.

Table 1. Datasets overview.

Platform	Description	#Templates(2k)	#Templates (Total)	Length (Max, Average)	Size
Android	Android framework	166	76,923	32, 13.31	183.37 MB
Apache	Apache web server error	6	44	14, 12.28	4.90 MB
BGL	Blue Gene/L supercomputer	120	619	84, 15.32	708.76 MB
Hadoop	Hadoop map reduce job	114	298	50, 14.82	48.61 MB
HDFS	Hadoop distributed file system	14	30	111, 12.45	1.47 GB
HealthApp	Health app	75	220	14, 4.93	22.44 MB
HPC	High performance cluster	46	104	47, 9.56	32.00 MB
Linux	Linux system	118	488	24, 14.39	2.25 MB
Mac	Mac OS	341	2214	249, 15.49	16.09 MB
OpenSSH	OpenSSH server	27	62	19, 13.81	70.02 MB
OpenStack	OpenStack infrastructure	43	51	31, 20.63	58.61 MB
Proxifier	Proxifier software	8	9	27, 13.73	2.42 MB
Spark	Spark job	36	456	22, 12.76	2.75 GB
Thunderbird	Thunderbird supercomputer	149	4040	132, 17.52	29.60 GB
Windows	Windows event	50	4833	42, 31.93	26.09 GB
Zookeeper	ZooKeeper service	50	95	26, 13.46	9.95 MB

Same as the prior work by Zhu et al. [15], we adopt the *parsing accuracy* (PA) metric to measure the effectiveness of our log parser. PA is defined as the ratio of correctly parsed log messages over the total number of log messages. After parsing, each log message will be assigned with an event id suggesting which event type it belongs to. Regarding an event type, we consider it as correct if and only if all its log messages in the ground truth are parsed with the same event id. PA is stricter than the standard evaluation metrics, such as precision, recall, and F1-measure.

4.2. Accuracy

As illustrated in Table 2, we compare the accuracy of LogPunk with eight other baseline log parsers on 16 log datasets. According to the row, we can compare the PA of different log parsers on the same dataset. Moreover, the column demonstrates the accuracy distribution of the same log parser across datasets. Following the prior work [15], PA values greater than 0.9 are highlighted in bold, and the best PA values of each dataset are marked with asterisk "*".

From Table 2, we can observe that LogPunk achieves the best accuracy on ten datasets out of 16, which significantly outperforms other baseline methods. In addition to that, LogPunk achieves over 0.9 accuracy on 13 datasets. In the remaining datasets, LogPunk also has a comparable accuracy. On average, LogPunk has the best accuracy of 0.919, followed by Drain+ of 0.877.

In addition, Spell+ and Drain+ are better than the original Spell and Drain. Their accuracy is improved by 0.055 and 0.011, respectively, which shows that our signature method is effective and can be applied to other log parsers as an enhancement.

The average accuracy of the HDFS and Apache datasets almost reaches 100%. Because HDFS and Apache logs have relatively simple structures, and the number of templates is small. Meanwhile, some datasets could not be parsed accurately due to their complex structure and abundant event templates, such as Mac and Linux. All log parsers perform poorly on the Proxifier dataset because it is skewed. It only has eight templates in the sample set, while template E8 takes up 947 out of 2000, and it is not easy to parse.

Table 2. Accuracy of log parsers on different datasets.

Dataset	IPLoM	LenMa	AEL	Spell	Spell+	Drain	Drain+	LogPunk	Best
Android	0.712	0.88	0.682	**0.919**	**0.922**	**0.911**	**0.913**	**0.936** *	0.936
Apache	**1** *	**1** *	**1** *	**1** *	**1** *	**1** *	**1** *	**1** *	1
BGL	**0.939**	0.69	**0.957**	0.787	0.822	**0.963**	**0.97**	**0.979** *	0.979
Hadoop	**0.954**	0.885	0.869	0.778	0.795	**0.948**	**0.949**	**0.992** *	0.992
HDFS	**1** *	**0.998**	**0.998**	**1** *	**0.998**	**0.998**	**0.998**	**0.998**	1
HealthApp	0.822	0.174	0.568	0.639	0.686	0.78	0.78	**0.901** *	0.901
HPC	0.829	0.83	**0.903**	0.654	0.898	0.887	**0.926**	**0.939** *	0.939
Linux	0.672	0.701	0.673	0.605	0.739	0.69	0.749 *	0.741	0.749
Mac	0.671	0.698	0.764	0.757	0.804	0.787	0.858 *	0.852	0.858
OpenSSH	0.54	**0.925**	0.538	0.554	0.803	0.788	0.788	**0.995** *	0.995
OpenStack	0.331	0.743	0.758	0.764	0.764	0.733	0.733	**1** *	1
Proxifier	0.517	0.508	0.495	0.527 *	0.527 *	0.527 *	0.527 *	0.504	0.527
Spark	**0.92**	0.884	**0.905**	**0.905**	**0.905**	**0.92**	**0.92**	**0.923** *	0.923
Thunderbird	0.663	**0.943**	**0.941**	0.844	**0.95**	**0.955**	**0.955** *	**0.951**	0.955
Windows	0.567	0.566	0.69	**0.989**	**0.99**	**0.997** *	**0.997** *	**0.996**	0.997
Zookeeper	**0.962**	0.841	**0.921**	**0.964**	**0.964**	**0.967**	**0.967**	**0.995** *	0.995
Average	0.756	0.767	0.791	0.793	0.848	0.866	0.877	0.919	N.A.

LogPunk has the best accuracy for the following reasons. First, our effective signature method with a collision index of 1.25 avoids template crowding in one signature group. Even though the whole template set is complex, the situation is much simpler in each signature group. Second, adequate delimiters ensure the separation of variables and templates to deal with some complex situations. Third, we provide two hyperparameters (cf. Section 3.3. *similarity threshold* and *prefix threshold*), which can be set flexibly according to the data feature.

4.3. Robustness

In this part, we evaluate the robustness of log parsers on different datasets. Figure 3 shows the accuracy distribution of each log parser across the 16 log datasets in the boxplot. Each box has five horizontal lines from the bottom to the top, corresponding to the minimum, 25th percentile, median, 75th percentile, and maximum accuracy values, respectively. Diamond marks indicate outliers since LenMa only has an accuracy of 0.174 on HealthApp, and LogPunk gets 0.504 on Proxifier and 0.741 on Linux. Although LogPunk seems to perform poorly on Proxifier and Linux datasets, other methods do not work well either, and we have even achieved the highest accuracy on Linux.

For comparison, the log parser is arranged in ascending order of the average precision from left to right. We can observe that LogPunk has the highest average accuracy and the minimum variance, which means robustness. We also observed that Spell+ is more robust than Spell, and Drain+ and Drain have little difference in robustness. Because Drain has some strong assumptions, the preceding tokens of the same log type are the same. Such assumptions are the main reason for Drain's error in accuracy.

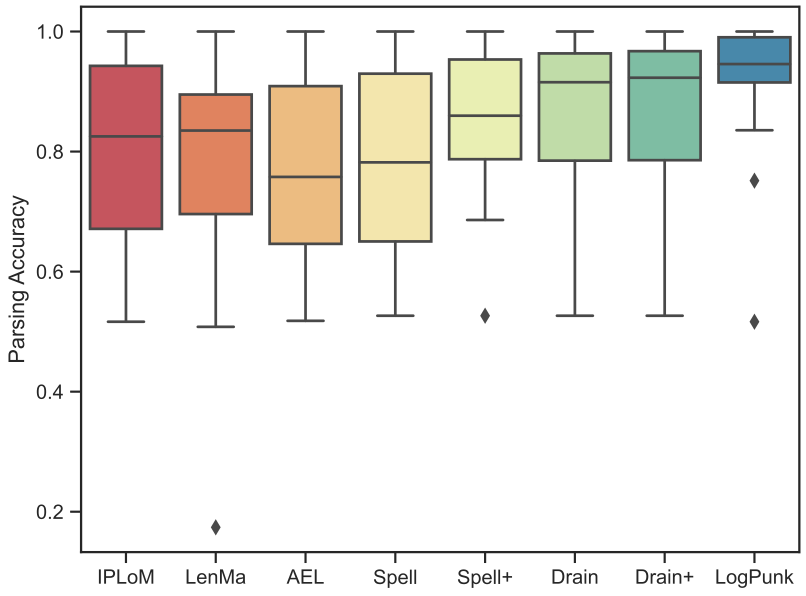

Figure 3. Accuracy distribution of log parsers across different types of logs.

4.4. Efficiency

With the increase in data volume, efficiency has become a vital attribute of log parsers. The running time of the whole parsing process is recorded to measure the efficiency of log parsers. In this experiment, we choose BGL and Android as the datasets. They are different types of systems and have been used in prior work [23,26]. We vary the volume from 1 M to 100 M for each dataset, and the hyperparameters are fine-tuned on 2 k log samples. We obtain different sample sizes by truncating the raw log files.

The results are presented in Figure 4. The parsing time increases with the raising of log size on both datasets. The efficiency of LogPunk is much better than other log parsers on the BGL dataset, but LogPunk is close to AEL on the Android dataset. Because there are many short log templates on the Android dataset, which lack punctuation features but have many log messages, such as"startAnimation end" "startAnimation begin". Such log templates will get the same signature, resulting in too many templates (The most one has 42 templates) in the same signature group. However, we search the signature group one by one, which will seriously affect the efficiency in such cases. Fortunately, inverted index [30] and prefix tree [26] can improve search efficiency.

Except for LenMa, the parsing time of all parsers increases linearly with the size of the dataset. LenMa cannot process 50 m log data files in ten minutes, so we remove it for comparison in the rest sizes.

We observe that the parsing time of Android differs from the BGL dataset, which means that the efficiency of a log parser also depends on the type of logs. Android needs more time than BGL. It is because Android contains more event templates and is more complex. In addition, we find that the Android dataset contains some noise log messages, which appear in the form of one or two Chinese words. These log templates will be crowded in the same signature group, resulting in low efficiency when searching signature groups.

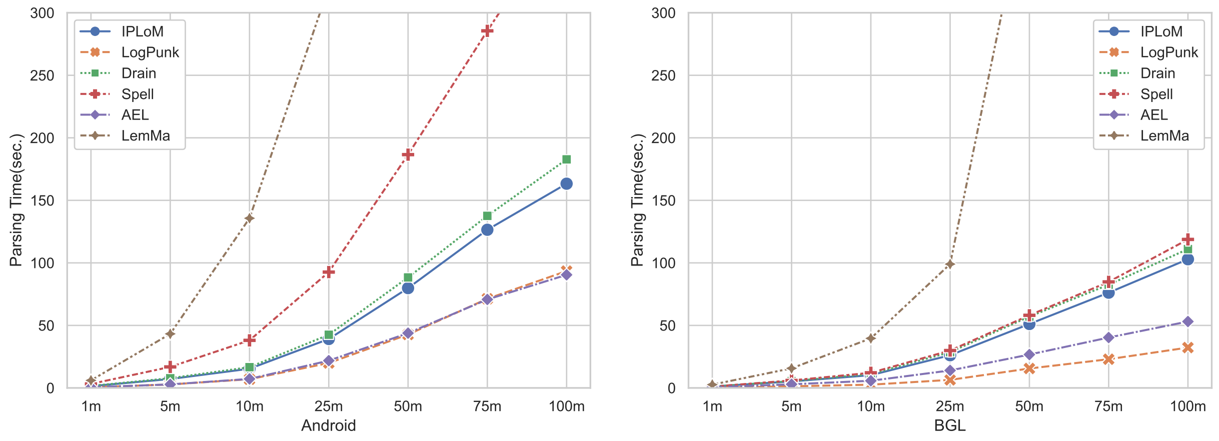

Figure 4. Parsing time of log parsers on different volumes of logs.

5. Discussion

In this section, we compare LogPunk with tree-based methods and discuss the validity.

Comparison with tree-based methods. As mentioned in Section 2, tree-based methods have two main problems: (1) needs to maintain a complex tree structure and updates the tree frequently as templates change. The updating process is time-consuming and complex; (2) does not guarantee the longest common subsequence (due to the limitation of the prefix tree itself). Unlike tree-based approaches, LogPunk works in a hash-like manner. Once a log signature is calculated, we can immediately locate the corresponding signature group. What we need to maintain is only a mapping relationship from the log signature to the signature group. This solution with a simple data structure brings more efficiency and accuracy improvements to LogPunk.

Moreover, the invariance of log signature also brings potential benefits of parallel computing. Since most of the current log parsing methods are single-threaded, multi-threaded log parsers will greatly improve the efficiency of log parsing.

External validity. LogPunk achieves a parsing accuracy of over 0.9 on more than half of the evaluated datasets. We cannot ensure that LogPunk can achieve the same high accuracy on other log datasets not tested in this work. However, it should also be noted that our evaluated log datasets come from systems in different domains. LogPunk is superior to other log parsers not only in accuracy, but also in efficiency. A future work is to verify the effectiveness of LogPunk on more types of log data and apply LogPunk to the practical log dataset from the production environment.

Internal validity. The internal threat to validity mainly lies in the implementations of LogPunk and compared approaches. To reduce this threat, the implementation of LogPunk was completely based on the most primitive Python library and did not use any third-party library. We have also carefully examined the source code. Regarding compared approaches, we adopted their open-source implementations from the LogPai benchmark [15] directly.

6. Related Work

Log parsing plays an important role in automatic log analysis and has been widely studied in recent years. Generally, log parsing approaches are divided into three categories: *rule-based*, *source code-based*, and *data-driven parsing* [17]. In this work, we focus on data-driven log parsing approaches. The advantage of these approaches is that they do not rely on application-specific knowledge heavily. In general, existing data-driven log parsing techniques could be grouped under three categories: *frequent pattern mining*, *clustering*, and *heuristics* [15].

(1) Frequent Pattern Mining: SLCT [19], LFA [20], and LogCluster [31] propose automated log parsers that parse log messages by mining the frequent tokens in log files. These approaches first count token frequencies and then use a predefined threshold to identify

the static parts of log messages. The intuition is that if a log event occurs frequently, then the static template parts will occur more times than the dynamic parts from variables. SLCT applies frequent pattern mining to log parsing for the first time. LFA utilizes the token frequency in each log message instead of the whole log data to parse infrequent logs. LogCluster improves SLCT and is robust to shifts in token position. All the above three methods are offline and need to traverse all log data to count the token frequency. In contrast, LogPunk is an online log parser.

(2) Clustering: Many previous studies regard log parsing as a clustering problem and propose many clustering approaches to solve this problem. From this perspective, log messages sharing the same templates are grouped into one cluster and various approaches to measure the similarity (or distance) between two log messages have been proposed. LKE [32], LogSig [21], and LogMine [22] propose offline clustering methods. LKE employs a k-means clustering algorithm based on weighted edit distance to extract log events from free text messages. LogSig groups log messages with the same frequent subsequence into a predefined number of clusters. LogMine clusters log messages from bottom to top and identifies the most suitable log template to represent each cluster.
SHISO [24] and LenMa [25] are both online methods. SHISO employs Euclidean distance to measure the similarity between logs and generate a score. If the score is smaller than the pre-defined threshold, SHISO makes a cluster of the similar two. LenMa proposes an online clustering method using the length information of each word in log messages. Additionally, it measured the similarity between two log messages based on cosine similarity, to determine which cluster the new coming log message should be added to. Despite performing well on test datasets, these two methods perform poorly on public datasets. To ensure robustness, LogPunk has been tested on 16 datasets from different systems.

(3) Heuristics: Different from general text data, log messages have some unique characteristics, which can be used for log parsing. AEL [29] uses heuristics based on domain knowledge to identify dynamic parts (e.g., tokens following "is" or "are") in log messages, then clusters log messages into the same template set if they have the same structure of dynamic parts. IPLoM [23] iteratively partitions log messages into finer clusters, firstly by the number of tokens, then by the position of tokens, and lastly by the association between token pairs. Spell [26] supposes that template tokens often take most of the log message, and variable tokens take only a small portion. So, it utilizes an LCS-based approach to measure log similarity and to find the most similar template. Drain [17] uses a fixed-depth tree to parse logs. Each layer encodes specially designed rules for log parsing. In the first layer, Drain searches by log message length, and in the following layers, searches by preceding tokens. By doing so, log messages with the same length and preceding tokens are clustered into the same groups placed on the leaf nodes. The tree-based Spell and Drain outperform other methods in the previous benchmark [15] and are state-of-the-art log parsers at present. LogPunk overcomes the defect of tree structures (cf. Section 2) and parses logs in a hash-like manner.

7. Conclusions

A qualified general-purpose online log parser should be robust and efficient, which meets the reliability requirements of modern software systems. At present, the state of the art online log parsers are tree-based. Such a structure brings low robustness and inefficiency. To overcome these limitations, we get inspiration from log punctuations and propose a hash-like method.

This paper proposed LogPunk, a robust and efficient log parser based on our novel log punctuations signature method. The candidate signature group is quickly located according to the log signature, which improves the efficiency. To solve the problem of signature collision, we design the log similarity function to find the most similar template, which improves the robustness.

Experiments are conducted on 16 public log datasets. Our experimental results show that LogPunk obtains the highest accuracy on ten datasets out of 16 datasets. Especially, LogPunk achieves the best average accuracy of 0.919 among the other five baseline log parsers. In addition, experiments also show that LogPunk is robust and efficient.

Finally, some future works are analyzed as follows.

(1) Automated parameters tuning. Logpunk has two hyperparameters *similarity threshold* and *prefix threshold* (cf. Section 3.3). During the experiment, these two hyperparameters are fine-tuned manually. This process is time-consuming, and the obtained hyperparameters may not be optimal. A mechanism for automated parameters tuning can greatly improve this situation.

(2) Punctuation table generation. The *punctuation table* (cf. Section 3.2) determines the log signature and affects the whole log parsing process. We presented a punctuation table by eliminating the punctuations appearing in variables, and it performs well on the 16 evaluated datasets. For a new system, if we customize a punctuation table for it, we may get better log parsing results. It is desirable to find a way to generate a customized punctuation table automatically for an unknown system.

(3) Variable type identification. Existing log parsers treat all variables in the parsing result as strings. However, obviously, each variable has its specific type information (e.g., number, IP, URL, file path, etc.) and it is useful to detect the variable-related anomaly. If log parsing not only identifies variable but also variable types, it will bring more initiative to downstream tasks.

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

Funding: This research received no external funding.

Data Availability Statement: Publicly available datasets were analyzed in this study. This data can be found here: [https://github.com/logpai/loghub].

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

Abbreviations

The following abbreviations are used in this manuscript:

AIOps	Artificial Intelligence for IT Operations
LCS	Longest Common Subsequence
PA	Parsing Accuracy
URL	Uniform Resource Locator
Symbols	
EV	Log Content
e_i	A Log Template
v_i	A List of Variables
E	The Set of All Log Templates
L	A Sequence of Log Messages
l_m	A Log Message
$\boldsymbol{t_i}$	A List of Tokens
t_j	The Token with Index j
T	The Set of All Tokens
n	Message Length

References

1. Cito, J.; Leitner, P.; Fritz, T.; Gall, H.C. The Making of Cloud Applications: An Empirical Study on Software Development for the Cloud. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2015; pp. 393–403. [CrossRef]

2. Barik, T.; DeLine, R.; Drucker, S.; Fisher, D. The bones of the system: A case study of logging and telemetry at microsoft. In Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering Companion, Austin, TX, USA, 14–22 May 2016; pp. 92–101. [CrossRef]
3. Forestiero, A.; Mastroianni, C.; Papuzzo, G.; Spezzano, G. A Proximity-Based Self-Organizing Framework for Service Composition and Discovery. In Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, Melbourne, VIC, Australia, 17–20 May 2010, pp. 428–437. [CrossRef]
4. Forestiero, A.; Mastroianni, C.; Meo, M.; Papuzzo, G.; Sheikhalishahi, M. Hierarchical approach for green workload management in distributed data centers. In *European Conference on Parallel Processing*; Springer: New York, NY, USA, 2014; pp. 323–334. [CrossRef]
5. Mi, H.; Wang, H.; Zhou, Y.; Lyu, M.R.T.; Cai, H. Toward Fine-Grained, Unsupervised, Scalable Performance Diagnosis for Production Cloud Computing Systems. *IEEE Trans. Parallel Distrib. Syst.* **2013**, *24*, 1245–1255. [CrossRef]
6. 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; Association for Computing Machinery: New York, NY, USA, 2019; pp. 807–817. [CrossRef]
7. 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. In Proceedings of the International Joint Conferences on Artificial Intelligence Organization, Macao, China, 10–16 August 2019; pp. 4739–4745. [CrossRef]
8. Zhou, X.; Peng, X.; Xie, T.; Sun, J.; Ji, C.; Liu, D.; Xiang, Q.; He, C. In *Latent Error Prediction and Fault Localization for Microservice Applications by Learning from System Trace Logs*; Association for Computing Machinery: New York, NY, USA, 2019; pp. 683–694. [CrossRef]
9. Chen, Y.; Yang, X.; Lin, Q.; Zhang, H.; Gao, F.; Xu, Z.; Dang, Y.; Zhang, D.; Dong, H.; Xu, Y.; et al. Outage Prediction and Diagnosis for Cloud Service Systems. In The World Wide Web Conference, San Francisco, CA, USA, 13–17 May 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 2659–2665. [CrossRef]
10. Zaman, T.S.; Han, X.; Yu, T. SCMiner: Localizing System-Level Concurrency Faults from Large System Call Traces. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering, San Diego, CA, USA, 11–15 November 2019; pp. 515–526. [CrossRef]
11. Cotroneo, D.; De Simone, L.; Liguori, P.; Natella, R.; Bidokhti, N. How Bad Can a Bug Get? An Empirical Analysis of Software Failures in the OpenStack Cloud Computing Platform. 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; Association for Computing Machinery: New York, NY, USA, 2019; pp. 200–211. [CrossRef]
12. 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. [CrossRef]
13. Lou, J.G.; Fu, Q.; Yang, S.; Xu, Y.; Li, J. Mining Invariants from Console Logs for System Problem Detection. In Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, 23–25 June 2010; pp. 1–14.
14. Lou, J.G.; Fu, Q.; Yang, S.; Li, J.; Wu, B. Mining program workflow from interleaved traces. In Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, 25–28 July 2010; pp. 613–622. [CrossRef]
15. 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, Montreal, QC, Canada, 25–31 May 2019; pp. 121–130.
16. Beschastnikh, I.; Brun, Y.; Ernst, M.D.; Krishnamurthy, A. Inferring models of concurrent systems from logs of their behavior with CSight. In Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May 2014; pp. 468–479. [CrossRef]
17. 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, Honolulu, HI, USA, 25–30 June 2017; pp. 33–40. [CrossRef]
18. Dai, H.; Li, H.; Chen, C.S.; Shang, W.; Chen, T.H. Logram: Efficient log parsing using n-gram dictionaries. *IEEE Trans. Softw. Eng.* **2020**. [CrossRef]
19. Vaarandi, R. A data clustering algorithm for mining patterns from event logs. In Proceedings of the 3rd IEEE Workshop on IP Operations & Management, Kansas City, MO, USA, 3 October 2003; pp. 119–126. [CrossRef]
20. Nagappan, M.; Vouk, M.A. Abstracting log lines to log event types for mining software system logs. In Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories, Cape Town, South Africa, 2–3 May 2010; pp. 114–117.
21. Tang, L.; Li, T.; Perng, C.S. LogSig: Generating system events from raw textual logs. In Proceedings of the 20th ACM International Conference on Information and Knowledge Management, Glasgow, UK, 24–28 October 2011; pp. 785–794.
22. Hamooni, H.; Debnath, B.; Xu, J.; Zhang, H.; Jiang, G.; Mueen, A. Logmine: Fast pattern recognition for log analytics. In Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, Indianapolis, IN, USA, 24–28 October 2016; pp. 1573–1582.

23. 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, Paris, France, 28 June 2009–1 July 2009; pp. 1255–1264.
24. Mizutani, M. Incremental mining of system log format. In Proceedings of the 2013 IEEE International Conference on Services Computing, Santa Clara, CA, USA, 28 June–3 July 2013; pp. 595–602. [CrossRef]
25. Shima, K. Length matters: Clustering system log messages using length of words. *arXiv* **2016**, arXiv:1611.03213.
26. 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; pp. 859–864. [CrossRef]
27. Du, M.; Li, F. Spell: Online Streaming Parsing of Large Unstructured System Logs. *IEEE Trans. Knowl. Data Eng.* **2019**, *31*, 2213–2227. [CrossRef]
28. 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.
29. 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. [CrossRef]
30. Huang, S.; Liu, Y.; Fung, C.; He, R.; Zhao, Y.; Yang, H.; Luan, Z. Paddy: An event log parsing approach using dynamic dictionary. In Proceedings of the NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, 20–24 April 2020; pp. 1–8.
31. Vaarandi, R.; Pihelgas, M. Logcluster—A data clustering and pattern mining algorithm for event logs. In Proceedings of the 2015 11th International Conference on Network and Service Management, Barcelona, Spain, 9–13 November 2015; pp. 1–7. [CrossRef]
32. Fu, Q.; Lou, J.G.; Wang, Y.; Li, J. Execution anomaly detection in distributed systems through unstructured log analysis. In Proceedings of the 2009 Ninth IEEE International Conference on Data Mining, Miami Beach, FL, USA, 6–9 December 2009; pp. 149–158. [CrossRef]