Computer Systems
Science and Engineering

Tech Science Press

DOI: 10.32604/csse.2023.037505
Article

An Efficient Way to Parse Logs Automatically for Multiline Events

Mingguang Yu[1,2] **and Xia Zhang**[1,2,*]

[1]School of Computer Science and Engineering, Northeastern University, Shenyang, 110169, China
[2]Neusoft Corporation, Shenyang, 110179, China
*Corresponding Author: Xia Zhang. Email: zhangx@neusoft.com
Received: 06 November 2022; Accepted: 06 January 2023

Abstract: In order to obtain information or discover knowledge from system logs, the first step is to perform log parsing, whereby unstructured raw logs can be transformed into a sequence of structured events. Although comprehensive studies on log parsing have been conducted in recent years, most assume that one event object corresponds to a single-line message. However, in a growing number of scenarios, one event object spans multiple lines in the log, for which parsing methods toward single-line events are not applicable. In order to address this problem, this paper proposes an automated **l**og **p**arsing method for **m**ultiline **e**vents (LPME). LPME finds multiline event objects via iterative scanning, driven by a set of heuristic rules derived from practice. The advantage of LPME is that it proposes a cohesion-based evaluation method for multiline events and a bottom-up search approach that eliminates the process of enumerating all combinations. We analyze the algorithmic complexity of LPME and validate it on four datasets from different backgrounds. Evaluations show that the actual time complexity of LPME parsing for multiline events is close to the constant time, which enables it to handle large-scale sample inputs. On the experimental datasets, the performance of LPME achieves 1.0 for recall, and the precision is generally higher than 0.9, which demonstrates the effectiveness of the proposed LPME.

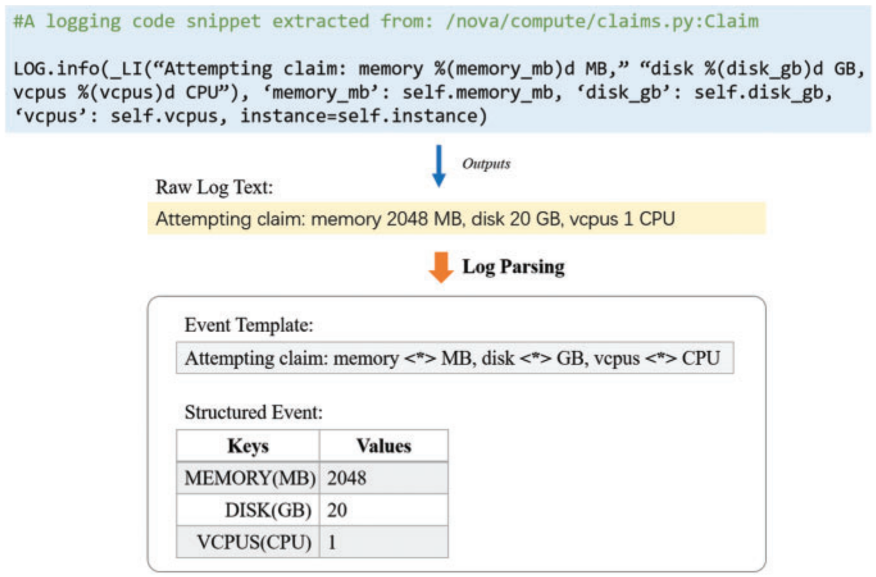
Keywords: Log parsing; log management; log analysis; system maintenance

1 Introduction

1.1 Background

Modern large-scale information systems continuously generate a substantial volume of log data. These data record the system running state, operation results, business processes, and detailed information on system exceptions. Thus, log analysis techniques have attracted considerable attention from researchers in the past decade. Many distinguished works have emerged, including detecting program running exceptions [1,2], monitoring network failures and traffic [3,4], diagnosing performance bottlenecks [5], and analyzing business [6] and user behavior [7].

Logs are printed by logging statements, such as “log.info (...)” or “print (...)” written by programmers. The contents and formats of logs are free because virtually no strict restrictions are placed on them while coding. However, structured input is required for most data mining models used in log analysis techniques. Therefore, the first step of log analysis is log parsing, in which unstructured log messages in plain text are transformed into structured event objects. Fig. 1 refers to some logging code in OpenStack’s sources as an example to illustrate the log printing and parsing process described above. In the log parsing shown in Fig. 1, a resource request event is obtained from the original log through log parsing.



```
#A logging code snippet extracted from: /nova/compute/claims.py:Claim

LOG.info(_LI(“Attempting claim: memory %(memory_mb)d MB,” “disk %(disk_gb)d GB,
vcpus %(vcpus)d CPU”), ‘memory_mb’: self.memory_mb, ‘disk_gb’: self.disk_gb,
‘vcpus’: self.vcpus, instance=self.instance)
```

Raw Log Text:

```
Attempting claim: memory 2048 MB, disk 20 GB, vcpus 1 CPU
```

Event Template:

```
Attempting claim: memory <*> MB, disk <*> GB, vcpus <*> CPU
```

Structured Event:

Keys	Values
MEMORY(MB)	2048
DISK(GB)	20
VCPUS(CPU)	1

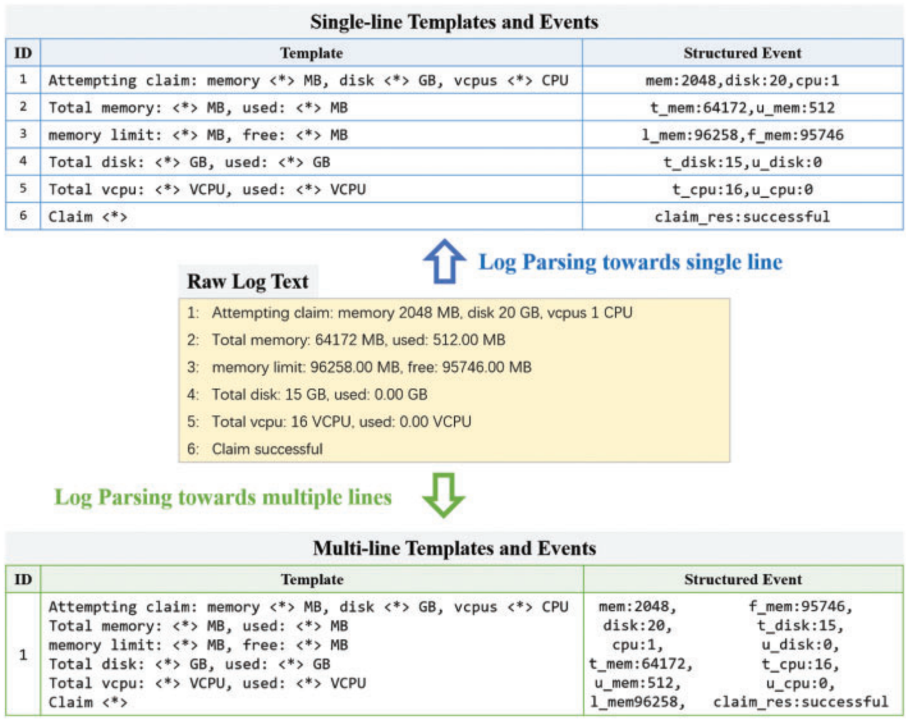
Figure 1: Illustrative example of log parsing

Traditional approaches to archive log parsing rely heavily on manually customized regular expressions. However, the rapid growth in log volume has brought unbearable labor and time costs. At the same time, artificial rules can hardly keep up with the frequent updates of modern software systems. For these reasons, many research efforts on automated log parsing techniques have emerged [8–13], which are dedicated to contributing automated log parsing methods to overcome the shortcomings of the manual method.

1.2 Motivation

With the rapid development of advanced technologies, such as cloud computing and the Internet of Things [14,15], the logs generated by modern information systems are becoming increasingly complex. In the log’s output by complex systems, it is common for multiple lines of text in a log to form an event object. That poses new challenges to log parsing, one of which is that existing log parsing efforts assume that an event maps to one text line in the log.

Fig. 2 compares single-line event parsing and multiline event parsing with an example. The yellow background color area in the middle of Fig. 2 shows a sample of raw logs; if parsed to single-line events, as shown by the upwards blue arrow in the figure, the six lines of raw log text will yield six separate event objects. However, these six lines of log text are strongly correlated and record a unit event together. If they are considered separately, fragmented information will hinder the subsequent analysis work. For example, the 6th line represents a claim. Suppose it is not associated with the context. In that case, the subsequent analysis will be confused by questions such as the specific parameters of the successful claim, how much memory is requested for the successful claim, and how many CPU cores are requested for the successful claim. However, if the raw log text is parsed correctly to a single multiline event, as shown by the downwards green arrow in the figure, the above problems will not arise. Therefore, it is essential to determine how to identify multiline events in complex logs.


Log Parsing towards single line
Log Parsing towards multiple lines

Single-line Templates and Events

ID	Template	Structured Event
1	Attempting claim: memory <*> MB, disk <*> GB, vcpus <*> CPU	mem:2048,disk:20,cpu:1
2	Total memory: <*> MB, used: <*> MB	t_mem:64172,u_mem:512
3	memory limit: <*> MB, free: <*> MB	l_mem:96258,f_mem:95746
4	Total disk: <*> GB, used: <*> GB	t_disk:15,u_disk:0
5	Total vcpu: <*> VCPU, used: <*> VCPU	t_cpu:16,u_cpu:0
6	Claim <*>	claim_res:successful

Raw Log Text

```
1: Attempting claim: memory 2048 MB, disk 20 GB, vcpus 1 CPU
2: Total memory: 64172 MB, used: 512.00 MB
3: memory limit: 96258.00 MB, free: 95746.00 MB
4: Total disk: 15 GB, used: 0.00 GB
5: Total vcpu: 16 VCPU, used: 0.00 VCPU
6: Claim successful
```

Multi-line Templates and Events

ID	Template	Structured Event
1	Attempting claim: memory <*> MB, disk <*> GB, vcpus <*> CPU Total memory: <*> MB, used: <*> MB memory limit: <*> MB, free: <*> MB Total disk: <*> GB, used: <*> GB Total vcpu: <*> VCPU, used: <*> VCPU Claim <*>	mem:2048, disk:20, cpu:1, t_mem:64172, u_mem:512, l_mem96258, f_mem:95746, t_disk:15, u_disk:0, t_cpu:16, u_cpu:0, claim_res:successful

Figure 2: The difference between single-line event parsing and multiline event parsing

To the best of our knowledge, there is no current research on log parsing that focuses on the problem of multiline event parsing. In order to address this problem, this paper innovatively proposes LPME, an automated **l**og **p**arsing method for **m**ultiline **e**vents. LPME is an iterative scanning algorithm based on the results of single-line text templates, and it employs a set of heuristic rules to identify multiline event objects.

1.3 Contributions

The contributions of this paper are summarized as follows: 1) We present the problem of multiline event parsing, which is explained based on practical experience. To address this problem, we design a multiline event-oriented parsing method called LPME. 2) We perform a thorough analysis of the complexity of the algorithm and conduct experiments on four datasets from different backgrounds to illustrate the effectiveness and feasibility of LPME.

The rest of the paper is organized as follows: Section 2 examines related research. Section 3 illustrates the details of LPME. Section 4 presents the experimental analysis. Finally, we conclude this paper and discuss future work in Section 5.

2 Related Work

Log analysis plays a vital role in service maintenance. Log parsing is the first step in automated log analysis [8]. Traditional methods of log parsing rely on handcrafted regular expressions or grok patterns to extract event templates. Although straightforward, manually writing ad hoc rules requires a deep understanding of the logs, and considerable manual effort is required to register different rules for various kinds of logs.

In order to reduce the manual effort devoted to log parsing, many studies have investigated automated log parsing [16]. Xu et al. [17] obtained log templates through system source code analysis; however, in most cases, the source code is inaccessible. Therefore, most existing automated methods favor data-driven approaches to analyzing the log data to obtain templates of the events. Data-driven log parsing techniques can be roughly classified into three main categories: frequent pattern mining, clustering, and heuristic rules [8].

Frequent pattern mining is used to discover frequently occurring line templates from event logs; this approach assumes that each event is described by a single line in the event log and that each line pattern represents a group of similar events. Simple logfile clustering tool (SLCT) [18] is the first log parser to utilize frequent pattern mining. Furthermore, LogCluster [19] is an extension of SLCT that is robust to shifts in token positions. Logram [9] uses n-gram dictionaries to achieve efficient log parsing; it parses log messages into static text and a dynamic variable by counting the number of appearances of each n-gram. Paddy [10] clusters log messages incrementally according to Jaccard similarity and length features. It uses a dynamic dictionary structure to search template candidates efficiently.

The second category is cluster-based approaches, which formulate log parsing as a clustering problem and use various techniques to measure the similarity and distance between two log messages (e.g., log key extraction (LKE) [20], LogSig [21], LogMine [22], and length matters (LenMa) [23]). For example, LKE employs a hierarchical clustering algorithm based on the weighted edit distance between pairwise log messages.

The last category is heuristic approaches, which perform well in terms of accuracy and efficiency. Compared to general text data, log messages have some unique characteristics. In addition, some methods use heuristics to extract event templates. For example, Drain [24] used a fixed-depth tree to represent the hierarchical relationship between log messages. Each tree layer defines a rule for grouping log messages (e.g., log message length, preceding tokens, and token similarity). Iterative partitioning log mining (IPLoM) [25] applies an iterative partition strategy to partition log messages into groups according to the token amount, token position, and mapping relation. Abstracting execution logs (AEL) [26] groups logs by comparing the occurrence times of constants and variables and then obtains log templates if they have the same static components.

In addition to the above three categories, other data-driven methods exist. For example, Spell [11] proposed an online log parsing method based on the longest common subsequence (LCS) and can dynamically extract real-time log templates. In contrast, reference [12] built a graph in which each node represents a log message and clusters logs according to the word count and Hamming similarity. LogParse [13] is a novel method that transforms the log parsing problem into a word classification problem.

Unfortunately, the above research assumes that one event object corresponds to a single text line. However, as mentioned in Section 1.2, a single event often corresponds to multiple text lines in many complex logs. To the best of our knowledge, there has been no research work on multiline event parsing.

Therefore, in this paper, we propose multiline event parsing and design a solution based on existing work to fill this gap.

3 Methods

In this section, we first present the main process of LPME and then describe each sub-step separately before finally analyzing the algorithm complexity.

3.1 Algorithm Overview

Regarding whether the problem of multiline event parsing can be solved by simply generalizing existing single-line event-oriented parsing methods, our answer is no. The underlying reason for our response is that multiline event parsing implies a sub-problem of line division. An "event" can be composed of x lines of text, where x is indeterminate. Therefore, based on previous related research, this paper proposes LPME, a log parsing method for multiline events. Fig. 3 illustrates the LPME framework. LPME is a two-stage process with phases: "Phase-1: Parsing Templates for Single Lines" and "Phase-2: Discovering Templates of Multiline". The initial input to LPME, i.e., the "Raw Logs Sample," is a continuous n-line sample taken from the original log. Phase 1 could process the "Raw Logs Sample" by employing any existing single-line parsing method, such as AEL [26] or IPLoM [25], and even online parsing methods, such as Drain [24] or Spell [11]. LPME treats the output of Phase 1 as the intermediate result, which is input into Phase 2 to obtain the multiline templates. Phase 1 can utilize existing methods. Thus, this paper focuses on Phase 2.

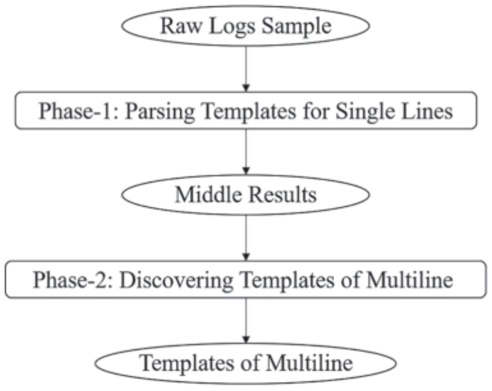

Raw Logs Sample
Phase-1: Parsing Templates for Single Lines
Middle Results
Phase-2: Discovering Templates of Multiline
Templates of Multiline

Figure 3: Framework of LPME

For convenience, we use S to represent the "Middle Results" in Fig. 3. S is the output of Phase 1 and serves as the input to Phase 2. S can be assumed to be a sequence of n length, each element corresponding to the original n lines of the "Raw Logs Sample" with a one-to-one mapping. Each element includes the original text line and the single-line template. If sliding windows of size 2, 3, 4, . . . , n are used to scan S with a step size of 1 and the window fragments obtained during the scan are collected, then $n(n\text{-}1)/2$ window fragments can be obtained. We use a set $W = \{w|\ w$ represents all possible sliding window fragments of size 2 to $n\}$ to describe the result obtained via the above sliding scan. An element w in W has three key attributes: 1) the single-line template sequence, *subseq*, captured by the sliding window, 2) the start timestamp ts_s of the window sequence, and 3) the end timestamp ts_e of the window sequence. W can be represented in triplet form as $<subseq,\ ts_s,\ ts_e>$. To collect the *subseq* of all elements of W and remove the duplicates, we obtain the set SET_{seqs}. The focus of this paper is to find all *subseq* in SET_{seqs} that are indeed multiline events. The final results can be organized into a hash table R, of which the range of keys is $\{k \mid k$ is an integer and $2 \leq k \leq n\}$. $R[k]$ is a set, and each element represents a k-line event template. The k-line event template can be expressed as $template_k{}^i$ $(0 \leq i \leq R[k].size)$. $template_k{}^i$ records the corresponding *subseq* and some attached parameters to represent a multiline event template.

From a practical perspective, logs requiring multiline template analysis usually originate from complex systems, often as big data. In practice, we also tend to draw large sample sizes. Therefore, S may be a huge sequence. For example, its length may be on the order of 10^4 or even 10^5. In this case, if full scans collect R, then the time consumption of the task will be unbearable. Therefore, we must consider the actual situation to develop optimization methods to improve the efficiency of algorithm execution so that the final approach applies to production.

Experience from system log analysis indicates that the single-line templates constituting a multiline event are typically different from other ordinary single-line templates; they generally appear only in multiline events. Therefore, the multiline event template manifests as a cohesive-sequence template, with a priori characteristics similar to the frequent itemset. As a result, if the sequence *subseq* is not part of a multiline event, then the superset of *subseq* will not be part of a multiline event either. Conversely, if *subseq* is indeed a component of a multiline event, then any subset of *subseq* must also be a component of that multiline event. Moreover, we can obtain the corollary that if any subset of *subseq* is not a component of any multiline event, then *subseq* is not either.

Based on the above reasoning, LPME is designed to conduct an iterative layer-by-layer search. The mining results can be organized into a hash table R. The idea guiding the search is that $R[K+1]$ must grow from the pre-result $R[k]$; that is, $R[k]$ is fundamental for the derivation and evaluation of $R[K+1]$.

Next, we detail how to evaluate the candidate templates in the iterative process. To answer this question, we define some preliminary concepts. Let $template_k{}^i$ be the template to be evaluated. $template_k{}^i$ is collected by a sliding window of length k, and $template_k{}^i$ represents an element of set W. In $template_k{}^i$, the single-line sequence *subseq* consists of s_j $(j = 1, 2, \ldots, k)$. We define the cohesion support for $template_k{}^i$ by component s_j as Eq. (1).

$$CohesionSup\left(template_k^i, s_j\right) = \frac{Total\ Occurrence\ of\ s_j\ in\ R\left[k\right]}{Total\ Occurrence\ of\ s_j\ in\ S} \quad (1)$$

In turn, we can obtain the cohesion coefficient of $template_k{}^i$ according to Eq. (2).

$$CohesionCoef\left(template_k^i\right) = \frac{\sum_{j=1}^{n} CohesionSup\left(template_k^i, s_j\right)}{n} \quad (2)$$

Based on the above definition, the main criterion for evaluating candidate $template_k{}^i$ can be based on the cohesion coefficient. The core logic is that the higher *CohesionCoef* is, the more likely $template_k{}^i$ will be treated as a multiline event template.

Fig. 4 presents the overall multiline template discovery process, performed layer-by-layer. The process starts with the search and evaluation of the templates in layer $k = 2$, and then the search and evaluation of templates in layer k $(k > 2)$ can be executed based on layer $k-1$. The final merge step is a bottom-up merge of the preliminary results of the iterative growth to eliminate redundancy due to possible inclusion relationships between adjacent layers.

The core step shown in Fig. 4 is "Search for R[k]," which contains two branches, including cases $k = 2$ and $k > 2$. Each branch includes two similar steps, "Collect the potential templates" and

"Evaluate and filter the templates." However, there are subtle differences. The first difference is in the collection method. No previous works consider the case where $k = 2$, and the collection of items to be evaluated involves all the original adjacent s_j, which requires a complete traversal of S. When $k > 2$, there is no need to completely traverse S since the $k-1$ layer results are already available. The results are directly expanded to k layers with some specific strategies based on the results of the $k-1$ layers. The second difference is in the evaluation method. When $k = 2$, the cohesion support must be calculated twice separately to obtain the cohesion coefficient, while if $k > 2$, there is no need to calculate each cohesion support again because the cohesion coefficients of the $k-1$ layer results are already obtained. Thus, only one calculation is needed for the growing component.

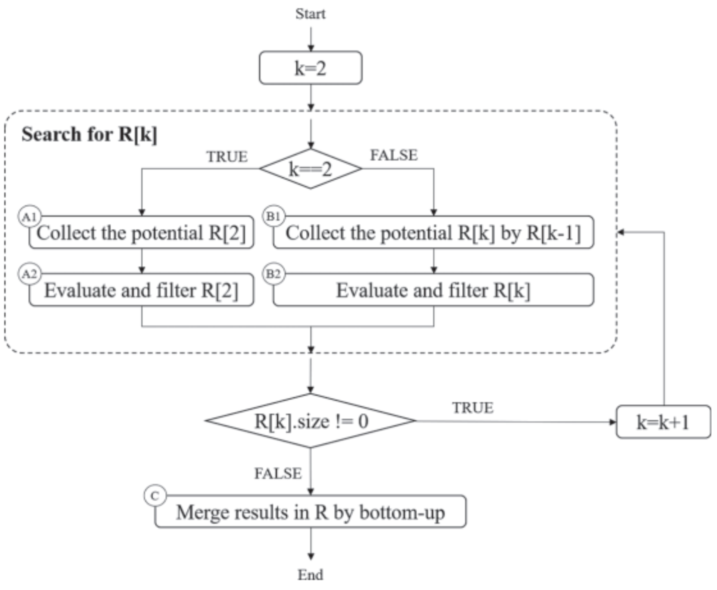

Start
k=2
Search for R[k]
TRUE
k==2
FALSE
A1 Collect the potential R[2]
B1 Collect the potential R[k] by R[k-1]
A2 Evaluate and filter R[2]
B2 Evaluate and filter R[k]
R[k].size != 0
TRUE
k=k+1
FALSE
C Merge results in R by bottom-up
End

Figure 4: Overall process for multiline template discovery

3.2 *Detail Processes*

This section details the crucial steps of the "Search for $R[k]$" module shown in Fig. 4.

Step-A1: Collect the potential templates belonging to $R[2]$ to prepare for the next step of evaluation and screening.

As mentioned previously, $R[2]$ is a special stage result in the iterative process, which is the starting point of the whole iterative exploration. Compared with iterations in the higher layers, the initial collection and determination of $R[2]$ results are slightly different. A more detailed explanation of this step is provided in Algorithm 1, which details the process of scanning S. In Algorithm 1, a control parameter, MAX_WINODW_TS_SPAN, is used in the scan to determine whether there is a possibility that two adjacent line templates can join into a two-line template. MAX_WINODW_TS_SPAN limits the maximum time span between the start and end rows in a multiline event template. This logic is easy to understand: usually, there is no substantial delay between steps for a single multiline event. According to our experience, MAX_WINODW_TS_SPAN should be set to 3–5 s.

Algorithm 1: Collect the candidate *template*$_2$

```
Require: $S$; Maximum time span of multiline event MAX_WINODW_TS_SPAN
Ensure: $ct_2set$ - Set of the candidate $template_2$
1: initialize an empty $ct_2set$ to store candidate templates
2: $i = 0$
3: while $i < S$.length-1 do
4:     $s_i = S[i]$
5:     $s_{i+1} = S[i + 1]$
6:     if $s_{i+1}$.timestamp - $s_{i+1}$.timestamp $\leq$ MAX_WINODW_TS_SPAN then
7:         get $template_2^{(s_i,s_{i+1})}$ from $ct_2set$, if not exists then construct one and put it into $ct_2set$
8:         $template_2^{(s_i,s_{i+1})}$.occurrence $++$
9:     end if
10: end while
11: return $ct_2set$
```

Step-A2: Evaluate the candidate set obtained from Step-A1 and acquire the filtered $R[2]$.

A detailed description of this step is provided in Algorithm 2. The input of Step-A2 is *ct*$_2$*set*, which is the output of Step-A1. Since the parameters required for computing the cohesion coefficient are available after Step-A1, Step-A2 no longer needs to access S again but only needs to traverse *ct*$_2$*set*.

Two necessary conditions exist in the evaluation process. One condition is whether the occurrence of a multiline template is above the threshold MIN_OCCURRENCE, which means those general events should have a certain repeatability. If the candidate's statistical occurrence obtained in the preliminary collection is lower than MIN_OCCURRENCE, it should be screened. The other condition is the cohesion coefficient of the multiline templates. Because the higher CohesionCoef is, the more likely it is that the candidate template will be treated as a real multiline event template when evaluating multiline event template candidates, MIN_COHESION_COEF is used to screen the multiline candidates. Since the calculation of the cohesion coefficient for two-line templates does not have a prior basis, we must perform calculations for each component's cohesion support of the two constituent elements. After obtaining the cohesion coefficient, the candidate templates can be filtered according to MIN_COHESION_COEF to obtain the $R[2]$ results.

Algorithm 2: Evaluate and filter the *ct*$_2$*set* to get $R[2]$

```
Require: $ct_2set$; Minimal occurrences of multiline event MIN_OCCURRENCE; Minimal CohesionCoef of multiline event MIN_COHESION_COEF
Ensure: $R[2]$
1: for each $template_2^i$ in $ct_2set$ do
2:     if $template_2^i$.occurrence < MIN_OCCURRENCE then
3:         remove $template_2^i$ from $ct_2set$
4:         continue
5:     end if
6:     $CohesionCoef\left(template_2^i\right) = \frac{CohesionSup\left(template_2^i, s_1\right) + CohesionSup\left(template_2^i, s_2\right)}{2}$
7:     if $CohesionCoef\left(template_2^i\right) \leq$ MIN_COHESION_COEF then
8:         remove $template_2^i$ from $ct_2set$
```

(Continued)

Algorithm 2: Continued

```
9:         continue
10:   end if
11: end for
12: $R[2] = ct_2set$
```

Step-B1: Collect all candidate templates that may belong to $R[k]$ ($k > 2$) based on $R[k-1]$ to prepare for the next step of evaluation and selection.

As mentioned previously, multiline template discovery explores $R[k]$ based on $R[k-1]$. Step-A1 and Step-A2 can be regarded as initialization steps. Based on $R[2]$, the subsequent iterations already have the starting conditions. Like the process of collecting and evaluating $R[2]$, each subsequent iteration involves collecting and evaluating the results of the current layer for filtering. The detailed process of the initial collection of $template_k$ in the $R[k]$ ($k > 2$) layer is described in Algorithm 3.

Algorithm 3: Collect all of the candidate $template_k$ based on $R[k\text{-}1]$ ($k > 2$)

```
Require: $S$; $R[k-1]$
Ensure: $ct_kset$
1: Initialize an empty $ct_kset$ to store candidate templates
2: $i = 0$
3: for each pair $\left(template^p_{k-1}, template^q_{k-1}\right)$ in the $R[k\text{-}1]$ do
4:    if $template^p_{k-1}.subseq[1: last] == template^q_{k-1}.subseq[0: last-1]$ then
5:         $nSubseq = template^p_{k-1}.subseq$.append($template^q_{k-1}.subseq[last]$)
6:         construct $template_k$
7:         $template_k.subseq = nSubseq$
8:         $template_k.CohesionCoef = template^p_{k-1}.CohesionCoef$
9:         put $template_k$ into $ct_kset$
10:   end if
11: end for
12: return $ct_kset$
```

Compared to the collection of ct_2set in Step-A1, Step-B1 has support from the lower layer $R[k-1]$. Therefore, a proper growth strategy can replace the full traversal of S, which significantly improves execution efficiency.

Step-B2: Evaluate the ct_kset output from Step-B1 to obtain the filtered $R[k]$ ($k > 2$) result.

In contrast to the evaluation for $R[2]$, we do not need to compute the cohesion support for each component line of the candidate template in ctkset. Since the $R[k-1]$ result is already determined, for the ctkset candidate template, we only need to compute the cohesion support of the last grown line based on the $k-1$ layer. A detailed description of this sub-process is provided in Algorithm 4.

Algorithm 4: Evaluate and filter the ct_kset to obtain $R[k]$ result

```
Require: $ct_kset$; $S$; Maximum time span of multiline event MAX_WINODW_TS_SPAN;
Minimal CohesionCoef of multiline event MIN_COHESION_COEF;
Ensure: $R[k]$;
1: for each $template_k$ in $ct_kset$ do
2:       $t = template_k[last]$
```

(Continued)

Algorithm 4: Continued

```
3:   $t\_occurrences$ = accumulate the number of occurrences of t in the S
4:   $p\_occurrences$=accumulate the number of occurrences of $template_k.subseq$ (within MAX_WINODW_TS_SPAN) in the S
5:   $t\_CohesionSup = t\_occurrences/p\_occurrences$
6:   $template_k.CohesionCoef = (template_k.CohesionCoef \times (k-1) + t\_CohesionSup)/k$
7:   if $template_k$.CohesionCoef < MIN_COHESION_COEF then
8:       remove $template_k$ from $ct_kset$
9:   end if
10: end for
11: $R[k] = ct_kset$
```

Step-C: Review result R to eliminate redundant items.

Since R is obtained by layer-by-layer growth, there may be situations that higher-layer templates contain lower-layer templates. Therefore, R must be reviewed after its initial acquisition to eliminate redundancy. The situation may be complicated, such as assuming $R[3]$ contains $s_u s_v s_w$ and $R[4]$ contains $s_u s_v s_w s_x$ and $s_u s_v s_w s_y$. The correct result has two possible orientations: 1) eliminating $s_u s_v s_w$ in $R[3]$ and keeping only the two templates in $R[4]$; 2) $s_u s_v s_w$ in $R[3]$ has the reasons to be kept independently, so it is necessary to keep all three templates. To accurately eliminate the redundancy, R must be reviewed from the bottom up. The detailed process is described in Algorithm 5.

Algorithm 5: Merge result from $R[k]$ to $R[k+1]$

```
Require: R; GROWTH_FACTOR
Ensure: filtered $R$;
1: $k = 2$
2: while $k < R$.size do
3:   for each $template_k^i$ in $R[k]$ do
4:     count = 0
5:     for each $template_{k+1}^j$ in $R[k+1]$ do
6:       if $template_{k+1}^j$ extends from $template_k^i$ then
7:         $count = count + template_{k+1}^j.occurrences$
8:       end if
9:     end for
10:    if $abs(count - template_k^i.occurrences)/template_k^i.occurrences <$ GROWTH_FACTOR then
11:      remove $template_k^i$ from $R[k]$
12:    end if
13:  end for
14:  $k = k + 1$
15: end while
```

Algorithm 5 describes the process of eliminating possible redundancies arising from adjacent entailment relations from R. Specifically, for some template in the lower layer of two adjacent layers, the total number of occurrences of the higher-level templates extending from it is calculated, and the redundancy of the targeted lower-layer template is determined by comparing the calculated result with its occurrence number. If the two occurrence numbers are close, the targeted lower-layer template is determined to be redundant, and if not, it is kept as a valid result. A threshold parameter, GROWTH_FACTOR, is used to control the trade-off when making interlayer redundancy judgments. Notably, the algorithm can benefit from support information obtained from the preorder steps to determine whether the inclusion relation is true. With Step C, the redundant subsequences in R will be eliminated from the low to the high layers.

3.3 Algorithm Parameters

In this subsection, several control parameters involved in LPME are briefly described. LPME has four critical control parameters that make it more applicable. Users can configure these parameters with reasonable values for different context environments to drive LPME to produce optimal results. These four parameters belong to two categories: those used to control the capacity of the sliding window, namely, MAX_WINODW_TS_SPAN, and those used to adjust the evaluation criteria to filter the initial collection of multiline template candidates, namely, MIN_OCCURRENCE, MIN_COHESION_COEF, and GROWTH_FACTOR.

MAX_WINODW_TS_SPAN limits the maximum time span between the start and end rows in a multiline event template. Usually, there is no substantial printing latency between the multiple lines recording the same event. According to our experience, MAX_WINODW_TS_SPAN should be set to 3–5 s.

In order to filter out incorrect multiline template candidates, a frequency threshold must be defined according to field experience. This threshold is named MIN_OCCURRENCE. If the evaluated items have a lower frequency than MIN_OCCURRENCE, they will be filtered out because they are not sufficiently representative.

Because of the logic that the higher $CohesionCoef$ is, the more likely it is that the candidate template will be treated as a real multiline event template, when evaluating multiline event template candidates, the most crucial evaluation indicator is the value of $CohesionCoef$. According to the definition in Eq. (2) in Section 3.1, the range of the cohesion support is [0.0, 1.0]. The larger the cohesion support value of the evaluated multiline event, the more likely it is to be true. MIN_COHESION_COEF is used to screen the multiline candidates. However, according to practical experience, the ideal value of 1.0 is not suitable for use as the real evaluation criterion. Noise interference is inevitable in the production environment. Additionally, the discovery of multiline templates is based on the recognition of single-line templates, and that baseline is usually not 100% exact. Therefore, MIN_COHESION_COEF should be set to a value less than but close to 1.

In the iterative layer-by-layer search to obtain R, since the judgment input of layer $k + 1$ originates from layer k, once the iteration of layer $k + 1$ is complete, it is necessary to check whether there are redundant templates within these two adjacent layers. For example, consider the case where there is one template $s_u s_v s_w$ at layer three and two templates $s_u s_v s_w s_x / s_u s_v s_w s_y$ at layer 4. There is a trade-off for the above case: whether $s_u s_v s_w$ is retained in the final result. We introduce the GROWTHTH_FACTOR parameter, which takes a value in [0, 1], to control the trade-off when making interlayer redundancy judgments. The smaller the GROWTHTH_FACTOR is, the more likely it is that only the low-layer results are kept; the larger the GROWTHTH_FACTOR is, the more likely it is that only the high-layer results are extended.

3.4 Complexity Analysis

Based on the explanation of Step-A1 and Step-A2 in Section 3.2, in the computing process for R [2], one full scan of S is performed using a sliding window of size 2, and a total of (n–1) calculations are performed. The subsequent computations for $R[k]$ ($k > 2$) do not require another full scan of S

but are based on the results of $R[k-1]$, which grows by matching the prefixes and suffixes between each pair in $R[k-1]$. For example, a multiline template, *template*$_5$, with a length of five, is computed as shown in Fig. 5. The figure shows five layers, corresponding to 5 iterations of the layer-by-layer growth calculation. Each dashed box at every layer in Fig. 5 is an operation of matching and evaluating. The operation can benefit from maintaining the indexes on some required information, including the template prefixes and suffixes, as well as the positions and counts of s_i in S so that the complexity of the matching and evaluation process can be considered as constant time with order $O(1)$. Therefore, in Fig. 5, the computation numbers for the $<s_1s_2s_3s_4s_5>$ template are exactly the number of dashed boxes; in this case, $1 + 2 + 3 + 4 = 10$. Generally, for a multiline template, *template*$_l$, with length l, the number of computations required is $1 + 2 + 3 + \cdots + (l\text{-}2) + (l\text{-}1) = l \times (l\text{-}1)/2$. Therefore, the complexity of LPME is not related to the scale of the input S but is related to the scale of the output result, including the length l of the multiline template and the number of types of multiline templates v. In summary, the time complexity of LPME can be expressed as $\sum_{i=1}^{v} l_i \times (l_i - 1)/2$. From the practical application perspective, the length l of the final multiline template is much smaller than the length n of S, and in most cases, it is at most a few dozen lines. In addition, the number of multiline event types v contained in a batch of samples is generally approximately a few dozen. Therefore, the time complexity of the actual execution is on the order of approximately 10^2. Moreover, the length of the input S is n, and n is often on the order of 10^4 or 10^5. Thus, the time complexity of the actual execution is far less than $O(n)$. If l and v are predictable, it is closer to a constant time.

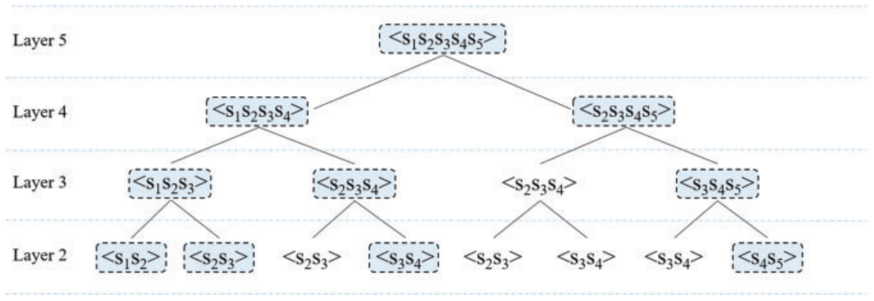


Figure 5: The computation for a multiline template *template*$_5$

Concerning space complexity, the total space complexity depends on the size of the intermediate results generated during the traversal process. The intermediate results are the multiline template candidates. For simplicity's sake, each template line is treated as a storage unit, and the space complexity is calculated based on the number of multiline template candidates. The number of multiline template candidates corresponds to the number of dashed boxes in Fig. 5. The result is $\sum_{i=1}^{v} l_i \times (l_i - 1)/2$. Therefore, similar to the time complexity, the size of the space occupied by the algorithm is at the same level as the number of final multiline templates present in the input sample. The number of multiline templates in an input sample is extremely limited, generally from a dozen to several dozen. In summary, the space complexity of LPME, on average, is close to the constant space complexity.

4 Evaluation

In this section, we perform an experimental evaluation of LPME on four real datasets. The experimental results corroborate the theoretical analysis and demonstrate the effectiveness of LPME.

4.1 Experimental Setting

The datasets used in our experiments are outlined in Table 1. The essential information of the datasets is as follows.

1) Windows OS logs are generated by the component-based servicing (CBS) module of Microsoft Windows. The logs record various components' loading, updating, and unloading processes.
2) OpenStack logs are generated by OpenStack. The logs record the running status of OpenStack. OpenStack is an open-source cloud computing management platform that manages and controls many computing, storage, and network resources in data centers and provides cloud hosts.
3) HealthApp logs are generated by a mobile application named HealthApp. The logs trace the running status of this app.
4) Payment System logs are from the payment system of a commercial bank. These logs record the traces of service calls in the distributed system.

Table 1: Summary of the experimental datasets

Log source	Lines	Data size	Time	Accessibility
Windows OS	35,040	4.79 MB	11 days	Public
OpenStack	52,312	14.7 MB	6 h	Public
HealthApp	253,395	22.4 MB	10 days	Public
Payment system	308,388	38.3 MB	48 h	Private

Datasets 1–3 are publicly available. They are obtained from Zhu et al. [8], who reviewed the research in the field of automated log parsing and summarized the datasets used in previous research, from which we select three representative logs from an operating system, middleware, and an application. In addition to the three public datasets, we test LPME with logs from a payment system of a commercial bank, with whom we cooperate on a log-driven artificial intelligence for IT operations (AIOPS) project. This dataset cannot be disclosed due to confidentiality. These logs record traces of service calls in a distributed system. The trace of an entire payment transaction consists of multiple single-line logs. Compared to the three publicly available datasets, this dataset is complex. After a manual review, we obtain the number of single-line and multiline templates in these four datasets and list them in Table 2. We published the multiline templates of public datasets online [27].

Table 2: The statistics of the event templates in the test datasets

Log source	Number of single-line templates	Number of multiline templates
Windows OS	50	2
OpenStack	43	5
HealthApp	75	2
Payment system	104	23

The settings of the experimental parameters are presented in Table 3. MAX_WINODW_TS_SPAN, MIN_OCCURRENCE, MIN_SUPPORT, and GROWTH_FACTOR are the control parameters of the algorithm, of which the values come from the empirical optimization results.

INPUT_SAMPLE_SIZE is the size of the input sample. This parameter should be set appropriately to ensure that the sample contains as many multiline event types as possible.

Table 3: The experimental parameters

Parameters	Experimental datasets			
	Windows OS	OpenStack	HealthApp	Payment system
MAX_WINODW_TS_SPAN	3	3	3	3
MIN_OCCURRENCE	50	60	600	600
MIN_SUPPORT	0.9	0.95	0.98	0.98
GROWTH_FACTOR	0.01	0.02	0.02	0.02
INPUT_SAMPLE_SIZE	2,000	2,000	2,000	5,000

We use the F1-score to evaluate the effectiveness of LPME. The F1-score is pervasively used in clustering and retrieval algorithm evaluation [28]. Its definition is shown in Eq. (3).

$$F1 - Score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{3}$$

The definitions of precision and recall are given in Eqs. (4) and (5), respectively.

$$Precision = TP/(TP + FP) \tag{4}$$

$$Recall = TP/(TP + FN) \tag{5}$$

In the above formula, TP (true positive) represents the correct induction of multiline events; FP (false positive) represents erroneous inductions that are not real multiline events; FN (false negative) represents results that should be multiline events but are not recognized. The statistics in Table 2 are used as the ground truth to calculate the F1 score.

To the best of our knowledge, no other work on automated parsing of multiline events has been published. Therefore, no other algorithms have the same purpose for comparative analysis. Thus, the experimental work focuses on the effectiveness of LPME for different types and sizes of logs. Theoretically, LPME is not dependent on pre-parsing for single-line templates, and any available single-line template parsing algorithm can be applied. To verify this assumption, for the pre-parsing of single-line logs in LPME, we use AEL [26], IPLoM [25], Drain [24], and Spell [11]. For the implementation of these algorithms, we referenced the published code (available online at [https://github.com/logpai/logparser]) from [8]. We code the program in Java 8. All the experiments are conducted on a server with an Intel(R) Core(TM) i5-8250U CPU @ 1.6 GHz, 1.80 GHz, 16 GB RAM, and Windows 11 installed.

4.2 Effectiveness

We divided the experiments into four groups according to the pre-parsing method, and each group was validated on four datasets. The evaluation results are provided in Table 4. As seen from the resulting data in Table 4, the F1 score of most of the experiments exceeds 0.9. Each group mainly obtained individual lower F1 scores for the OpenStack dataset because the total number of multiline event types in the OpenStack dataset is only five. When one or two false-positive results appear in the parsing result, it will considerably affect the precision. Therefore, for the above case, the effectiveness of LPME is sufficient to achieve the target of assisting manual recognition of multiline events.

According to the cross-group comparison of the results, different pre-parsing algorithms will not lead to conspicuous differences in the effectiveness of subsequent multiline event parsing. The prerequisite for correct parsing of multiline events is the accuracy of the pre-parsing, especially the accuracy of extracting the single-line templates contained in the multiline events. LPME is independent of the selected technical route of the pre-parsing algorithm. In our experiments, all four pre-parsing algorithms can correctly identify the single-line templates composing the multiline events in each dataset. However, for the OpenStack dataset, the precision is lower because some false multiline templates are misidentified.

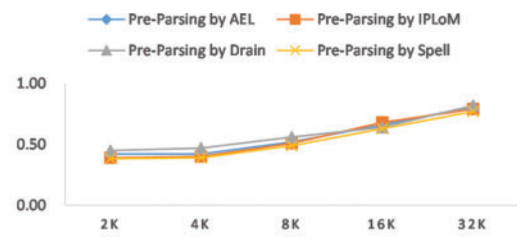
Overall, LPME acquires acceptable results on different datasets that are sufficient to assist humans in identifying multiline events efficiently and correctly.

Table 4: The evaluation results. The experiments are divided into four groups according to the pre-parsing method, and each group is validated on four datasets

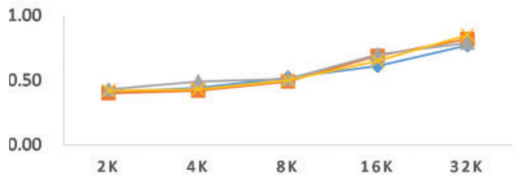
Group no.	Pre-parsing methods	Datasets	F1	Precision	Recall
1	AEL	Windows System	1.00	1.00	1.00
		OpenStack	0.83	0.71	1.00
		HealthApp	1.00	1.00	1.00
		Payment System	0.94	0.88	1.00
2	IPLoM	Windows System	1.00	1.00	1.00
		OpenStack	0.91	0.83	1.00
		HealthApp	1.00	1.00	1.00
		Payment System	0.96	0.92	1.00
3	Drain	Windows System	1.00	1.00	1.00
		OpenStack	1.00	1.00	1.00
		HealthApp	1.00	1.00	1.00
		Payment System	0.94	0.88	1.00
4	Spell	Windows System	1.00	1.00	1.00
		OpenStack	0.91	0.83	1.00
		HealthApp	1.00	1.00	1.00
		Payment System	0.98	0.96	1.00

4.3 Performance and Scalability

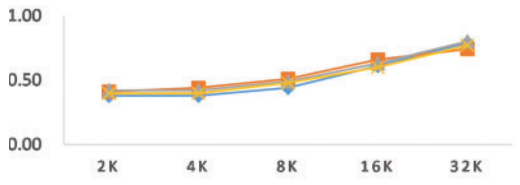
In the big data era, the log volume is continuously growing. Meanwhile, the INPUT_SAMPLE _SIZE parameter in the above validation experiments is generally set as large as possible to ensure the integrity of the parsing results. This setting applies intense pressure to the performance and scalability of log parsing methods. Whether LPME can achieve satisfactory throughput for large-scale parsing samples is critical. To evaluate the scalability of LPME, we gradually increase the input sample size. Then, we use several comparison groups to assess the increasing time cost of running LPME. For the Windows System, OpenStack, and HealthApp datasets, INPUT_SAMPLE_SIZE is set to 2, 4, 8, 16, and 32 k. For the payment system dataset, INPUT_SAMPLE_SIZE is set to 5, 10, 20, 40, and 80 k. The experimental results are shown in Fig. 6. The time cost recorded in our experiment is only for the multiline event parsing process and does not include pre-parsing. As seen from the resulting data in Fig. 6, regardless of the preceding part, the running time of LPME does not increase considerably when the input size doubles; only a slight increase is observed. The main reason for the slight growth is that the increasing input scale increases the I/O overhead of the first load and traversal process. In addition, when comparing the performance of different pre-parsing methods on the same dataset, no substantial differences were observed; pre-parsing methods do not considerably affect the runtime of multiline event parsing in LPME. The phenomena observed in the experiment are consistent with the conclusions of the algorithm complexity in Section 3.4. The actual run time of LPME is not correlated with the size of input S but with the output's scale. In our experiments, both the number of types and the length of the multiline template are within an order of 10^2, such that LPME's performance is close to constant time. Another notable point is that due to the periodicity of the logs, an extreme increase in sample size is equivalent to the effect of repeating samples. Therefore, after the input samples reach some threshold, further increasing the sample size no longer positively impacts the algorithm's effectiveness.



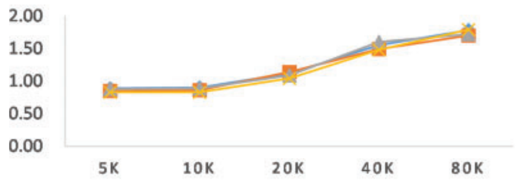
(a) Windows System



(b) OpenStack



(c) HealthApp



(d) Payment System

Figure 6: The scalability test results. The horizontal axis represents different input sizes. The vertical axis represents the running time. The unit of the running time is seconds, and the presented value is the average of ten runs

4.4 *Interference of Noise*

Due to factors such as concurrent processes, it cannot be ruled out that some multiline events are interleaved with other unrelated log outputs. Therefore, consecutively output multiline events may become discontinuous in the printed log text. This situation is regarded as noise disruption. We examine the noise tolerance of LPME as follows. We deliberately insert noisy logs into the interior of the multiline events to randomly disturb the experimental datasets. We control the noise level by the probability p. The larger p is, the greater the amount of noise added to the datasets. For the four datasets, we take $p = 0\%$, $p = 10\%$, $p = 15\%$ and $p = 20\%$. For every experiment, we repeat the tests ten times and take the average F1 score for comparison. The results are shown in Fig. 7. The result when $p = 0\%$ is the original noise-free result. Fig. 7 shows that as p increases, the F1-score obtained by LPME decreases rapidly. Moreover, the performance of LPME does not decrease linearly with an increasing value of p but accelerates. Starting from $p = 15\%$, LPME can hardly obtain the correct parsing result. The reason for the above phenomenon is that the LPME is designed based on cohesiveness, but the added noise damages the cohesiveness. Therefore, noise can easily lead to algorithm failure. Although noise is not naturally present in the validation dataset, we artificially create noise for the experiments, and such noise may exist in other types of system logs. Therefore, this problem will need to be solved in future work.

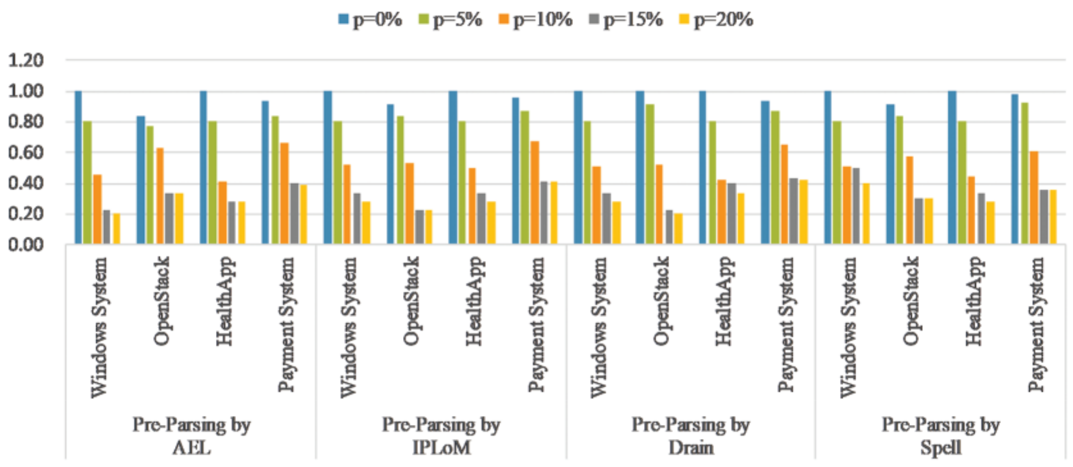


Figure 7: Testing for noise interference

5 Conclusion and Future Work

5.1 *Conclusion and Discussion*

Automated log parsing is required for log mining and analysis. Still, existing research on automated parsing assumes that each event object corresponds to only a single line of log text, which is inconsistent with current application requirements. Based on previous research, this paper proposes LPME, an automated parsing method for multiline events. LPME is a layer-by-layer iterative search algorithm based on heuristic and empirical rules. In addition to the theoretical analysis, we experimentally test the proposed algorithm on four real datasets, including three publicly available datasets and one confidential dataset. Evaluations show that the actual time complexity of LPME parsing for multiline events is close to the constant time, which enables it to manage large-scale sample inputs. On the experimental datasets, the performance of LPME achieves 1.0 for recall, and the precision is generally higher than 0.9. The experimental results corroborate the theoretical analysis and confirm the effectiveness and practicability of LPME.

In addition, we give the following notes about limitations and crucial assumptions in this paper. First, as discussed in Section 4.4, LMPE currently shows limits when dealing with noisy log data. The experimental data show that the performance of LPME is almost dissipated at noise probability = 15%. Although there is no naturally occurring noise in the dataset used for the validation in this paper, various types of noise inevitably exist in other logs in reality. Therefore, it is necessary to supplement the response to this issue in future work. Second, LPME is essentially an offline batch data processing method that requires the user to provide samples for processing. This paper assumes that the user can select sample logs of appropriate size and content. Although we tested the performance of LPME in Section 4.3 and concluded that LPME has a good execution time performance for the growth of sample inputs, LPME can handle larger data samples. However, if the samples are not selected sufficiently, the results obtained by LPME will be incomplete. Therefore, it will be necessary to provide a more scientific method to guide users in selecting log samples.

5.2 Future Work

In future work, we will consider the possible noise interference and the sample selection problem mentioned above. In this regard, we envisage that a preprocessing module can be added to LPME to preprocess the total amount of logs. The purpose of preprocessing is 1) to mark the noisy data so that the subsequent steps can ignore the noisy log messages and 2) to mark the most reasonable range of log samples, which is guaranteed to contain all types of log messages without being too large. Machine learning techniques will be one of the possible routes to explore to implement the preprocessing module. In particular, recurrent neural network (RNN) technology with contextual memory represented by long short term memory (LSTM) [29,30], which has a natural echo with the characteristics of log message streams, will be one of the technologies worthy of verification in the future. For LPME, these future efforts will effectively enhance the robustness, expand the scope of application, and improve the ease of use.

Funding Statement: The authors received no specific funding for this study.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] C. Zhang, X. Peng, C. Sha, K. Zhang, Z. Fu *et al.,* "DeepTraLog: Trace-log combined microservice anomaly detection through graph-based deep learning," in *IEEE/ACM 44th Int. Conf. on Software Engineering (ICSE)*, Pittsburgh, PA, USA, pp. 623–634, 2022.

[2] R. Sinha, R. Sur, R. Sharma and A. K. Shrivastava, "Anomaly detection using system logs: A deep learning approach," *International Journal of Information Security and Privacy*, vol. 16, no. 1, pp. 1–15, 2022.

[3] S. Zhang, L. Song, M. Zhang, Y. Liu, W. Meng *et al.,* "Efficient and robust syslog parsing for network devices in datacenter networks," *IEEE Access*, vol. 8, pp. 30245–30261, 2020.

[4] M. Abolfathi, I. Shomorony, A. Vahid and J. H. Jafarian, "A game-theoretically optimal defense paradigm against traffic analysis attacks using multipath routing and deception," in *Proc. of the 27th ACM on Symp. on Access Control Models and Technologies*, New York, NY, USA, pp. 67–78, 2022.

[5] R. Wang, S. Ying and X. Jia, "Log data modeling and acquisition in supporting SaaS software performance issue diagnosis," *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, no. 9, pp. 1245–1277, 2019.

[6] M. Macák, D. Kruzelova, S. Chren and B. Buhnova, "Using process mining for Git log analysis of projects in a software development course," *Education and Information Technologies*, vol. 26, no. 5, pp. 5939–5969, 2021.

[7] Y. Tao, S. Guo, C. Shi and D. Chu, "User behavior analysis by cross-domain log data fusion," *IEEE Access*, vol. 8, pp. 400–406, 2020.

[8] J. Zhu, S. He, J. Liu, P. He, Q. Xie *et al.,* "Tools and benchmarks for automated log parsing," in *Proc. of the 41st Int. Conf. on Software Engineering: Software Engineering in Practices*, Montreal, QC, Canada, pp. 121–130, 2019.

[9] H. Dai, H. Li, C. -S. Chen, W. Shang and T. -H. Chen, "Logram: Efficient log parsing using nn-gram dictionaries," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 879–892, 2022.

[10] S. Huang, Y. Liu, C. J. Fung, R. He, Y. Zhao *et al.,* "Paddy: An event log parsing approach using dynamic dictionary," in *IEEE/IFIP Network Operations and Management Symp.*, Budapest, Hungary, pp. 1–8, 2020.

[11] M. Du and F. Li, "Spell: Online streaming parsing of large unstructured system logs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 11, pp. 2213–2227, 2019.

[12] H. Studiawan, F. Sohel and C. Payne, "Automatic event log abstraction to support forensic investigation," in *Proc. of the Australasian Computer Science Week*, Melbourne, VIC, Australia, vol. 1, pp. 1–9, 2020.

[13] W. Meng, Y. Liu, F. Zaiter, S. Zhang, Y. Chen *et al.,* "LogParse: Making log parsing adaptive through word classification," in *29th Int. Conf. on Computer Communications and Networks*, Honolulu, HI, USA, pp. 1–9, 2020.

[14] G. Ayoade, A. El-Ghamry, V. Karande, L. Khan, M. F. Alrahmawy *et al.,* "Secure data processing for IoT middleware systems," *The Journal of Supercomputing*, vol. 75, no. 8, pp. 4684–4709, 2019.

[15] C. Qiu, H. Yao, C. Jiang, S. Guo and F. Xu, "Cloud computing assisted blockchain-enabled internet of things," *IEEE Transactions on Cloud Computing*, vol. 10, no. 1, pp. 247–257, 2022.

[16] D. El-Masri, F. Petrillo, Y. -G. Guéhéneuc, A. Hamou-Lhadj and A. Bouziane, "A systematic literature review on automated log abstraction techniques," *Information and Software Technology*, vol. 122, no. 2, pp. 106276, 2020.

[17] W. Xu, L. Huang, A. Fox, D. A. Patterson and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proc. of the 27th Int. Conf. on Machine Learning (ICML-10)*, Haifa, Israel, pp. 37–46, 2010.

[18] R. Vaarandi, "Mining event logs with SLCT and LogHound," in *IEEE/IFIP Network Operations and Management Symp.: Pervasive Management for Ubioquitous Networks and Services*, Salvador, Bahia, Brazil, pp. 1071–1074, 2008.

[19] R. Vaarandi and M. Pihelgas, "LogCluster-a data clustering and pattern mining algorithm for event logs," in *11th Int. Conf. on Network and Service Management*, Barcelona, Spain, pp. 1–7, 2015.

[20] Q. Fu, J. -G. Lou, Y. Wang and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *The Ninth IEEE Int. Conf. on Data Mining*, Miami, Florida, USA, pp. 149–158, 2009.

[21] L. Tang, T. Li and C. -S. Perng, "LogSig: Generating system events from raw textual logs," in *Proc. of the 20th ACM Conf. on Information and Knowledge Management*, Glasgow, United Kingdom, pp. 785–794, 2011.

[22] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang *et al.,* "LogMine: Fast pattern recognition for log analytics," in *Proc. of the 25th ACM Int. Conf. on Information and Knowledge Management*, Indianapolis, IN, USA, pp. 1573–1582, 2016.

[23] K. Shima, "Length matters: Clustering system log messages using length of words," arXiv, 1611.03213, 1–10, 2016.

[24] P. He, J. Zhu, Z. Zheng and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *2017 IEEE Int. Conf. on Web Services*, Honolulu, HI, USA, pp. 33–40, 2017.

[25] A. Makanju, A. N. Zincir-Heywood and E. E. Milios, “A lightweight algorithm for message type extraction in system application logs,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 11, pp. 1921–1936, 2012.

[26] Z. M. Jiang, A. E. Hassan, G. Hamann and P. Flora, “An automated approach for abstracting execution logs to execution events,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 4, pp. 249–267, 2008.

[27] LPME, 2022. [Online]. Available: https://github.com/yumg/lpme

[28] P. He, J. Zhu, S. He, J. Li and M. R. Lyu, “Towards automated log parsing for large-scale log data analysis,” *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 6, pp. 931–944, 2018.

[29] A. Althobaiti, A. A. Alotaibi, S. Abdel-Khalek, E. M. Abdelrahim, R. F. Mansour *et al.,* “Intelligent data science enabled reactive power optimization of a distribution system, sustainable computing,” *Informatics and Systems*, vol. 35, pp. 100765, 2022.

[30] C. You, Q. Wang and C. Sun, “sBiLSAN: Stacked bidirectional self-attention LSTM network for anomaly detection and diagnosis from system logs,” in *Intelligent Systems and Applications-Proc. of the 2021 Intelligent Systems Conf.*, Amsterdam, The Netherlands, vol. 296, pp. 777–793, 2021.