An Effective Approach for Parsing Large Log Files

Conference Paper · October 2022

DOI: 10.1109/ICSME55016.2022.00009

4 authors, including:

Issam Sedki
Concordia University Montreal
6 PUBLICATIONS **15** CITATIONS
SEE PROFILE

Abdelwahab Hamou-Lhadj
Concordia University Montreal
179 PUBLICATIONS **2,511** CITATIONS
SEE PROFILE

Mohammed A. Shehab
Concordia University Montreal
44 PUBLICATIONS **1,109** CITATIONS
SEE PROFILE

An Effective Approach for Parsing Large Log Files

Issam Sedki
ECE, Concordia University
Montreal, Canada
issam.sedki@concordia.ca

Abdelwahab Hamou-Lhadj
ECE, Concordia University
Montreal, Canada
wahab.hamou-lhadj@concordia.ca

Otmane Ait-Mohamed
ECE, Concordia uUniversity
Montreal, Canada
otmane.aitmohamed@concordia.ca

Mohammed A. Shehab
ECE, Concordia University
Montreal, Canada
mohammed.shehab@concordia.ca

***Abstract*—Because of their contribution to the overall reliability assurance process, software logs have become important data assets for the analysis of software systems. Logs are often the only data points that can shed light on how a software system behaves once deployed. Unfortunately, logs are often unstructured data items, hindering viable analysis of their content. There are studies that aim to automatically parse large log files. The primary goal is to create templates from raw log data samples that can later be used to recognize future logs. In this paper, we propose ULP, a Unified Log Parsing tool, which is highly accurate and efficient. ULP combines string matching and local frequency analysis to parse large log files in an efficient manner. First, log events are organized into groups using a text processing method. Frequency analysis is then applied locally to instances of the same group to identify static and dynamic content of log events. When applied to 10 log datasets of the LogPai benchmark, ULP achieves an average accuracy of 89.2%, which outperforms the accuracy of four leading log parsing tools, namely Drain, Logram, SPELL and AEL. Additionally, ULP can parse up to four million log events in less than 3 minutes. ULP is available online as an open source and can be readily used by practitioners and researchers to parse effectively and efficiently large log files so as to support log analysis tasks.**

***Index Terms*—Log Parsing, Log Abstraction, Log Analytics, Software Logging, Software Maintenance and Evolution.**

I. Introduction

Logs are generated by logging statements inserted by developers in the source code. An example of a logging statement is shown in Figure 1, which is a code snippet extracted from a Hadoop Distributed File System (HDFS) Java source file. The generated log event (Figure 1) is composed mainly of two parts: the log header and log message. The log header typically contains the timestamp (e.g., 081109 283519), the process id (147), the log level (INFO), and the logging function (dfs.DataNodePacketResponder). The log message contains static tokens (usually text) such as "Received block", "size of", "from" in the example of Figure 1 and dynamic tokens, which represent variable values (`blk_-1680999687919862986`, `91178`, `/10.250.14.224`).

Log files include a plethora of information on the execution of a software system that is used to help with different software engineering activities, such as anomaly detection [1] [2] [3] [4], debugging and comprehension of system failures [5] [6] [7] [8] [9], testing and performance analysis [10] [11] [12], operational intelligence [9] [11] [13] [14], failure prediction [9] [15], detection of data leakage [16], and more recently, AI for IT Operations (AIOps) [14] [17].

Logs, on the other hand, have traditionally been difficult to work with. Typical log files may be considerably large (in the order of millions of events) [8] [18] [19]. Furthermore, the logging practice is known to be ad hoc, with no defined best practices [20]. To make matters worse, logs are mainly unstructured data files due to the lack of a standardized format [1] [15] [19]. As a result, extracting relevant information from large raw log files [21] [22] can be a daunting and time-consuming task.

In this paper, we focus on the problem of log parsing, which consists of (semi-)automatically converting unstructured raw log events into a structured format that would facilitate future analysis. Log parsing is further reduced to the problem of parsing log messages. This is because log headers usually follow the same format within the same log file. Parsing a log message consists of automatically distinguishing the static text from the dynamic variables. The result of parsing the log event in Figure 1 is the extraction of the template in `Received Block <*> of size <*> from <*>`, which identifies the log message structure. One way to parse log events would be to use regular expressions [23] [24]. The problem is that typical industrial log files may contain hundreds of log templates as shown by Chow et al. [13] and Mi et al. [25]. Furthermore, as the system evolves, new log formats are produced due to the use of multiple logging libraries, necessitating the ongoing updating of the regular expressions. The use of regular expressions to parse various types of logs is simply impractical, which has led researchers to develop more intelligent log parsing techniques (see [20] for a good survey). Existing approaches, however, suffer from many limitations [20] including their reliance on domain knowledge, inability to demarcate static content from dynamic variables for complex log files, and use of advanced machine learning algorithms, which require parameter tuning.

In this paper, we propose ULP (Unified Log Parser), a

Logging statement: LOG.info("Received Block"+ block + "of size" + block_size + " from" + sender_ip)

Raw log line: 081109 283519 147 INFO dfs.DataNodePacketResponder: Received block blk_-1680999687919862986 of size 91178 from /10.250.14.224

Log template: Received Block <*> of size <*> from <*>

Fig. 1. An example of a logging statement, the generated log event, and the expected log template

simple yet powerful approach that recognizes log structures from any log files without prior domain knowledge or the use of complex machine learning techniques. ULP relies on string matching and local frequency analysis. It begins by grouping similar log events into groups using a string matching similarity technique. It then uses frequency analysis on instances of each group to distinguish between static and dynamic log message tokens. We conjecture that tokens that are more often repeated in the same group of similar log events are more likely static tokens than dynamic tokens. This is not the first time that frequency analysis is used in log parsing. Other tools such as Drain [26] and Logram [27] also use frequency analysis. However, these tools apply frequency analysis to the entire log file, which makes it hard to find a clear demarcation between static and dynamic tokens. ULP, on the other hand, applies frequency analysis to log events that belong to the same group rather than to the entire log dataset, increasing the likelihood of distinguishing between static and dynamic tokens.

We compared ULP to four major log parsing tools, namely Drain [26], AEL [24], SPELL [28], and Logram [27] by applying them to 10 log datasets from the LogPai benchmark[1]. Our findings show that ULP outperforms existing tools in parsing all the log datasets. Our technique has an average accuracy of 89.2 %, while the second-best method, Drain, has an average accuracy of 73.7 %. Furthermore, ULP can parse big files containing up to 4 million log events in under 3 minutes.

ULP is available as an open source tool and accessible online[2]. Practitioners can readily embed ULP into their log analytic suite and not have to worry about creating parsers that are tailored to specific log files, which we believe may result in improved productivity and better software maintenance.

The organization of this paper is as follows: Section II looks at the actual approaches used in log parsing and how they compare to our solution. We present the ULP approach in Section III. Section IV focuses on the evaluation of ULP using 10 log files. Section V highlights the distinctiveness of our technique and explores the reasons why ULP outperforms other comparable approaches, followed by threats to validity. We conclude the paper in Section VII and discuss future directions.

[1]https://github.com/logpai

[2]http://zenodo.org/record/6425919

II. Related Work

In recent years, log analysis has received a great deal of attention from researchers and practitioners due to the increasing need to understand complex systems at runtime. One of the most comprehensive survey of log parsing techniques is the one proposed by El-Masri et al. [20] in which the authors proposed a quality model for classifying log parsing techniques and examined 17 different log parsing tools using this model. Existing tools can be categorized into groups based on the techniques they use, namely rule-based parsing tools, frequent token mining, natural language processing, and classification/clustering approaches. We discuss the main approaches in what follows and conclude with a general comparison of ULP with these techniques.

Jiang et al. [29] introduced AEL (Abstracting Execution Logs), which is a log parsing method that relies on textual similarity to group log events together. AEL starts by detecting trivial dynamic tokens using hard-coded heuristics based on system knowledge (e.g., IP addresses, numbers, memory addresses). The resulting log events are then tokenized and assigned to bins based on the number of terms they contain. This grouping is used to compare the log messages in each bin and abstracts them into templates. The problem with AEL is that it assumes that events that contain the same number of terms should be grouped together, resulting in many false positives.

Vaarandi et al. [30] [31] proposed SLCT (Simple Logfile Clustering Tool). The authors used clustering techniques to identify log templates. SLCT groups log events together based on their most common frequent terms. To this end, the approach relies on a density-based clustering algorithm to recognize the dynamic tokens, SLCT uses frequency analysis across all log lines in the log file. LogCluster [17] is an improved version of SLCT proposed by the same authors. LogCluster extracts all frequent terms from the log messages and arranges them into tuples. Then, it splits the log file into clusters that contain at least a certain number of log messages.

Another clustering approach is the one proposed by Makanju et al., which is called IPLOM (Iterative Partitioning Log Mining) [32]. IPLOM employs a heuristic-based hierarchical clustering algorithm. In this approach, the first step is to partition the log messages. For this, the authors used heuristics considering the size of log events. The next step is to further divide each partition based on the highest

number of similar terms. Fu et al. proposed LKE (Log Key Extraction) [33], another clustering-based approach, using a distance-based clustering technique. Log events are grouped together using weighted edit distance, giving more weight to the terms that appear at the beginning of log events. Then, LKE splits the clusters until each raw log level in the same cluster belongs to the same log key and extracts the common parts of the raw log key from each cluster to generate event types.

Hamooni et al. proposed LogMine [34], which uses MapReduce to abstract heterogeneous log messages generated from various systems. The LogMine algorithm consists of a hierarchical clustering module combined with pattern recognition. It uses regular expressions based on domain knowledge to detect a set of dynamic variables. Then, it replaces the real value of each field with its name. It then clusters similar log messages with the friends-of-friends clustering algorithm.

Natural Language Processing (NLP) techniques have also been used for log parsing. Logram, a recent approach proposed by Dai et al. [27], is an automated log parsing approach developed to address the growing size of logs, and the need for low-latency log analysis tools. It leverages n-gram dictionaries to achieve efficient log parsing. Logram stores the frequencies of n-grams in logs and relies on the n-gram dictionaries to distinguish between static tokens and dynamic variables. Moreover, as the n-gram dictionaries can be constructed concurrently and aggregated efficiently, Logram can achieve high scalability when deployed in a multi-core environment without sacrificing parsing accuracy. Kobayashi et al. proposed NLP-LTG (Natural Language Processing–Log Template Generation) [35], which considers event template extraction from log messages as a problem of labeling sequential data in natural language. It uses Conditional Random Fields (CRF) [36] to classify terms in log messages as static or dynamic. To construct the labeled data (the ground truth), it uses human knowledge and regular expressions. Thaler et al. [37] proposed NLM-FSE (Neural language Model-For Signature Extraction) , which trains a character-based neural network to classify static and dynamic parts of log messages.

He et al. [26] proposed Drain, a tool that abstracts log messages into event types using a parse tree. Drain algorithm consists of five steps. Drain starts by preprocessing raw log messages using regular expressions to identify trivial dynamic tokens, just like ULP. Then, it builds a parse tree using the number of tokens in log events. Drain assumes that tokens that appear in the beginning of a log message are most likely static tokens. It uses a similarity metric that compares leaf nodes to event types to identify log groups.

Spell (Streaming Parser for Event Logs using an LCS) [28] is a log parser, which converts log messages into event types. Spell relies on the idea that log messages that are produced by the same logging statement can be assigned a type, which represents their longest common sequence. The LCS of the two messages is likely to be static fields.

The main difference between ULP and existing approaches lies in the way ULP is designed. ULP leverages the idea that static and dynamic tokens of log events can be better identified if we use frequency analysis locally on instances of log events that belong to the same group. To this end, it uses a string matching technique to create groups of similar events. This is contrasted with techniques that use clustering alone (e.g., AEL and IPLOM) and those that apply frequency analysis to the entire log file (e.g., Drain and Logram), i.e., globally. As we will see in the evaluation section, these design choices make ULP a very accurate and efficient log parser compared to leading open source log parsers.

III. Approach

ULP consists of the following steps: pre-processing, grouping of similar log events, and the generation of log templates using local frequency analysis. The first step is a pre-processing step where the header such as the timestamp, the log level, and the logging function are identified. We also detect trivial dynamic tokens such as IP and MAC addresses based on common regular expressions. The second step of ULP is to identify similar log events and group them together. To this end, we use text similarity measures. Once the groups of similar log events are formed, we apply frequency analysis to instances of each group to determine the static and dynamic tokens, and derive the various log templates that are then mapped back to each log event. Algorithm 1 shows the steps of ULP. We explain each step in more detail in the following subsections. To illustrate our approach, we use the sample log events from the HDFS log dataset (one of the datasets used to evaluate ULP) shown in Figure 2. We added a line number to each log event to help explain the approach.

A. Pre-processing

The pre-processing of log events begins by delineating the header information, including the timestamp, process ID, log level, and logging function (Lines 1–3, Algorithm 1). Prior research showed that this information can be readily identified using simple regular phrases [27]. Figure 2 shows that all the HDFS log events of the running example begin with a timestamp (e.g., 081109 203615), a process ID (148), a log level (INFO), and a logging function `dfs.DataNode$PacketResponder`. Another essential part of the pre-processing step is the identification of trivial dynamic variables such as IP and MAC addresses. Identifying such variables can improve the parsing accuracy as shown by He et al. [38] and all the tools studied in this paper (i.e., Drain [26], SPELL [28], Logram [27] and AEL [24]) include this step in their approach. For ULP, this step also increases the chances of identifying similar log events that should be instances of the same group. Grouping of similar events is discussed in more detail in the next subsection. We created regular expressions to detect the following trivial dynamic variables: Mac addresses, IPV6 addresses, URLs (beginning with HTTP and HTTPS), numerical in hexadecimal format, Dates such as `2002-03-24` and `2002-03-24`, Time in the format `hh:mm:ss`. These regular expressions can be found

```
1  081109 203615 148 INFO dfs.DataNode$PacketResponder: PacketResponder 1 for block
   blk_38865049064139660 terminating

2  081109 203807 222 INFO dfs.DataNode$PacketResponder: PacketResponder 0 for block
   blk_-6952295868487656571 terminating

3  081109 204005 35 INFO dfs.FsNameSystem: BLOCK* NameSystem.addStoredBlock: blockMap
   updated: 10.251.73.220:50010 is added to blk_7128370237687728475 size 67108864

4  081109 204015 308 INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block
   blk_8229193803249955061 terminating

5  081109 208106 329 INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block
   blk_-6670958622368987959 terminating

6  081109 204132 26 INFO dfs.FsNameSystem: BLOCK* NameSystem.addStoredBlock: blockMap
   updated: 10,251.43.115:50010 is added to blk_3050920557425079149 size 67105864

7  081109 204328 34 INFO dfs.FsNameSystem: BLOCK* NameSystem.addStoredBlock: blockMap
   updated: 10,251.203.80:50010 is added to blk_7688946331004732825 size 67105864

8  081109 201453 24 INFO dfs.FsNameSystem: BLOCK* NameSystem.addStoredBlock: blockMap
   updated: 10.250.11.85:50010 is added to blk_2377150260125000006 size 67108064

9  081109 204525 512 IFO dfs.DatatlodefPacketResponder: PacketResponder 2 for block
   blk_572492839787299681 terminating

10 081109 201655 556 INFO dfs.DataNode$PacketResponder: Received block
   blk_3587505140051952248 of size 67IO8864 from /10.251.42.84

11 081109 204722 567 INFO dfs.DataNode$PacketResponder: Received block
   blk_5407003568334525940 of size 67108864 from /10.251.214.112

12 081109 204815 653 INFO dfs.DataNode$PacketResponder: Received block
   blk_9212264480425680329 of size 67108864 from /10.251.214.111
```

Fig. 2. HDFS log events used as a running example

in ULP code repository[3]. Furthermore, ULP allows users to define additional regular expressions to represent domain-specific variables. It is worth mentioning that all the regular expressions used by ULP are generic and independent from the log files to be parsed, unlike other tools that include regular expressions for detecting log-specific variables at the risk of limiting its generalization. For instance, `r'blk_(|-)[0-9]+'` is used in Logram to detect block ids within HDFS log files.

The result of applying the pre-processing step to the HDFS running example is shown in Figure 3, where the header information is detected and removed, and the IP addresses in Lines 3, 6, 7, 10, 11, and 12 are replaced by <*>, indicating that they are dynamic tokens.

```
1  PacketResponder 1 for block blk_38865049064139660 terminating
2  PacketResponder 0 for block blk_-6952295868487656571 terminating
3  BLOCK* NameSystem.addStoredBlock: blockMap updated: <*> is added to
   blk_7128370237687728475 size 67108864
4  PacketResponder 2 for block blk_8229193803249955061 terminating
5  PacketResponder 2 for block blk_-6670958622368987959 terminating
6  BLOCK* NameSystem.addStoredBlock: blockMap updated: <*> is added to
   blk_3050920557425079149 size 67105864
7  BLOCK* NameSystem.addStoredBlock: blockMap updated: <*> is added to
   blk_7688946331004732825 size 67105864
8  BLOCK* NameSystem.addStoredBlock: blockMap updated: <*> is added to
   blk_2377150260125000006 size 67108064
9  PacketResponder 2 for block blk_572492839787299681 terminating
10 Received block blk_3587505140051952248 of size 67IO8864 from <*>
11 Received block blk_5407003568334525940 of size 67108864 from <*>
12 Received block blk_9212264480425680329 of size 67108864 from <*>
```

Fig. 3. Results of pre-processing the HDFS log events example

B. Grouping similar log events

The second step of ULP is to group similar log events together (Lines 4 to 6 of Algorithm 1). This grouping will help later distinguish between the static and dynamic tokens by applying frequency analysis to instances of each group. For example, the log messages of Lines 1, 2, 4, 5, and 9 all deal with terminating `PacketResponder` (used by HDFS to manage the processing of data into a series of pipeline nodes) and only vary in terms of the task_number and the block_id. So if we can put these log messages into the same group, we can easily see that the static tokens (`PacketResponder, for, block, terminating`) appear in all the log events of that group, and that the dynamic tokens, i.e., the `task_number` and the `block_id`, vary from one log event to another, hence the idea of using frequency analysis on instances of the same group instead of applying it to the entire log file, which may not lead to such a clear demarcation.

Our grouping strategy relies on a string matching technique. We measure the similarity of two log events based on the number of tokens they contain combined with the tokens that are most likely static tokens (i.e., tokens that do not contain digits and/or special characters). To do this, for each log event, we first count the number of words it contains. A word is defined as any token that is delimited by a

[3]http://zenodo.org/record/6425919

Algorithm 1: Overall ULP Algorithm

```
Data: LogEvents
Result: LogTemplates, GroupOfLogs
1  foreach LogEvent ∈ RawLogfile do
2      LogEvent ← RemoveHeader(LogEvent)
3      LogEvent ← RemovePunctuation(LogEvent)
4      LogEvent ←
         RemoveObviousDynamicTokens(LogEvent)
       /* Generate an id for a Log event
          based on the tokens that are not
          deemed dynamic, and the count of
          tokens                          */
5      LogEvent.Id ← GenerateIds(LogEvent)
6  GroupOfLogs ←
     LogEvents.GroupBy(LogEvent.Id)
   LogTemplates ← []
7  foreach g ∈ GroupOfLogs do
       /* Dimensions reduction to improve
          algorithm performance           */
8      subGroup ← Sample(g)
       /* The template will be constructed
          base on one of the Log events by
          removing dynamic tokens and
          replacing them by <*>           */
9      template ← subGroup[0]
       /* Get the group vocabulary        */
10     Tokens ← GetVocabulary(subGroup)
       /* Count occurences avoid duplicate
          for each Log event              */
11     subGroup.CountTokensAvoidDuplicatePerEvent()
12     foreach t ∈ Tokens do
           /* static tokens appear all the
              time in the subgroup, we
              remove dynamic tokens out of
              the template                */
13         if t.count<subGroup.length then
14             p ← template.Remove(t)
15         LogTemplates.Append(template)
16         g.template = template
17     GroupOfLogs ←
         mergeGroupWithSimilarTemplate(GroupOfLogs)
```

space character. Note we do not use any other delimiters to prevent splitting a dynamic variable into multiple tokens. For example, the token `04:09:19.989` is considered by ULP as one token. Then, we identify tokens that only contain alphabetical letters. In other words, we ignore tokens that contain digits and/or special characters, which are most likely dynamic tokens. Finally, we convert the log event into a string that results from concatenating the alphabetical tokens and the total number of tokens. Two log events are grouped together if there is an exact match (i.e., 100%) between their corresponding strings. For example, the two following log events: "`block x23 from 125.12.1.1 allocated to block x45`" and "`block x23 from 125.125.123.144 allocated to block x125`" will be grouped together since they contain the same number of tokens (8) and the same alphabetical tokens and that also appear in the same order `block from allocated to block`.

Another alternative design would be to consider similar but not necessarily identical strings. For this, we would need to define a threshold beyond which two log events can be deemed similar. Setting such a threshold may be a difficult task since it may vary from one group of events to another. We deliberately opted for a grouping technique that requires an exact match to prevent the use of thresholds, which may necessitate human intervention or advanced statistical methods to determine the right threshold.

Applying the grouping approach to the log messages of Figure 3 results in three groups. The first one consists of log messages 1, 2, 4, 5, and 9, which contain the static token `PacketResponder`. The second pattern consists of log messages 3, 6, 7, and 8, representing the message `BLOCK* NameSystem.addStoredBlock: blockMap update`. The last one contains log messages 10, 11, and 12 for the `Received block` log event. At this stage, we have only identified the groups. The next step will leverage local frequency analysis to detect the dynamic tokens and associate a template to log events within each group.

C. Generation of log templates using local frequency analysis

In this step, we analyze the occurrences of the tokens of each group of log events by counting the number of times each token appears in the log events that belong to the same group (Lines 7–16 of Algorithm 1). As explained earlier, we expect to see static tokens appear all the time in each log event of the same group (because of the way our grouping technique works), while the dynamic tokens are expected to occur only in some log events and not all of them. Therefore, we consider anything that occurs less that the maximum frequency as a dynamic token. It should be noted that in our approach, we are more interested in the occurrences of a dynamic token over several log events than in a single log event. Counting the same token twice in the same log event raises its frequency across several log events in the same group, which introduces a bias. Duplicates in the same log event are counted only once to prevent this bias. For example, if the token `block` occurs twice in the same log event, it will be counted as one occurrence in this event, not two. Table 1 displays the frequency of the tokens of log events of the first group, which consists of the following events:

1) `PacketResponder 1 for block blk_38865049064139660 terminating`
2) `PacketResponder 0 for block blk_-6952295868487656571 terminating`
3) `PacketResponder 2 for block blk_8229193803249955061 terminating`

4) `PacketResponder 2 for block blk_-6670958622368987959 terminating`
5) `PacketResponder 2 for block blk_572492839287299681 terminating`

In this group, the terms `PacketResponder`, `for`, `block`, `terminating` appear five times (the maximum frequency). The other tokens (task numbers 0, 1, 2, and block ids `blk_3886504906413966O`, `blk_-6952295868487656571`, `blk_8229193803249955061`, `blk_-6670958622368987959`, `blk_572492839287299681`) appear less than five times. ULP classifies them as dynamic tokens.

TABLE I
EXAMPLE OF A FREQUENCY ANALYSIS RESULT APPLIED TO THE FIRST GROUP OF LOG EVENTS

Term	Frequency	Classification
PacketResponder	5 out of 5	static token
0	1 out of 5	dynamic token
1	1 out of 5	dynamic token
2	3 out of 5	dynamic token
for	5 out of 5	static token
block	5 out of 5	static token
blk_38865049064139660	1 out of 5	dynamic token
blk_-6952295868487656571	1 out of 5	dynamic token
blk_8229193803249955061	1 out of 5	dynamic token
blk_-6670958622368987959	1 out of 5	dynamic token
blk_572492839287299681	1 out of 5	dynamic token
terminating	5 out of 5	static token

The resulting template from applying local frequency analysis to this group of events is: `PacketResponder <*> for block <*> terminating`. The generated log templates when applying local frequency analysis to the log events of the running example are shown below. Except for `67108864` (the size of a data block in HDFS), ULP was able to uncover all the static and dynamic tokens. ULP was no able to detect the dynamic token `67108864` because we are examining a small sample of log events. In practice, the application of ULP to large HDFS log files should be able to detect this variable at some point in time when a different variable appears in another log event of the same group.

1) `PacketResponder <*> for block <*> terminating`
2) `BLOCK* NameSystem.addStoredBlock: blockMap updated: <*> is added to <*> size 67108864`
3) `Received block <*> of size 67108864 from <*>`

IV. EVALUATION

In this section, we evaluate the effectiveness of ULP in parsing logs of 10 log datasets of the LogPai benchmark [23] available online[4]. The datasets consist of a collection of log files, generated from various systems including Apache, HPC, HDFS as shown in Table II. They are used extensively in the literature (e.g., Drain [26], SPELL [28], and Logram [27]).

This evaluation aims to answer the following two research questions:

1) RQ1. What is the accuracy of ULP in parsing the 10 log files of the LogPai benchmark and how does it compare to leading log parsing tools?
2) RQ2. What is the efficiency of ULP and how does it compare to leading log parsing tools?

TABLE II
LOGPAI DATASETS

Datasets	Description	Size
Apache	Apache server error log	5.1MB
BGL	Blue Gene/L supercomputer log	743MB
HDFS	Hadoop distributed file system log	1.47GB
Hadoop	Hadoop mapreduce job log	2MB
HPC	High performance cluster	32MB
Proxifier	Proxifier software log	2.42MB
Spark	Spark job log	166MB
Thunderbird	Thunderbird supercomputer log	29.60GB
Openstack	OpenStack software log	41MB
Zookeeper	ZooKeeper service log	10MB

We evaluated ULP using accuracy and efficiency. We also compared ULP to four leading log parsing tools, namely Drain [26], AEL [24], SPELL [28] and Logram [27]. We selected these tools because prior studies [23] [27] [38] showed that these tools yield the highest accuracy and efficiency compared to other log parsing tools such as SLCT. We ran the same experiments with the selected log parsers using the most recent version of their publicly accessible source code.

A. Accuracy

Each log dataset of the LogPai benchmark used in this study comes with a subset of 2,000 log events that have been parsed manually by the LogPai team. The log templates were identified and each log event out of the 2,000 events was associated with a specific log template. This ground truth dataset is meant for researchers to test their Log parsers and has been used by many log parsing tools such as Drain [26], AEL [24], Lenma [39], IPLoM [32], and Logram [27]. We also use it in this study to evaluate ULP and to compare ULP with existing tools. Table III shows an example of events from the Apache 2,000 labelled log events where a log event (represented here by an id starting with "E") is mapped to a template that was extracted manually by the LogPai team.

The way accuracy is measured in related studies is based on the work of Zhu et al. [23] where the authors compared the accuracy of 13 log parsing tools including some of the tools used in this paper (Drain, AEL, and Spell). Logram, which was published after the work of Zhu et al. [23], also uses the same approach. Zhu et al.'s accuracy metric is based on the number of log events that are correctly identified as belonging to the same template when compared to the ground truth. This metric, however, does not check if the template

[4]https://zenodo.org/record/3227177#.YUqmXtNPFRE

TABLE III
AN EXAMPLE OF MANUALLY LABELED LOG EVENTS FROM THE APACHE LOG DATASET

Event ID	Event Template
E1	jk2_init() Found child <*>in scoreboard slot <*>
E2	workerEnv.init() ok <*>
E3	mod_jk child workerEnv in error state <*>
E4	[client <*>] Directory index forbidden by rule: <*>
E5	jk2_init() Can't find child <*>in scoreboard
E6	mod_jk child init <*><*>

in question is the same as the one in the ground truth. In our opinion, this metric is misleading since it does not assess the ability of a log parser to extract the exact templates as the ones in the ground truth. In other words, Zhu et al.'s metric is necessarily but not sufficient. In this paper, we go one step further by not only comparing if the log events are correctly classified as having the same template, but also checking that the templates we extract are exactly the same as the ones in the ground truth. More precisely, to measure the accuracy of ULP, we compare the templates extracted by ULP to those provided by LogPai for the 2,000 log events of the log datasets shown in Table II. The accuracy is the number of matches divided by 2,000. We apply the same procedure to other log parsing tools and compare our results to theirs. A match is considered if the following requirements are satisfied: (1) all static tokens are detected and displayed in the correct location in the ground truth file; (2) all dynamic variables were identified and replaced by <⋆>; (3) all dynamic variables are shown in the same order as the ground truth; (4) no extra static or dynamic tokens were added. Table IV shows an example of the manual comparison we performed.

TABLE IV
AN EXAMPLE THAT SHOWS HOW WE MEASURE THE ACCURACY OF ULP

Ground truth Template	ULP Generated Template	Match or Not	Explanation
Cannot open channel to <*>at election address /<*>:<*>	Cannot open channel to <*>at election address <*>]	1	Static text is detected correctly. Dynamic tokens are identified. the gap is that ground truth is interpreting IP address with port as 2 variables and the parsing tool as one, which is acceptable.
Expiring session <*>, timeout of <*>ms exceeded	[expiring session <*>timeout of 10000ms exceeded]	0	Only one dynamic token has been identified out of two. The parsing is then considered incorrect.

Results: Table V shows the results of ULP accuracy. **ULP has the best accuracy in parsing all the log datasets** in comparison to all the other tools assessed in this study (i.e., Logram, SPELL, AEL, and Drain). Additionally, our approach has an **average accuracy of 89.2%**, whereas the second most accurate method, Drain, has an average accuracy of 73.7%. Table VI shows the effect size using Cliff's δ, which is a statistical test that indicates the magnitude of that difference [40]. The effect size is considered small when $0.147 \leq \delta < 0.33$, medium for $0.33 \leq \delta < 0.474$, and large for $\delta \geq 0.474$. [41]. Cliff's δ is defined using Equation(1). As shown in Table VI, the difference between the accuracy of ULP and that of

TABLE V
ACCURACY OF ULP COMPARED TO OTHER LOG PARSERS. WE HIGHLIGHTED THE HIGHEST RESULTS IN BOLD.

Name	Drain	Logram	Spell	AEL	ULP
HDFS	0.996	0.981	0.500	0.434	**0.999**
Apache	0.693	0.050	0.269	0.000	**0.699***
HPC	0.745	0.877	0.662	0.045	**0.944**
Proxifier	0.380	0.000	0.015	0.117	**0.979**
ThunderBird	0.868	0.114	0.781	0.021	**0.970**
Openstack	0.400	0.000	0.170	0.000	**0.801**
Spark	0.979	0.201	0.863	0.363	**0.995**
Zookeeper	0.962	0.722	0.918	0.046	**0.971**
Hadoop	0.546	0.125	0.293	0.000	**0.660***
BGL	0.810	0.457	0.702	0.004	**0.910**
AVG	0.737	0.352	0.517	0.103	**0.892**

TABLE VI
THE CLIFF'S EFFECT SIZE TEST RESULTS

Algorithm name	Cliff's Delta
Drain	**0.39**
Logram	**0.74**
Spell	**0.76**
AEL	**1.00**

Logram, SPELL, and AEL is significantly large (Cliff's $delat \geq 0.474$). It is medium in the case of Drain.

$$Cliff's\ \delta = \frac{\sum_i \sum_j sign(y_i - x_j)}{n_y . n_x} \quad (1)$$

We carefully examined the log templates that ULP missed to understand the causes. We found that some errors and inconsistencies in the manual labelling of the benchmark files misled the performance of ULP. For example, the token `CrazyIvan46` in the log event `labeling NETClientConnection evaluate CrazyIvan46 CI46 Perform CrazyIvan46` is considered as a static token, which is an error in the labelling of the data. This should be labelled as a dynamic token because it refers to a username. Another cause of misclassification is the presence of dynamic variables whose values do not change over a large number of log events. For example, the token "workers2.properties" was repeated 568 times in one of the datasets. ULP misclassified it as a static token. The good thing about ULP is that, unlike many existing tools including Drain, it does not make any assumptions about the order of static and dynamic tokens, which reduces significantly the number of false positives. Table VII shows some other examples of errors in the ground truth files causing the low accuracy results of ULP when applied to Hadoop and Apache logs. This seems to affect the other parsers as well (see Table V).

RQ1 Finding: The accuracy of ULP when applied to 10 log files of the LogPai benchmark is between 66% and 99.9% with a average of accuracy of 89.2%. ULP outperforms Drain, Logram, Spell, and AEL in the parsing of all the 10 log files.

TABLE VII
EXAMPLES OF ISSUES WITH THE GROUND TRUTH FILES FOR APACHE AND HADOOP LOG DATASETS

Log event	Template from ground truth file	Explanation of the issue with the ground truth file
Disk quotas dquot_6.5.1	Disk quotas dquot_<*>.<*>.<*>	Interpreting one dynamic token as many.
data_thread() got not answer from any [Thunderbird_A] datasource	data_thread() got not answer from any [Thunderbird_<*>] datasource	A dynamic variable is a whole, can not be split. The dynamic variable should replace the whole string Thunderbird_34.
Warning: we failed to resolve data source name dn910 dn911 dn912 dn913 dn914 dn915	Warning: we failed to resolve data source name <*>	Multiple dynamic variables are identified as one. This may be confusing since the practitioner loose the info about the number of dynamic tokens contained in the event.
workerEnv.init ok workers2.properties	workerEnv.init ok workers2.properties	The token workers2.properties does not change and does not have any other value, which make him be interpreted as static (occurrences 568).

B. Efficiency

To evaluate ULP's efficiency, we record the execution time to complete the end-to-end parsing process. We repeated the experiment 10 times to avoid any bias and took the average of the execution times. We also run the same experiments for Drain, Logram, and SPELL on our computer and record the running parsing time for these programs in the same manner. We did not assess AEL's efficiency because it has a very poor accuracy (average accuracy of 10%, as indicated in Table V)). We run the experiments using a MacBook Pro laptop running macOS Big Sur version 11.4 and equipped with a 6 Intel Core i7 CPU operating at 2.2GHz, 32GB 2400MHz DDR4 RAM, and a 256 GB SSD hard drive. We use the datasets indicated in Table VIII, publicly accessible in the LogPai benchmark. We selected these datasets because they have previously been used to measure efficiency in other research studies [27] and [26]. We assess efficiency for each log dataset by running ULP and the other tools on random log samples of increasing size, as indicated by the number of log lines so as to examine how the parser's efficiency changes as the file become larger. This was by inspired by the way efficiency was assessed for Drain [26], which uses this log dataset splitting, except that Drain uses the file size rather than the number of log events.

TABLE VIII
NUMBER OF LOG EVENTS OF THE SAMPLE LOG FILES USED TO MEASURE EFFICIENCY

BGL	400	4,000	40,000	400,000	4,000,000
HPC	600	3,000	15,000	75,000	375,000
HDFS	1,000	5,000	10,000	100,000	1,000,000
Zookeeper	4,000	8,000	16,000	32,000	64,000
Spark	1,000	5,000	10,000	100,000	1,000,000

1) Results: Figure 4 shows the efficiency of ULP. ULP requires less than 50 seconds to parse one million log lines from the HDFS log file and less than three minutes to parse four million lines from the BGL log file. ULP's efficiency is comparable to that of Logram (the quickest log parsing tool evaluated in this paper). It is worth mentioning that Logram uses an upfront step to fine-tune the threshold for proper parsing. This step is not included in our analysis of Logram's efficiency because the corresponding code is not available in Logram's Github repository.

RQ2 Findings: ULP can parse up to 4 million log events in less than 3 minutes. It is more efficient than Drain and and Spell when applied to HDS, Spark, Zookepr, HPC, and BGL log files. It exhibits similar efficiency as Logram except for Zookeeper and for large HPC and BGL log files where Logram has a noticeably better efficiency.

V. DISCUSSION

The primary difference between ULP and existing approaches lies in the way ULP is designed. ULP leverages the idea that static and dynamic tokens of log events can be better identified if we use frequency analysis locally on instances of log events that belong to the same group rather than in the entire log file. The similarity technique used to group log events is also unique. This design choice yields excellent results as shown in the previous subsections. It should be noted, however, that the sole reliance on local frequency analysis does not guarantee the detection of all dynamic tokens. If the same dynamic token is repeated as many times as static tokens, it will be misclassified by ULP. One way to address this is to improve the pre-processing step by targeting such variables.

Additionally, as opposed to other log parsers, ULP makes no assumptions about the position of a static or dynamic token. Drain, for example, assumes that a token that appears in the beginning of a log message is a static token, which is not always valid. Furthermore, ULP is able to detect dynamic tokens and log templates from a variety of unknown log files without using domain knowledge regular expressions during the pre-processing stage such it is the case for Drain [26] and Logram [27]. ULP leverages only generic regular expressions.

Another assumption made by Drain's authors is that log events with a similar length belong to the same group without necessarily checking the content of the events, which results in classifying very different log events into the same group. ULP overcomes this problem by applying a rigorous string matching technique to ensures that log events can only be grouped together if they share the same static tokens. In Table IX, we summarized some of the parsing errors in Drain caused by the assumptions in the design of the tool. As for Logram, one of its main limitations consists of the way it deals with log events that appear only once. For these events, the whole log template is considered by Logram to be composed of only dynamic variables. Another limitation of logram is related to the use of n-grams, which leads to the situation where n-gram sequences may be considered as dynamic variables if they do not occur as frequently as other

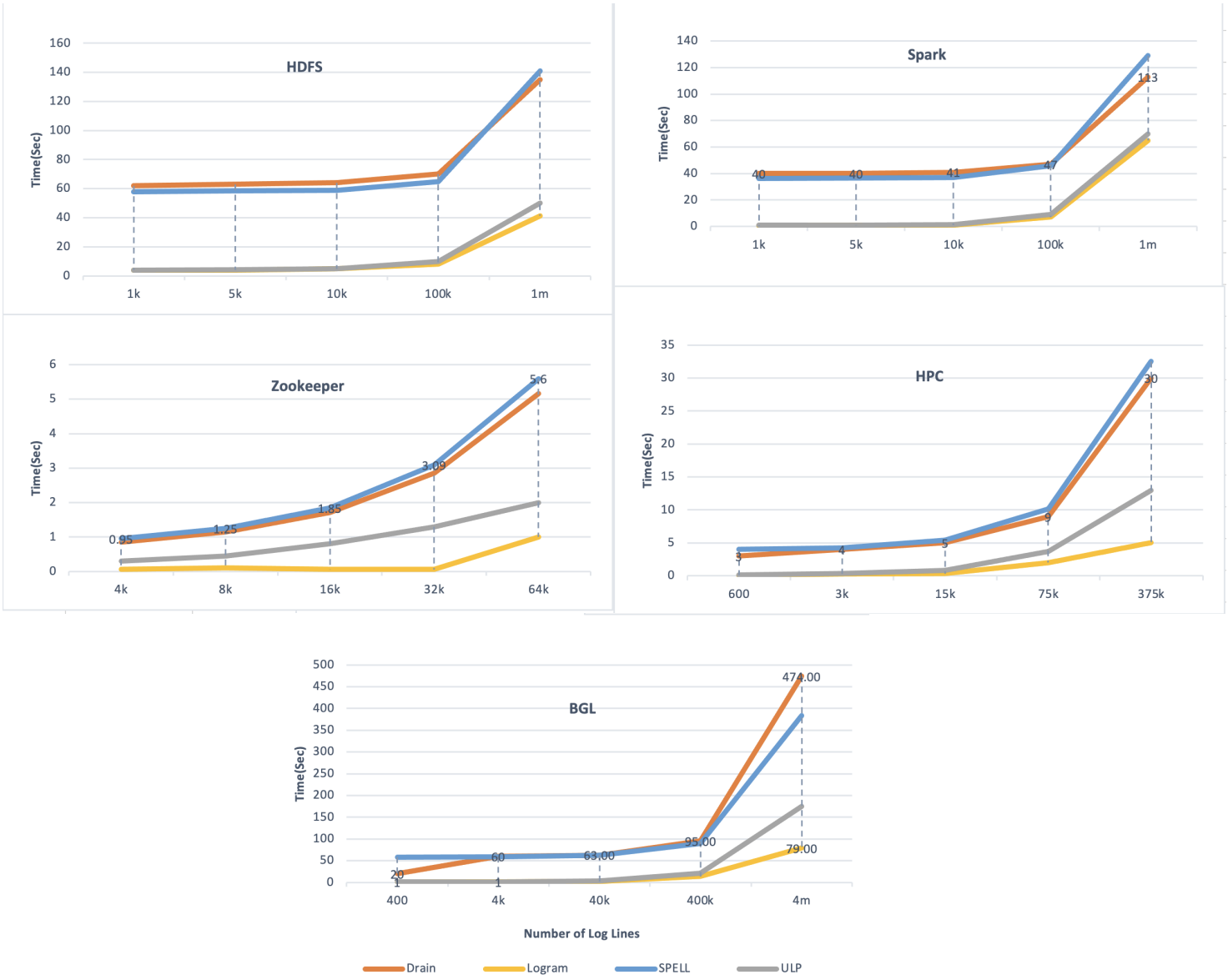

Fig. 4. Results of ULP efficiency. The x-axis represents the number of log events and the y-axis represents the execution time in seconds

n-grams. For example, in the log event "`Resolved 04DN8IQ.fareast.corp.microsoft.com to /default-rack`", the 2-gram "`Resolved 04DN8IQ.fareast.corp.microsoft.com`" appears only twice in the log file as opposed to the 2-gram "`to /default-rack`" which appears more frequently, the template generated for this log event is "`<*> <*> to /default-rack`", which is not valid ("`Revolved`" should be detected as a static token).

In some ways, our approach is closer in principle to that of AEL. It is possible to categorize log events based on linguistic commonalities using the AEL approach. However, starting with simple dynamic patterns, AEL uses hard-coded algorithms based on system information (e.g., IP addresses, numbers, and memory locations) to identify more complex patterns. The generated log events are then tokenized and binned based on the number of words they contain. This categorization evaluates the log messages in each bin before abstracting them into templates for use elsewhere in the system. The difficulty with AEL is that it assumes that events with the same number of words belong to the same group, resulting in many false positives when analyzing log events. ULP makes use of string matching similarity, which combines static tokens and the number of tokens in a log event, to overcome this problem.

VI. Threats to Validity

In this section, we discuss the threats to the validity of this study, which are organized as internal, external, conclusion, and reliability threats.

TABLE IX
EXAMPLE OF FLAWS INTRODUCED BY DRAIN PARSING ASSUMPTIONS

Log event	Drain grouping	Ground truth grouping	ULP grouping
attempt_123 TaskAttempt Transitioned from NEW to UNASSIGNED	<*><*>Transitioned from <*>to <*>	attempt_<*>TaskAttempt Transitioned from NEW to UNASSIGNED	<*>TaskAttempt Transitioned from NEW to UNASSIGNED
task_123 Task Transitioned from NEW to SCHEDULED	<*><*>Transitioned from <*>to <*>	task_<*>Task Transitioned from NEW to SCHEDULED	<*>Task Transitioned from NEW to SCHEDULED
task_123 Task Transitioned from SCHEDULED to RUNNING	<*><*>Transitioned from <*>to <*>	task_<*>Task Transitioned from SCHEDULED to RUNNING	<*>Task Transitioned from SCHEDULED to RUNNING
attempt_123TaskAttempt Transitioned from RUNNING to SUCCESS_CONTAINER_CLEANUP	<*><*>Transitioned from <*>to <*>	attempt_<*>TaskAttempt Transitioned from RUNNING to SUCCESS_CONTAINER_CLEANUP	<*>TaskAttempt Transitioned from RUNNING to SUCCESS_CONTAINER_CLEANUP
attempt_123 TaskAttempt Transitioned from RUNNING to FAIL_CONTAINER_CLEANUP	<*><*>Transitioned from <*>to <*>	attempt_<*>TaskAttempt Transitioned from RUNNING to FAIL_CONTAINER_CLEANUP	<*>TaskAttempt Transitioned from RUNNING to FAIL_CONTAINER_CLEANUP
kernel time sync disabled 12:56	kernel time sync <*><*>	kernel time sync disabled <*>	kernel time sync disabled <*>
kernel time sync enabled 09:45	kernel time sync <*><*>	kernel time sync enabled <*>	kernel time sync enabled <*>

Internal validity: Internal validity risks are those factors that have the potential to influence our outcomes. It is possible that mistakes happened during the implementation and testing of ULP, despite our best efforts. In order to reduce this hazard, we tested the tool on a large number of log files and manually reviewed its results on a limited number of samples. In addition, we make the tool and data accessible on Zenodo so that researchers may run the tool and check the results. Finally, in order to determine the correctness of ULP, we had to look at the disparities between the findings provided by ULP and the results acquired by the ground truth. This was accomplished in part via scripts and in part through manual checks. All efforts were made to minimize the possibility of mistakes.

Reliability Validity:. The potential of reproducing this research is referred to as reliability validity. The study's evaluation, replication, and reproducibility are made easier by the use of an open-source program. ULP and all the data used in this paper are available online on Zenodo[5].

Conclusion validity. The accuracy of the collected findings corresponds to the validity risks associated with the conclusion. We used ULP to analyze ten log files that have been commonly used in comparable investigations in the past. The accuracy and efficiency experiments were thoroughly reviewed to verify that the findings were appropriately interpreted, and we made every effort to do so. The tool and data that were used in every phase of this study are made accessible online to enable the evaluation and replication of our findings.

External validity: The generalizability of the findings is what is meant by external validity. We tested our findings on a total of ten log files from a variety of different software systems. Ten log files from the LogPai benchmark were used to evaluate ULP's performance. As a result, we cannot say with certainty that ULP's accuracy would be the same if it were applied to other datasets. However, since these datasets span a wide range of software systems from a variety of areas, they serve as a useful testbed for log parsing and analysis techniques. We do not claim that our findings can be applied to all available log files, particularly industrial and proprietary logs, to which we did not have access when conducting this research. We are currently working with industrial partners to apply ULP to their logs.

[5]http://zenodo.org/record/6425919

VII. CONCLUSION

We presented ULP, a powerful log parsing approach and tool. ULP differs from other tools in its design. It uses a novel way to distinguish between static and dynamic tokens of log events by applying string matching similarity to create groups of similar log events, and local frequency analysis to instances of the same group to distinguish between static and dynamic tokens. By doing so, ULP is capable with high accuracy to extract log templates that can be used to recognize and structure log events. Our approach confirms that is indeed possible to create an effective and efficient universal log parsers, which eliminates the need to develop specific parsers for various types of log files. Moreover, ULP is readily usable by practitioners to support various maintenance tasks that rely on log analytics. ULP is more accurate in parsing a representative set of 10 log files of the LogPai project than four leading open source log parsers. ULP is very efficient too. It took 3 minutes to parse up to 4 million Log events.

Future work should build on this work by focusing on the following aspects (a) apply ULP to more logs, especially those from industrial systems, (b) improve ULP by adding more regular expressions to identify other trivial dynamic variables such as domain-specific variables, and (c) conduct a time algorithm complexity analysis to determine with precision the best, worse, and expected running time of ULP, (d) improve the efficiency of the tool when applied to log files with a large number of templates, with high variability, and (e) adapt ULP to online parsing, which eliminates the need of a training set.

REFERENCES

[1] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in

Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 117–132. [Online]. Available: https://doi.org/10.1145/1629575.1629587

[2] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: system log analysis for anomaly detection," in *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*. IEEE, 2016, pp. 207–218.

[3] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016, pp. 102–111.

[4] M. Islam, K. Wael, and A. Hamou-Lhadj, "Anomaly detection techniques based on kappa-pruned ensembles," *IEEE Transactions on Reliability*, vol. 67, no. 1, pp. 212–229, 2018.

[5] T. Barik, R. DeLine, S. M. Drucker, and D. Fisher, "The bones of the system: a case study of logging and telemetry at microsoft," in *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016), Companion Volume*, 2016, pp. 92–101.

[6] J. Cito, P. Leitner, T. Fritz, and H. C. Gall, "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 (ESEC/FSE 2015)*, 2015, pp. 393–403.

[7] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang, "Capturing and enhancing in situ system observability for failure detection," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)*, 2018, pp. 1—-16.

[8] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007)*, 2007, pp. 575—584.

[9] Q. Lin, K. Hsieh, Y. Dang, K. Zhang, H.and Sui, Y. Xu, J. Lou, C. Li, Y. Wu, R. Yao, R. Chintalapati, and D. Zhang, "Predicting node failure in cloud service systems," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE 2018)*, 2018, pp. 480—490.

[10] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM 2008)*, 2008, pp. 307—-316.

[11] S. He, Q. Lin, J. Lou, H. Zhang, M. R. Lyu, and D. Zhang, "Identifying impactful service system problems via log analysis," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE 2018)*, 2018, pp. 60—-70.

[12] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 26–26.

[13] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, p. 217–231.

[14] Y. Dang, Q. Lin, and P. Huang, "Aiops: Real-world challenges and research innovations," in *Proceedings of the 41st International Conference on Software Engineering (ICSE 2019), Companion Volume*, 2019, pp. 4—-5.

[15] Q. Fu, J.-G. Lou, Q. Lin, R. Ding, D. Zhang, and T. Xie, "Contextual analysis of program logs for understanding system behaviors," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 397–400.

[16] R. Zhou, M. Hamdaqa, H. Cai, and A. Hamou-Lhadj, "Mobilogleak: A preliminary study on data leakage caused by poor logging practices," in *Proceedings of the Proc. of the International Conference on Software Analysis, Evolution, and Reengineering (SANER 2020), ERA Track*, 2020, pp. 577–581.

[17] R. Vaarandi and M. Pihelgas, "Logcluster - a data clustering and pattern mining algorithm for event logs," in *2015 11th International Conference on Network and Service Management (CNSM)*, 2015, pp. 1–7.

[18] M. Lemoudden and B. E. Ouahidi, "Managing cloud-generated logs using big data technologies," in *Proceedings of the International Conference on Wireless Networks and Mobile Communications, WINCOM 2015*, 2015, pp. 1—7.

[19] A. Miransky, A. Hamou-Lhadj, E. Cialini, and A. Larsson, "Operational-log analysis for big data systems: Challenges and solutions," *IEEE Software*, vol. 33, no. 2, pp. 55–59, 2016.

[20] D. El-Masri, P. Fabio, Y.-G. Guéhéneuc, A. Hamou-Lhadj, and A. Bouziane, "A systematic literature review on automated log abstraction techniques," *Information and Software Technology*, vol. 122, pp. 106–276, 02 2020.

[21] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora, "An exploratory study of the evolution of communicated information about the execution of large software systems," in *2011 18th Working Conference on Reverse Engineering*, 2011, pp. 335–344.

[22] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin, "Assisting developers of big data analytics applications when deploying on hadoop clouds," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 402–411.

[23] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. IEEE Press, 2019, pp. 121–130.

[24] 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*, vol. 20, no. 4, p. 249–267, 2008.

[25] H. Mi, H. Wang, Y. Zhou, M. R.-T. Lyu, and H. Cai, "Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 6, pp. 1245–1255, 2013.

[26] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *Web Services (ICWS), 2017 IEEE International Conference on*. IEEE, 2017, pp. 33–40.

[27] H. Dai, H. Li, W. Shang, T.-H. Chen, and C.-S. Chen, "Logram: Efficient log parsing using n-gram dictionaries," pp. 2–9, 2020.

[28] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *Data Mining (ICDM), 2016 IEEE 16th International Conference on*. IEEE, 2016, pp. 859–864.

[29] 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.

[30] R. Vaarandi, "Mining event logs with slct and loghound," in *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*. IEEE, 2008, pp. 1071–1074.

[31] ——, "A data clustering algorithm for mining patterns from event logs," in *IP Operations & Management, 2003.(IPOM 2003). 3rd IEEE Workshop on*. IEEE, 2003, pp. 119–126.

[32] 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.

[33] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 2009, pp. 149–158.

[34] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen, "Logmine: fast pattern recognition for log analytics," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, 2016, pp. 1573–1582.

[35] S. Kobayashi, K. Fukuda, and H. Esaki, "Towards an nlp-based log template generation algorithm for system log analysis," in *Proceedings of The Ninth International Conference on Future Internet Technologies*. ACM, 2014, p. 11.

[36] J. Lafferty, A. McCallum, and F. C. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," in *Proceedings of the International Conference on Machine Learning*, 2001, pp. 282–289.

[37] S. Thaler, V. Menkonvski, and M. Petkovic, "Towards a neural language model for signature extraction from forensic logs," in *Digital Forensic and Security (ISDFS), 2017 5th International Symposium on*. IEEE, 2017, pp. 1–6.

[38] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "An evaluation study on log parsing and its use in log mining," in *2016 46th Annual IEEE/IFIP*

International Conference on Dependable Systems and Networks (DSN). IEEE, 2016, pp. 654–661.

[39] K. Shima, "Length matters: Clustering system log messages using length of words," *CoRR*, vol. abs/1611.03213, 2016.

[40] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma, "Cliff´s delta calculator: A non-parametric effect size program for two groups of observations," *Universitas Psychologica*, vol. 10, no. 2, pp. 545–555, 2010. [Online]. Available: https://app.dimensions.ai/details/publication/pub.1113188555

[41] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen'sd indices the most appropriate choices," in *annual meeting of the Southern Association for Institutional Research*. Citeseer, 2006, pp. 1–51.