Investigating and Improving Log Parsing in Practice

Ying Fu
School of Big Data and Software Engineering, Chongqing University
Chongqing, China
fuying@cqu.edu.cn

Meng Yan*
School of Big Data and Software Engineering, Chongqing University
Chongqing, China
mengy@cqu.edu.cn

Jian Xu
Ant Group
China
jimmy.xj@antgroup.com

Jianguo Li*
Ant Group
China
lijg.zero@antgroup.com

Zhongxin Liu
College of Computer Science and Technology, Zhejiang University
Hangzhou, China
liu_zx@zju.edu.cn

Xiaohong Zhang
School of Big Data and Software Engineering, Chongqing University
Chongqing, China
xhongz@cqu.edu.cn

Dan Yang
School of Big Data and Software Engineering, Chongqing University
Chongqing, China
dyang@cqu.edu.cn

ABSTRACT

Logs are widely used for system behavior diagnosis by automatic log mining. Log parsing is an important data preprocessing step that converts semi-structured log messages into structured data as the feature input for log mining. Currently, many studies are devoted to proposing new log parsers. However, to the best of our knowledge, no previous study comprehensively investigates the effectiveness of log parsers in industrial practice. To investigate the effectiveness of the log parsers in industrial practice, in this paper, we conduct an empirical study on the effectiveness of six state-of-the-art log parsers on 10 microservice applications of Ant Group. Our empirical results highlight two challenges for log parsing in practice: 1) **various separators**. There are various separators in a log message, and the separators in different event templates or different applications are also various. Current log parsers cannot perform well because they do not consider various separators. 2) **Various lengths due to nested objects**. The log messages belonging to the same event template may also have various lengths due to nested objects. The log messages of 6 out of 10 microservice applications at Ant Group with various lengths due to nested objects. 4 out of 6 state-of-the-art log parsers cannot deal with various lengths due to nested objects. In this paper, we propose an improved log parser named Drain+ based on a state-of-the-art log parser Drain. Drain+ includes two innovative components to address the above two challenges: a statistical-based separators generation component, which generates separators automatically for log message splitting, and a candidate event template merging component, which merges the candidate event templates by a template similarity method. We evaluate the effectiveness of Drain+ on 10 microservice applications of Ant Group and 16 public datasets. The results show that Drain+ outperforms the six state-of-the-art log parsers on industrial applications and public datasets. Finally, we conclude the observations in the road ahead for log parsing to inspire other researchers and practitioners.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**.

KEYWORDS

Log parsing, Log analysis, Industrial study

ACM Reference Format:
Ying Fu, Meng Yan, Jian Xu, Jianguo Li, Zhongxin Liu, Xiaohong Zhang, and Dan Yang. 2022. Investigating and Improving Log Parsing in Practice. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3558947

1 INTRODUCTION

Logs are designed to record important information during systems running, such as important status, resource information, and business operation information. The rich running information in the logs enables practitioners to detect system anomalies (e.g., [3] [24] [27] [37] [45]), diagnose failures (e.g., [1] [4] [21] [30]), and predict failures (e.g., [2] [9] [10] [25]). Log messages generated by log statements in source code are semi-structured because the developers can use free text to record the important information in log statements based on their experience. Due to the lack of rigorous logging guidance, the logging styles of different developers in the same project are diverse.

*Corresponding author

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3558947

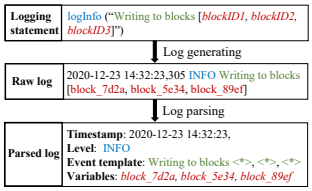

Figure 1: An example of log parsing.

The goal of log parsing is to extract the important information in the free text of raw log messages and convert them into a structured format, as Figure 1 shows. Most of automatic log analysis methods use structured log messages as input (e.g., [5] [12] [26] [29] [32] [48]). Therefore, log parsing is an essential process for automatic log analysis methods. With the development of modern software systems, the volume, complexity, and diversity of log messages are increasing. The traditional way of manually designing and maintaining regex for log parsing becomes very difficult. Therefore, many automatic log parsers are proposed to assist developers in software system maintenance activities and service quality assurance operations, such as anomaly detection, failure diagnosis, and performance diagnosis and improvement. Automatic log parsers play an important role in Artificial Intelligence for IT Operations (AIOps) [8] [13] [19] [20] [25].

Traditional source-code-based log parsers [36] [47] required to access the source code. While accessing the source code is not always possible such as third-party libraries. Data-driven log parsers can overcome this limitation of source-code-based log parsers. Data-driven log parsers apply data mining techniques to parse the log messages. Data-driven log parsers can be divided into frequent-pattern-mining-based, clustering-based, heuristic-based, and others according to the technology adopted. Frequent-pattern-mining-based log parsers include SLCT [43], LFA [35], LogCluster [44], and Logram [6]. The performance of this kind of log parser is sensitive to the threshold of frequent itemsets. Clustering-based log parsers include LKE [14], LogSig [42], LogMine [15], SHISO [34], and Lenma [40]. This kind of log parser mainly uses various similarity measures for clustering. The performance of this kind of log parser is sensitive to the pre-defined similarity threshold. Heuristic-based log parsers include AEL [22], IPLoM [31], and Drain [18]. This kind of log parser requires more manual participation when constructing grouping conditions. In addition to the above three kinds of log parsers, there are some other log parsers, such as Spell [11], which is based on the longest common subsequence, and MoLFI [33], which is based on the evolutionary algorithm. Although many log parsers have been proposed, only two recent studies [6] [49] comprehensively evaluate the log parsers on public datasets. To the best of our knowledge, no previous study comprehensively evaluate the log parsers on industrial projects.

To investigate the effectiveness of data-driven log parsers on industrial projects, we select one or two representative and state-of-the-art log parsers from each category and evaluate their effectiveness on the microservice applications of Ant Group[1] comprehensively. Six state-of-the-art log parsers (AEL, IPLoM, Lenma, Spell, Drain, and Logram) are selected. The average parsing accuracy of these log parsers ranges from 0.062 to 0.271, and the F-measure ranges from 0.240 to 0.639. While the average parsing accuracy of these log parsers on 16 public datasets ranges from 0.570 to 0.864, and the F-measure ranges from 0.823 to 0.977. These results show that the six log parsers perform significantly worse on industrial microservice applications of Ant Group than on public datasets. By investigating why the six log parsers perform worse on the microservice applications of Ant Group and studying the characteristics of log messages of the microservice applications, we highlight two challenges for log parsing in practice.

(1) **Various separators**. Various separators are used in all 10 microservice applications of Ant Group. There are various separators in a log message, and the separators in different event templates or different applications are also various, as shown in Figure 2. Case 1, 2, and 3 are three raw log messages of microservice applications at Ant Group. Due to confidentiality reasons, some information in the log messages is represented by *. Case 1 and Case 2 are generated by different event templates of an application. Case 2 and Case 3 are generated by different applications. We can observe that the separators used in the three cases are different (Case 1: ", [ ]", Case 2: ", ( )", and Case 3: ", [ ] ( )"). It is hard to decide which separators are used when parsing log messages. Current log parsers use the uniform separator (blank) for parsing log messages. The mis-splitting caused by various separators will significantly impact the effectiveness of the log parser, and all the log parsers do not consider such a challenge.

(2) **Various lengths due to nested objects**. The log messages belonging to the same event template may also have various lengths due to nested objects. The length of the log message refers to the number of split tokens. A nested object is a composite of container data types (e.g., dictionary and list) and atomic data types (e.g., bool and string). As Figure 3 shown, in Case 5, the nested object is a hierarchical composite of dictionary, bool, string, and list. The log messages of 6 out of 10 microservice applications have various lengths due to nested objects. The average proportion of the log messages with various lengths is 0.256 on the 10 microservice applications. As Figure 3 shown, Case 4 and Case 5 are the log messages of Ant Group which belong to the same event template. For confidentiality reasons, some information in the log messages is represented by *. Due to the null value of the nested objects in Case 4, the lengths of Case 4 and Case 5 are different. Four of the six studied log parsers (AEL, IPLoM, Lenma, and Drain) cannot deal with this case since they assume that the length of the log messages' content belonging to the same log event should be the same.

In this paper, we propose an improved log parser named Drain+ based on a state-of-the-art log parser Drain. Drain+ includes two innovative components to address the above challenges: a statistical-based separators generation component, which generates separators automatically for log message splitting, and a candidate event template merging component, which merges the candidate event templates by a template similarity method.

Specifically, firstly, we comprehensively study the effectiveness of six state-of-the-art log parsers (AEL, IPLoM, Lenma, Spell, Drain, and Logram) on 10 microservice applications of Ant Group. Secondly, we investigate why the state-of-the-art log parsers perform significantly worse on these microservice applications. And we summarize two challenges that the log parsers encountered. Thirdly,

[1]Ant Group, formerly known as Ant Financial, is the owner of one of the world largest mobile online payment services Alipay.

Case 1:
2020/12/1,16:53:42,441,"**hRuleDecisionHelper,2041436524461,**SwitchResult[is**Out=false,**Value=<null>][0b40a2,218.**.165.**.20]"

Case 2:
2020/12/1,16:54:04,647,"****: **,AccountNo=<*>,20156(0b40797e6627,218.**.165.**)

Case 3:
2020-12-17 15:23:34,230 [/// -] WARN **-**-ACTION ****[284(TradeResult[resultCode=AE02,tradeError=TradeError[childErrors=[],clazzName=com.***,code=****,parameters=<null>,level=*,info=<null>,property=<null>],success=false,errorContext=<null>**EventContext=**EventContext[**Code=MC48,****Code=MC51,******Code=519,traceID=0185],resultExtInfo=<null>])(0b2)][0b85, 18.**.165.**]

Figure 2: Examples of various separators in the log messages of Ant Group.

Case 4:
sys|m_c_p|2efefae3b-725-42bc-ber2-c746exwwqeRZ82,207f2802a4,Chr_ex,isSupport:false,{brower_bit=<null>,os=true,bus=false,os_ver=true,b_ls=ture,mhc_b=true,mobces=true,j_b= false,mhc}

Case 5:
sys|m_c_p|20cfae3b-7d2a-42bc-bb69-c746ebb7fRZ82,2e02e4f734,adit,isSupport:false,{brower_bit={create_*:[*,*,*]*, …},os=true,bus=true,os_ver=true,b_ls=ture,mhc_b=true,mobces=true,j_b=true,*}

Figure 3: Examples of various lengths due to nested objects in the log messages of Ant Group.

we propose a log parser named Drain+ that includes two innovative components to deal with the two challenges of log parsing, especially on industrial projects. Fourthly, we investigate the effectiveness of two innovative components of Drain+ (automatic separator generation and template merging). Finally, we conclude the observations in the road ahead for log parsing to inspire other researchers and practitioners. In summary, the main contributions of the paper are as follows:

(1) We conduct an empirical study to evaluate the effectiveness of six log parsers on industrial microservice applications. The results show that current log parsers cannot perform well in practice compared to their effectiveness on public datasets. We highlight two challenges that limit the effectiveness of log parsing in practice, i.e., various separators and various lengths due to nested objects.

(2) We propose an improved log parser named Drain+ based on Drain. Drain+ includes two innovative components: a statistical-based separators generation component, which can deal with various separators, and a candidate event template merging component, which can deal with various lengths due to nested objects.

(3) We evaluate the effectiveness of Drain+ on 10 microservice applications of Ant Group and 16 public datasets. The evaluation results show that Drain+ outperforms the six state-of-the-art log parsers on industrial microservice applications and public datasets.

Paper organization. The remainder of the paper is organized as follows. Section 2 presents the background of log parsing. Section 3 presents our study setup, including the log datasets, the log parsing process, the studied log parsers, and the used evaluation measures. Section 4 presents an empirical study on the log messages of microservice applications at Ant Group. Section 5 presents the design of Drain+. Section 6 presents our study results. Section 7 discusses the threats to the validity of our evaluation and how far are log parsing. Section 8 surveys prior work related to log parsing. Section 9 concludes the paper.

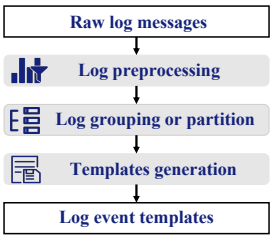

Figure 4: The framework of log parsing.

2 BACKGROUND

2.1 An Overview of Log Parsing Process

The log parsing process involves several key steps, as Figure 4 shows:

Log preprocessing. This step mainly contains two processing: 1) Content preprocessing. Log parsers extract the log messages' content by using pre-defined regular expressions and removing some common variables by regular expressions, such as IP addresses. The regular expressions used in the two processing are defined manually based on domain knowledge. It is worth noting that the processing of common variables is not a necessary step. Some log parsers need processing before parsing, while others can parse raw logs directly. 2) Word splitting. After extracting the content of log messages, the content needs to be split into tokens. This is an important processing step before applying various techniques for log grouping or log partition. The effect of the content splitting has a significant impact on the log parsing accuracy.

Log grouping or partition. After content splitting, different log parsers use different techniques to group or partition the log messages. This step is the most important processing of log parsing. The log messages that are considered to belong to the same template by the log parser are grouped.

Templates generation. After log messages grouping, the event templates contained in log messages are generated. The tokens contained in all log messages which belong to the same group are considered static text. The other tokens are considered dynamic variables and replaced by the pre-defined dynamic variables symbols. It is worth noting that the frequent occurrence of dynamic variables can have a negative impact on log parsing.

2.2 Studied Automatic Log Parsers

Currently, many data-driven-based log parsers have been proposed. The data-driven-based log parsers can be classified into four categories according to the techniques adopted, including heuristics-based, clustering-based, frequent-pattern-mining-based, and longest-common-subsequence-based. We select one or two state-of-the-art log parsers from each category as the representatives and evaluate their effectiveness on the microservice application of Ant Group. Six log parsers are selected, including AEL [22], IPLoM [31], Lenma [40], Spell [11], Drain [18], and Logram [6], based on the following criteria: 1) The log parser does not rely on source code since we

Table 1: The studied log parsers.

Log parser	Technique	Preprocessing	Year
AEL	Heuristics	Yes	2008
IPLoM	Heuristics	No	2012
Lenma	Clustering	No	2016
Spell	Longest common subsequence	No	2016
Drain	Heuristics	Yes	2017
Logram	Frequent pattern mining	Yes	2020

cannot access the source code from which the logs are generated; 2) The log parser is of high efficiency; 3) The log parser is representative and the state-of-the-art one in its corresponding category. Table 1 shows the studied log parsers in our study. The second column represents the technique the log parser adopted, and the third column represents whether the log parser requires a preprocessing before starting log parsing.

To make our paper self-contained, we briefly introduce these approaches:

AEL. AEL is a heuristic-based method proposed by Jiang et al. [22]. Firstly, AEL uses heuristics to recognize the dynamic tokens in log messages. Secondly, it separates the log messages into different groups according to each log message's number of words and parameters. Thirdly, generating the event templates.

IPLoM. IPLoM (Iterative Partitioning Log Mining) is a heuristic-based method proposed by Makanju et al. [31]. IPLoM adopts an iterative partition strategy based on the characteristics of log messages to parse logs. IPLoM parses log by three steps iterative partitioning process. Firstly, partition by the number of tokens in log content. In this process, the log messages with the same token length are partitioned into the same groups. The log messages with different token lengths are partitioned into different groups. Secondly, partition by token position. In this process, IPLoM partitions the log messages by the column with the least number of variables. Thirdly, partition by the search for bijection mapping and generating log event template from every cluster.

Lenma. Lenma is a clustering-based method proposed by Shima et al. [40]. Firstly, Lenma creates a word length vector and word vector for each log message. Secondly, it calculates the similarity between two log messages and clusters of log messages into a group whose similarity value is larger than the pre-defined threshold. Thirdly, it generates a log event template for each cluster that has the same number of words.

Spell. Spell (Streaming parser for event logs using longest-common-subsequence) is proposed by Du et al. [11]. Spell uses a longest-common-subsequence-based method to parse log messages. Spell computes the longest common subsequence of two log messages to generate the static text of the event template.

Drain. Drain (Depth tree based online log parsing) is a heuristic-based method proposed by He et al. [18]. Drain uses a fixed-depth tree to represent the hierarchical relationship between log messages. In the first layer of the tree, Drain uses the length of log message content to group the log messages. So the nodes in the first layer of the tree are used to represent the log groups whose log messages are in different log message lengths. The second layer of the tree uses the tokens in the beginning positions of the log messages to select the next internal node. The third layer of the tree calculates the similarity between the log message and the log event template of each log group.

Logram. The Logram is a frequent-pattern-mining-based method proposed by Dai et al. [6]. Logram leverages n-grams to parse log messages. Firstly, Logram uses the log tokens extracted from each log message to build a 2-gram dictionary and a 3-gram dictionary. Secondly, it parses log messages by the previously built n-gram dictionaries. Thirdly, Logram generates an event template for each log message.

Drain, AEL, and Logram usually need to preprocess log messages with domain knowledge which is used to remove some common variables before log parsing to improve parsing accuracy further. While IPLoM, Lenma, and Spell can parse raw log messages directly to achieve a good performance [17].

Drain, IPLoM, AEL, and Lenma use the length of log message content when grouping or partitioning the log messages. They all assume that the length of the log message content belonging to the same log event should be the same. However, some log messages with various lengths belong to the same log event in the industrial environment. This assumption will limit the performance of these four parsers in industrial practice. Spell and Logram do not use the length of log message content. Spell computes the longest common subsequence of two log messages to generate the static text of the event template. It assumes the total length of parameter values in a log is more than half of its size. This assumption will be voidable in practice. Logram is different from the other log parsers that it uses the frequent n-grams in the log messages. The core insight of Logram is that the frequent n-grams are more likely to be static texts of event templates. Therefore, the accuracy of Logram is sensitive to the pre-defined threshold of n-gram frequency.

3 EXPERIMENTAL SETUP

3.1 Dataset

Data selection. We use the datasets[2] from Ant Group and public datasets [49]. A summary of the log messages produced by the microservice applications can be seen in Table 2. A summary of 16 public datasets can be seen in Table 3. We study the log messages produced by 10 microservice applications of Ant Group. We refer to the applications as App_1 to App_{10} due to confidentiality reasons. We select these microservice applications as: 1) Currently, these microservice applications are running online and used by millions of users on a daily basis, and a large volume of the latest log messages can be collected; 2) They differ in purposes; 3) We can conveniently communicate with these applications' operation and maintenance person when needed.

Data labeling. Data labeling aims to extract the event templates from log messages. Zhu et al. [49] already extracted the event templates manually for the public datasets. For the log messages from Ant Group, 2,000 log messages were randomly selected from each microservice application for manual event template extraction as ground truth followed Zhu et al. In Table 2 and Table 3, the Templates column presents the number of event templates in the each microservice application. The Avg (length) column presents the average length of log messages in each microservice application.

[2]The dataset does not contain any Personal Identifiable Information (PII) and is desensitized and encrypted. Adequate data protection was carried out during the experiment to prevent the risk of data copy leakage, and the data set was destroyed after the experiment. Besides, the data set is only used for academic research and does not represent any real business situation.

Table 2: Summary of the log messages produced by the microservice applications of Ant Group.

App	Templates	Avg (length)	Messages
App_1	20	17.00	2,000
App_2	14	23.25	2,000
App_3	5	99.46	2,000
App_4	11	12.25	2,000
App_5	61	17.18	2,000
App_6	18	16.39	2,000
App_7	13	215.68	2,000
App_8	6	6.65	2,000
App_9	10	49.96	2,000
App_{10}	9	10.62	2,000

Table 3: Summary of 16 public datasets.

Dataset	Templates	Avg (length)	Messages
HDFS	14	7.44	2,000
Hadoop	114	8.19	2,000
Spark	36	8.76	2,000
Zookeeper	50	6.30	2,000
BGL	120	6.32	2,000
HPC	46	3.48	2,000
Thunderbird	149	8.51	2,000
Windows	50	7.93	2,000
Linux	118	8.30	2,000
Andriod	166	5.40	2,000
HealthApp	75	2.80	2,000
Apache	6	6.28	2,000
Proxifier	8	9.35	2,000
OpenSSH	27	8.56	2,000
OpenStack	43	9.01	2,000
Mac	341	9.17	2,000

For the public datasets, we use white space to split the log messages to compute the length. For the log messages from Ant Group, we use the separators that are used in the microservice application.

3.2 Evaluation metrics

Accuracy Metric. To quantify the effectiveness of automated log parsers, we use **F-measure** and **PA** (Parsing Accuracy) used in prior studies [49] [18] as the accuracy metric. F-measure is a typical evaluation metric for clustering algorithms [39], and it is also used in the prior log parsing study [17]. The definition of F-measure is $F\text{-}measure = \frac{2*Precision*Recall}{Precision+Recall}$, where Precision and Recall are defined as $Precision = \frac{TP}{TP+FP}$, and $Recall = \frac{TP}{TP+FN}$. TP (True Positive) represents that two log messages with the same log event are grouped into the same group; FP (False Positive) represents that two log messages with different log events are grouped into the same group; FN (False Negative) represents that two log messages with the same log event are grouped into different groups.

PA is defined by Zhu et al. [49]. The log messages are considered to parse correctly if all log messages with the same log event are grouped in one group, and all log messages in the same group contain the same log event. The definition of PA is $PA = \frac{Correctly\ parsed\ log\ messages}{The\ total\ number\ of\ log\ messages}$.

4 AN EMPIRICAL STUDY

To understand the effectiveness of the studied log parsers and the challenges in industrial practice, we conduct an empirical study on industrial microservice applications of Ant Group. Here we target at the following research questions:

- **RQ1:** How effective are the state-of-the-art log parsers on industrial applications?
- **RQ2:** Why do the state-of-the-art log parsers perform significantly worse on industrial applications than on public datasets?

RQ1 aims to explore whether the six state-of-the-art log parsers are sufficiently good for industrial applications. RQ2 aims to investigate the challenges of log parsing in industrial practice.

4.1 RQ1: How effective are the state-of-the-art log parsers on industrial applications?

Motivation. Prior studies [6] [49] evaluate the effectiveness of the six studied log parsers on 16 public datasets. Their evaluation results show that these log parsers work well on most public datasets. In the evaluation of Zhu et al. [49], the average parsing accuracy of the five log parsers (Drain, IPLoM, Spell, AEL, and Lenma) on the 16 public datasets is greater than 0.7. In the evaluation of Dai et al. [6], the average accuracy of the three log parsers (Drain, AEL, and Logram) is greater than 0.7. However, how effective are the six studied log parsers on industrial applications in practice?

Methods. Firstly, we collect log messages produced by 10 microservice applications of Ant Group. Secondly, we randomly select 2,000 log messages from each microservice application and manually extract the event template as ground truth, following the study of Zhu et al. [49]. Thirdly, we use comma as the uniform separator to implement the six state-of-the-art log parsers. Please note that the original implementations of the six log parsers use blank as the uniform separator. However, blank is rarely used in the log messages of Ant Group, while comma is the most frequently used separator. Therefore, we choose to use comma instead of blank as the uniform separator. Fifthly, we run the six studied log parsers on 16 public datasets following the evaluation of Zhu et al. [49] and compare the effectiveness of the six log parsers on public datasets and the microservice applications of Ant Group.

Results. Table 4 shows the average effectiveness of six studied log parsers on 16 public datasets. Table 5 shows the effectiveness of six studied log parsers on 10 microservice applications. Based on these results, we make the following observations:

Table 5 shows the effectiveness of six studied log parsers on 10 microservice applications. The PA row presents the parsing accuracy of the log parser. We can observe that the average PA of six log parsers is less than 0.3, and the average F-measure of six log parsers is less than 0.8. While as Table 4 shown, the average parsing accuracy of six log parsers on 16 public datasets is greater than 0.5, and the average F-measure of six log parsers is greater than 0.8. The best average parsing accuracy of six log parsers on the 10 microservice applications is significantly worse than the worst average parsing accuracy of six log parsers on the 16 public datasets. And the best average F-measure of six log parsers on the 10 microservice applications is worse than the worst average F-measure of six log parsers on the 16 public datasets. This indicates that the effectiveness of six state-of-the-art log parsers is not sufficiently good for the log messages of industrial applications.

Summary for RQ1 👉 *The six state-of-the-art log parsers perform significantly worse on the microservices applications of Ant Group than on public datasets.*

Table 4: The average effectiveness of 6 studied log parsers on 16 public datasets.

Average effectiveness	AEL		IPLoM		Lenma		Spell		Drain		Logram	
	PA	F-measure	PA	F-measure	PA	F-measure	PA	F-measure	PA	F-measure	PA	F-measure
	0.791	0.968	0.756	0.968	0.766	0.936	0.793	0.961	0.864	**0.977**	0.570	0.823

The parsing accuracy highlighted in underline is the best among the six log parsers, and the F-measure highlighted in bold is the best among the six log parsers.

Table 5: The effectiveness of 6 studied log parsers on 10 microservice applications.

Dataset	AEL		IPLoM		Lenma		Spell		Drain		Logram	
	PA	F-measure	PA	F-measure	PA	F-measure	PA	F-measure	PA	F-measure	PA	F-measure
App_1	0.000	0.215	0.000	0.215	0.152	0.513	0.152	0.515	1.000	**1.000**	0.935	0.985
App_2	0.588	**0.879**	0.001	0.773	0.001	0.728	0.588	0.826	0.001	0.772	0.001	0.123
App_3	0.001	0.000	0.001	0.367	0.001	0.000	0.001	0.000	0.432	**0.842**	0.005	0.085
App_4	0.001	**0.878**	0.003	**0.878**	0.087	0.057	0.002	0.721	0.087	0.057	0.003	0.229
App_5	0.000	0.061	0.000	0.061	0.000	0.034	0.043	0.565	0.509	0.217	0.593	**0.284**
App_6	0.000	0.000	0.000	0.859	0.000	0.000	0.039	0.983	0.039	**0.983**	0.000	**0.983**
App_7	0.001	0.550	0.004	0.600	0.001	0.284	0.000	0.510	0.004	**0.602**	0.004	0.475
App_8	0.222	0.173	1.000	**1.000**	0.079	0.023	0.001	0.969	0.300	0.998	0.901	0.999
App_9	0.002	0.706	0.282	**0.942**	0.289	0.756	0.000	0.498	0.281	0.919	0.264	0.938
App_{10}	0.001	**0.934**	0.002	0.681	0.007	0.001	0.001	0.591	0.007	0.001	0.006	0.179
Average	0.082	0.440	0.129	0.638	0.062	0.240	0.083	0.618	0.266	**0.639**	0.271	0.528

The parsing accuracy highlighted in underline is the best among the six log parsers, and the F-measure highlighted in bold is the best among the six log parsers.

4.2 RQ2: Why do the state-of-the-art log parsers perform significantly worse on industrial applications than on public datasets?

Motivation. The results presented in Subsection 4.1 show that the average parsing accuracy of the six studied state-of-the-art log parsers is less than 0.3 on the microservice applications of Ant Group, which is significantly worse than on public datasets. The characteristics of log messages can affect the generalization ability of log parsers [33]. Does the change of log characteristics cause the poor effectiveness of the log parsers on industrial applications? In this subsection, we investigate the reasons for the poor effectiveness of the six log parsers on the microservice applications of Ant Group.

Methods. Firstly, we manually analyze the log messages that the six log parsers parse incorrectly to investigate the reasons that caused the poor effectiveness of the six log parsers. As a result, we observe two factors that may cause poor parsing effectiveness (i.e., various separators and various lengths due to nested objects). Secondly, we statistically analyze the distribution of the two factors on 10 microservice applications' log messages of Ant Group. Thirdly, we explore the impact of various separators by comparing the effectiveness of six studied log parsers using multi separators to implement vs. original using one uniform separator. We use three separators to implement the six log parsers and run the six log parsers on the 10 microservice applications. We chose to use these separators because the 10 microservice applications all used these separators. Fourthly, to explore the impact of various lengths due to nested objects, we investigate the distribution of various lengths due to nested objects in parsed correctly and incorrectly parsed log messages, respectively.

Results. As the results are shown in Table 6, Table 7, and Table 8 we make the following observations:

(1) As Table 6 shows, the Various separators column presents which separators are used in the microservice application. We can observe that various separators are used in all 10 studied microservice applications. **Various separators** are used in the log messages of microservice applications at Ant Group. Since the lack of rigorous specifications to guide developers' logging practices, different separators are used by the different developers in several log styles.

Table 6: The characteristics of the log messages of 10 microservice applications.

Dataset	Various-length proportion	Various separators
App_1	0.000	, [ ] ( ) -
App_2	0.413	, ; [ ] { } ( ) -
App_3	0.365	, [ ] ( )
App_4	0.161	, [ ] ( ) -
App_5	0.000	, [ ] ( ) -
App_6	0.000	, [ ] ( )
App_7	0.747	, | [ ] { } ( )
App_8	0.000	, [ ]
App_9	0.637	, ; [ ] { } ( ) -
App_{10}	0.234	, [ ] ( ) -
Average	0.256	

There are various separators in a log message, and the separators in different event templates or different applications are not the same. It is hard to decide which separators are used when parsing log messages. This is different from public datasets where the uniform separator (blank) is used in most log messages.

(2) As Table 6 shows, the Various-lengths proportion column presents the proportion of log messages that belong to the same event template with various lengths due to nested objects. If log messages belong to the same event template with various lengths, all of them are counted as various lengths log messages. As shown in Figure 3, Case 4 and Case 5 belong to the same event template, but they are of **various lengths due to needed objects**. The log messages of 6 out of 10 microservice applications with various lengths due to the nested objects, and the average proportion is 0.256.

(3) Table 7 shows the average effectiveness of six studied log parsers using multi separators to implement vs. original using one uniform separator on 10 microservice applications of Ant Group. We can observe that the average effectiveness of six log parsers using multi separators to implement is better than that of six log parsers using one uniform separator. This indicates that the separator has an impact on the effectiveness of the log parser. And using one uniform separator to split the log messages may degrade the effectiveness of log parsers when there are various separators in the log messages.

(4) Table 8 shows the average proportion of the log messages with various lengths in parsed correctly and incorrectly log messages.

Table 7: The average effectiveness of six studied log parsers using multi separators to implement vs. original using one uniform separator on 10 microservice applications.

Average effectiveness	AEL		IPLoM		Lenma		Spell		Drain		Logram	
	PA	F-measure	PA	F-measure	PA	F-measure	PA	F-measure	PA	F-measure	PA	F-measure
Original	0.082	0.440	0.129	0.638	0.062	0.240	0.083	0.618	0.266	0.639	0.271	0.528
Multi separators	**0.157**	**0.667**	**0.150**	**0.662**	**0.320**	**0.679**	**0.303**	**0.710**	**0.388**	**0.716**	**0.282**	**0.641**

Table 8: The average proportion of the log messages with various lengths in parsed correctly and incorrectly log messages.

Average proportion of log messages with various lengths	AEL		IPLoM		Lenma		Spell		Drain		Logram	
	parsed correctly	parsed incorrectly	parsed correctly	parsed incorrectly	parsed correctly	parsed incorrectly	parsed correctly	parsed incorrectly	parsed correctly	parsed incorrectly	parsed correctly	parsed incorrectly
	0	0.256	0	0.256	0	0.256	0	0.256	0	0.256	0	0.256

We can observe that all the log messages with various lengths are parsed incorrectly by all six log parsers. This indicates various lengths due to nested objects have an impact on the effectiveness of the log parsers.

By analyzing the reasons why the six state-of-the-art log parsers do not work well on the log messages of microservice applicaitons at Ant Group and studying the characteristics of log messages, we obtain the main problems that the log parsers fail to deal with. In summary, there are the following challenges:

- **Various separators**. There are various separators in a log messages. It is hard to decide which separators are used when parsing log messages.
- **Various lengths due to nested objects**. The log messages belong to the same event template may also have various lengths to nested objects. Four of the six studied log parsers (AEL, IPLoM, Drain, and Lenma) fail to deal with such cases since they assume that the length of the log messages' content belonging to the same log event should be the same.

Summary for RQ2 👉 *There are two challenges (various separators and various lengths due to nested objects) that cause the poor performance of log parsers on microservice applications of Ant Group.*

5 OUR APPROACH

In this section, we detail the core design of Drain+. The framework of Drain+ is shown in Figure 5. In the log partition step, we use the fixed depth tree of Drain to generate the candidate event templates, because: 1) As the evaluation results shown in Section 4.1, Drain has comparable average parsing accuracy with Logram which has the best average parsing accuracy on 10 Microservice applications of Ant Group and higher F-measure than Logram. And Drain achieves the best parsing accuracy and F-measure on more Microservice applications (5 and 4, respectively) of Ant Group than other log parsers. 2) In the prior study [49], the evaluation results show that Drain attains the highest accuracy on average and the smallest variance on the 16 public datasets. 3) Drain has high parsing efficiency [18]. The detailed step of Drain+ as follows.

Separators generating. Separators are generated for log message content splitting by a statistical-based method. Firstly, we define the candidate separator set. The candidate separators set includes ", | ; [ ] { } ( ) - ". Secondly, the occurrences of candidate separators contained in the log messages are counted for each dataset. Thirdly, the candidate separators that occur more than the predefined threshold of the log messages are selected as the separators. Here we set the default threshold to 0.2 and we discuss the impact of such threshold later. If there is a version update to the microservice application, the separators will be regenerated and updated.

Log preprocessing. In log preprocessing, we mainly perform two processing steps. Firstly, we extract the content of the log messages by using a pre-defined regular expression and replace the common variables such as IP addresses and simple URL addresses with variable symbols before log parsing based on domain knowledge. All the log messages of studied applications use the same regular expression. The previous study [17] has shown that the processing step of removing common variables can improve the parsing effectiveness. Secondly, we split the content of log messages by the separators generated in the generating separators process.

Log partition & templates generation. We use the fixed depth tree of Drain to generate the candidate event templates. Firstly, using the length of log messages content to partition the log messages. Secondly, using the beginning position token to partition the log messages further. Thirdly, generating the candidate templates.

Templates merging. The fixed depth tree uses the length of log message content to partition the log message in the first layer of the tree. It assumes that the length of log message content belonging to the same log event should be the same. While the length of log message content belonging to the same log event may not be the same. To reduce the negative impact caused by the assumption in the fixed depth tree, we use asymmetric Jaccard similarity to merge the candidate templates generated by the fixed depth tree. Jaccard similarity is computed as the number of shared terms over the number of all unique terms in both strings [38]. Firstly, we split all event templates into tokens and remove special symbols such as variable symbols. Only static text tokens are left. Secondly, generate event template tokens vector space by Sklearn [41]. The event templates T_1 and T_2 are transformed to vectors (T_{1vec}, T_{2vec}). Thirdly, we calculate the similarity of T_1 and T_2 based on the candidate template vectors. If the similarity of T_1 and T_2 is greater than the pre-defined threshold, we merge T_1 and T_2 and update the event template. Here we set the default threshold to 0.6 and we discuss the impact of such threshold later. When new log messages come, Drain+ calculates the similarity between the new candidate templates and the existing event template if new candidate templates are generated. If the similarity is greater than 0.6, Drain+ merges the new candidate templates into the current event templates; else, it creates new event templates.

$$J(T_1, T_2) = \frac{|T_{1vec} \cap T_{2vec}|}{|T_{1vec} \cup T_{2vec}|} \quad (1)$$

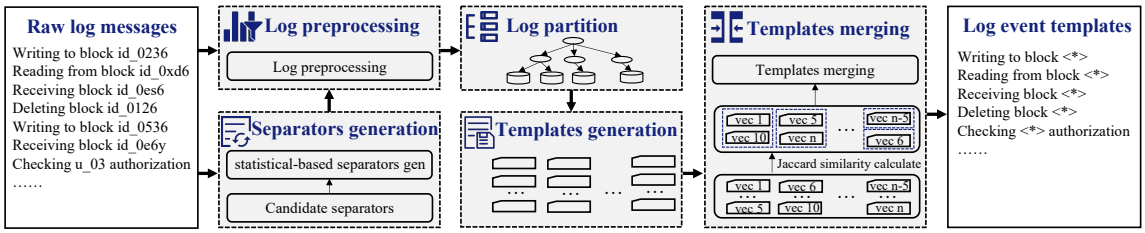

Figure 5: The framework of Drain+.

The main differences between Drain+ and Drain include: 1) Drain+ uses a statistical-based method to automatically generate separators for log message splitting instead of a uniform separator; 2) Drain+ uses template similarity to merge the candidate event templates to deal with the various lengths due to nested objects.

6 EVALUATION

6.1 Research Questions

In this paper, Drain+ makes two improvements to the Drain to deal with the challenges of log parsing on industrial applications described in Section 4.2. In this section, we intend to investigate the following two research questions:

RQ3: How effective is Drain+ on industrial applications of Ant Group and public datasets?

RQ4: How do the innovative components contribute to Drain+?

6.2 RQ3: How effective is Drain+ on industrial applications of Ant Group and public datasets?

Motivation. The two challenges (various separators and various lengths due to nested objects) in log parsing pose an obstacle to the six state-of-the-art log parsers. We propose an improved log parser named Drain+ that can deal with such challenges based on Drain.

In this section, we want to investigate the effectiveness of Drain+ on the microservice applications of Ant Group. We also investigate whether Drain+ can also be effective on public datasets or not.

Methods. Firstly, we run Drain+ on 10 microservice applications of Ant Group to evaluate the effectiveness of Drain+ on industrial projects; Secondly, we run Drain+ on 16 public datasets to evaluate the effectiveness of Drain+ on public datasets. Drain+ is only compared to Drain because Drain has the best average effectiveness among the six parsers on public datasets.

Results. As the results are shown in Table 9 and Table 10, we make the following observations:

(1) Table 9 shows the effectiveness of Drain+ on the log messages of the microservice applications at Ant Group. The PA row presents the parsing accuracy of the log parser. We can observe that the average PA of Drain+ is 0.846 and the average F-measure is 0.958, which are both better than Drain. Drain+ improves Drain by 218.0% in terms of average PA and by 49.9% in terms of average F-measure. We can observe that the PA of Drain on App_2, App_7, and App_{10} are close to 0. The poor PA of Drain on App_2 and App_{10} is mainly due to under-splitting, which causes Drain to partition the logs that belong to one event template into multiple event templates. The poor PA of Drain on App_7 is mainly caused by the log messages with various lengths due to nested objects. The proportion of the log messages with various lengths in App_7 is 0.747. We can also observe that the F-measure of Drain+ on App_2 isn't as good as that of Drain. We investigate the reason and find that the log messages of App_2 contain multilingual long descriptions of the product. Some separators in the descriptions lead to excessively splitting of log content. At the same time, some variables in the descriptions are identified as static text, which affects the clustering effect of candidate templates.

Table 9: The effectiveness of Drain+ on 10 microservice applications.

Dataset	Drain		Drain+	
	PA	F-measure	PA	F-measure
App_1	1.000	1.000	1.000	1.000
App_2	0.001	**0.772**	**0.590**	0.660
App_3	0.432	0.842	**0.982**	**1.000**
App_4	0.087	0.057	**0.902**	**1.000**
App_5	0.509	0.217	**0.990**	**1.000**
App_6	0.039	0.983	**0.912**	**1.000**
App_7	0.004	0.602	**0.490**	**0.990**
App_8	0.300	0.998	**1.000**	**1.000**
App_9	0.281	0.919	**0.590**	**0.930**
App_{10}	0.007	0.001	**1.000**	**1.000**
Average	0.266	0.639	**0.846**	**0.958**

Table 10: The effectiveness of Drain+ on public datasets.

Dataset	Drain		Drain+	
	PA	F-measure	PA	F-measure
HDFS	0.998	1.000	**1.000**	1.000
Hadoop	0.948	0.999	**0.954**	0.999
Spark	0.920	0.992	0.920	0.992
Zookeeper	0.967	1.000	0.967	1.000
BGL	0.941	0.999	0.941	0.999
HPC	0.887	0.991	0.887	0.991
Thunderbird	0.955	0.999	**0.969**	**1.000**
Windows	0.997	1.000	**1.000**	1.000
Linux	0.690	0.992	0.690	0.992
Andriod	0.911	**0.996**	**0.913**	0.995
HealthApp	0.780	0.918	**0.901**	**0.993**
Apache	1.000	1.000	1.000	1.000
Proxifier	0.527	0.785	**1.000**	**1.000**
OpenSSH	0.788	0.999	0.788	0.999
OpenStack	0.733	**0.993**	**0.807**	0.992
Mac	0.787	0.975	**0.856**	**0.980**
Average	0.864	0.977	**0.912**	**0.996**

(2) Table 10 shows the effectiveness of Drain+ on 16 public datasets. We can observe that Drain+ can also work well on the 16 public datasets. In general, Drain+ improves Drain by 5.6% in terms of average PA and by 1.9% in terms of average F-measure on the log messages of the 16 public datasets.

Summary for RQ3 👉 *Drain+ significantly improves Drain on microservice applications of Ant Group by 218.0% in terms of PA and by 49.9% in terms of F-measure on average. Additionally, Drain+ also outperforms Drain on public datasets on average.*

6.3 RQ4: How do the innovative components contribute to Drain+?

Motivation. We propose an improved log parser Drain+ that includes two innovative components (statistical-based separators generation and candidate event templates merging), based on Drain. The two innovative components can deal with the challenges described in Section 4.2. In this section, we conduct an ablation experiment to investigate how the two innovative components contribute to Drain+.

Methods. To investigate how the two innovative components (statistical-based separators generation and candidate event templates merging) contribute to Drain+, we compare Drain+ with two of its incomplete variants: 1) Drain + automatic separator generation component (for short, Drain_Auto_sep); 2) Drain + candidate event templates merging component (for short, Drain_Merging). Comparing Drain with Drain_Auto_sep, we can know the contribution of the statistical-based separator generation component. Comparing Drain with Drain_Merging, we can know the contribution of the candidate event templates merging component.

Results. As the results are shown in Table 11, we make the following observations:

(1) The second row presents the average effectiveness of Drain_Auto_sep on 10 microservice applications. We can observe that the average effectiveness of Drain_Auto_sep is better than that of Drain. Thus, we conclude that the statistical-based separators component can improve the parsing effectiveness.

(2) The third row presents the average effectiveness of Drain_Merging on 10 microservice applications. We can observe that the average effectiveness of Drain_Merging is better than that of Drain. Thus, we conclude that the candidate event templates merging component can improve the parsing effectiveness.

Table 11: The results of ablation experiment.

Drain	+ Auto_sep	+ Merging	PA	F-measure
✓			0.266	0.639
✓	✓		0.560	0.857
✓		✓	0.415	0.836
✓	✓	✓	0.846	0.958

The last row indicates Drain+.

Summary for RQ4 👉 *Both two innovative components (statistical-based separators generation and candidate event templates merging) make important contributions to Drain+.*

7 DISCUSSION

7.1 The effects of different parameters

There are two important parameters in Drain+: the separators' occurrence threshold in the statistical-based separators generation component and the similarity threshold in the candidate event templates merging component. We set the default separators' occurrence threshold to 0.2. We explore the effect of the separators' occurrence threshold from 0.1 to 1 (step = 0.1). We find that if the separators' occurrence threshold is small than 0.2, one token is split into multiple tokens. If the separators' occurrence threshold is greater than 0.4, that can result in multiple tokens not being split. We set the default similarity threshold to 0.6. We explore the effect of the similarity threshold from 0.1 to 1 (step = 0.1). We find that if the similarity threshold is small than 0.5, log messages that do not belong to the same event template will be merged incorrectly on most microservice applications. If the similarity threshold is greater than 0.8, part of the log messages with various lengths will not be merged on most microservice applications. Thus, our default settings are reasonable.

7.2 Threats to validity

External validity. Threats to external validity relate to the generalizability of our results. In this work, we evaluate Drain+ on 10 microservice applications of Ant Group. Drain+ achieves an average PA 0.846 and an average F-measure 0.958. Since only one company's log messages are used in our case study, we cannot claim that Drain+ can achieve high PA and F-measure on other companies' log messages. However, although Drain+ is proposed based on the findings on the datasets of one company, it is a general-purpose method and can be easily adopted in other projects. Our evaluation results on 16 public datasets confirm this argument.

Internal validity. Threats to internal validity relate to the potential deficiency of our work. Drain+ includes two innovative components to deal with the two challenges of log parsing. However, Drain+ cannot generate the separators based on the context information of log messages. For example, Drain+ cannot determine the candidate separators in a product description are not used as separators on App_2. Nevertheless, Drain+ achieves higher PA and F-measure than the six studied log parsers on industrial application and public datasets. In future work, we will continue to study the industrial log message parsing based on our findings in this work. Drain+ can be further improved by optimizing the separator generation based on the context information of log messages.

7.3 Road ahead for log parsing

Based on our findings, we now present some observations about the road ahead for log parsing.

Evaluating log parser on current public datasets is not enough. Our experiment results show that the six state-of-the-art log parsers work significantly worse on the Microservice applications of Ant Group than on public datasets. The average parsing accuracy is less than 0.3, and the average F-measure is less than 0.65. This indicates that the six state-of-the-art log parsers are still not powerful enough for industrial applications, and evaluating log parsers on current public datasets is not enough. We suggest evaluating the log parser on both public datasets and industrial projects and expanding public dataset with the projects that use more diverse logging practices.

We need an adaptive log parser. Firstly, adapting the various separators in a more granular way. We need a log parser that can determine whether a candidate separator is a separator based on its context information. For example, in App_2, the candidate separators in the product description are not used as separators. Secondly, adapting the changes of log characteristics by incremental learning or fine-tuning. We need a log parser that can automatically learn the characteristics from log messages to deal with the changes of log characteristics rather than make assumptions based on log characteristics to build the model. For example, Drain, IPLoM, AEL, and Lenma use the characteristic that the length of log messages

when building the model. This causes their performance to degrade when the log characteristics change significantly.

8 RELATED WORK

This paper investigates the effectiveness of six state-of-the-art log parsers comprehensively on industrial applications and proposes an improved log parser named Drain+ based on Drain. Therefore, we divide our related work into two aspects: log parsing and empirical study of log parsing.

8.1 Log parsing

In recent years, log parsing has been widely studied, and many log parsers have been proposed. In general, existing log parsers can be categorized into rule-based, source-code-based, and data-driven-based log parsers.

Rule-based log parsers mainly used heuristic rules designed by developers or researchers manually to parse log messages [7] [16]. The challenges of rule-based parsing include two aspects: first, it requires substantial human effort to construct the rules and maintain the rules as evolution of logging statements [23] [46]; second, the coverage of these methods is limited.

Source-code-based log parsers mainly used static analysis techniques to generate log event templates by analyzing the log statement in the source code. Xu et al. [47] used static source analysis to extract all log printing statements from the source code and types of variables contained in the log messages. Nagappan et al. [36] presented a cost-effective automated approach to parse the log lines in code into sequences of events. Due to source code often being unavailable for log parsing and various difficulties may be encountered when using static analysis technology for parsing, source code-based parsing is less popular than data-driven parsing.

Data-driven-based log parsers mainly use data mining techniques to parse log messages. Data-driven-based log parsers can be classified into heuristic-based, clustering-based, frequent-pattern-mining-based, and others according to the technology adopted. Heuristic-based log parsers include AEL [22], IPLoM [31], and Drain [18]. This kind of log parser mainly uses the characteristics of log messages and the conditions defined by experts to group log messages, then generate event templates from groups. Heuristic-based log parsers require more expert experience. Clustering-based log parsers include LKE [14], LogSig [42], LogMine [15], SHISO [34], and Lenma [40]. This kind of log parser mainly uses various similarity measures for clustering. The similarity between log messages in the same cluster is less than the pre-defined threshold, and the similarity between log messages of different clusters is greater than the threshold. The performance of clustering-based log parsers is sensitive to the pre-defined similarity threshold. Frequent-pattern-mining-based log parsers include SLCT [43], LFA [35], LogCluster [44], and Logram [6]. The general procedures of this kind of log parser are 1) traversing log messages and constructing frequent itemsets; 2) grouping log messages based on the frequent itemsets, and log messages that are considered to belong to the same event template are grouped into a cluster; 3) generating event templates from each cluster. The performance of this kind of log parser is sensitive to the threshold of frequent itemsets. In addition to the above three kinds of log parser, there are other log parsers, such as Spell [11] uses the longest common subsequence algorithm to recognize the static text in the log messages, then generate event templates. MoLFI [33] is based on the evolutionary algorithm. UniParser [28] is based on the deep learning algorithm. In this paper, we propose an improved log parser named Drain+ based on Drain. Drain+ enhances Drain with two innovative components: the statistical-based separator generation component and the candidate template merging component.

8.2 The empirical study of log parsing

In recent years, some empirical studies have investigated the effectiveness of parsing parsers on public datasets. He. et al. [17] have conducted an evaluation study of four representative log parsers (SLCT, IPLoM, LKE, and LogSig) on five datasets. They package the four log parsers into a toolkit and open source it. Zhu et al. [49] have made a comprehensive evaluation study of 13 log parsers on 16 public datasets. Our work contains an empirical study of automatic log parsing but differs from existing studies several aspects. Firstly, we comprehensively evaluate the six state-of-the-art log parsers on 10 industrial microservice applications. Secondly, we study the characteristics of the log messages of the 10 microservice applications. The results highlight two challenges (various separators and various lengths due to nested objects) of log parsing in practice. In addition, we propose an improved log parser named Drain+ based on the findings of our empirical study.

9 CONCLUSION

In this work, firstly, we comprehensively study the effectiveness of six state-of-the-art log parsers (AEL, IPLoM, Lenma, Spell, Drain, and Logram) on 10 microservice applications of Ant Group. The evaluation results show that the state-of-the-art log parsers perform significantly worse on the 10 microservice applications than on public datasets. Secondly, we investigate why the state-of-the-art log parsers perform significantly worse on the 10 microservice applications. Our empirical results highlight two challenges (various separators and various lengths due to nested objects) for log parsing in practice. Thirdly, we propose an improved log parser named Drain+ that includes two innovative components to deal with the two challenges of log parsing. We conduct an extensive experiment on 10 microservice applications of Ant Group and 16 public datasets to evaluate the effectiveness of Drain+. The results show that Drain+ outperforms the state-of-the-art log parsers on industrial applications and public datasets. Fourthly, we conduct an ablation study to investigate the contribution of two innovative components of Drain+. The results show that both innovative components make important contributions to Drain+. Finally, we conclude the observations in the road ahead for log parsing to inspire other researchers and practitioners.

Acknowledgment. This work was supported in part by the National Key Research and Development Project (No. 2018YFB2101200), the Fundamental Research Funds for the Central Universities (No. 2022CDJKYJH001), the Natural Science Foundation of Chongqing (No. cstc2021jcyj-msxmX0538), the Postdoc Foundation of Chongqing (No. 2020LY13), and the research fund from Ant Group. Zhongxin Liu gratefully acknowledges the support of Zhejiang University Education Foundation Qizhen Scholar Foundation.

REFERENCES

[1] Anton Babenko, Leonardo Mariani, and Fabrizio Pastore. 2009. Ava: Automated interpretation of dynamically detected anomalies. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. 237–248.

[2] Eduardo Berrocal, Li Yu, Sean Wallace, Michael E Papka, and Zhiling Lan. 2014. Exploring void search for fault detection on extreme scale systems. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–9.

[3] Jakub Breier and Jana Branišová. 2015. Anomaly detection from log files using data mining techniques. In *Information Science and Applications*. Springer, 449–457.

[4] An Ran Chen. 2019. An empirical study on leveraging logs for debugging production failures. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-C)*. IEEE, 126–128.

[5] Mike Chen, Alice X Zheng, Jim Lloyd, Michael I Jordan, and Eric Brewer. 2004. Failure diagnosis using decision trees. In *International Conference on Autonomic Computing, 2004. Proceedings.* IEEE, 36–43.

[6] Hetong Dai, Heng Li, Che Shao Chen, Weiyi Shang, and Tse-Hsun Chen. 2020. Logram: Efficient log parsing using n-gram dictionaries. *IEEE Transactions on Software Engineering* (2020).

[7] Carlos Viegas Damasio, Peter Fröhlich, Wolfgang Nejdl, Luis Moniz Pereira, and Michael Schroeder. 2002. Using extended logic programming for alarm-correlation in cellular phone networks. *Applied Intelligence* 17, 2 (2002), 187–202.

[8] Yingnong Dang, Qingwei Lin, and Peng Huang. 2019. AIOps: real-world challenges and research innovations. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-C)*. IEEE, 4–5.

[9] Anwesha Das, Frank Mueller, and Barry Rountree. 2020. Aarohi: Making real-time node failure prediction feasible. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1092–1101.

[10] Anwesha Das, Frank Mueller, Charles Siegel, and Abhinav Vishnu. 2018. Desh: deep learning for system health prediction of lead times to failure in hpc. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 40–51.

[11] Min Du and Feifei Li. 2018. Spell: Online streaming parsing of large unstructured system logs. *IEEE Transactions on Knowledge and Data Engineering* 31, 11 (2018), 2213–2227.

[12] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1285–1298.

[13] Nosayba El-Sayed, Hongyu Zhu, and Bianca Schroeder. 2017. Learning from failure across multiple clusters: A trace-driven approach to understanding, predicting, and mitigating job terminations. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1333–1344.

[14] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 ninth IEEE international conferenAbstracting log lines to log event types for mining software system logsce on data mining*. IEEE, 149–158.

[15] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. 2016. Logmine: Fast pattern recognition for log analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 1573–1582.

[16] Stephen E Hansen and E Todd Atkins. 1993. Automated System Monitoring and Notification with Swatch.. In *LISA*, Vol. 93. 145–152.

[17] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. 2016. An evaluation study on log parsing and its use in log mining. In *2016 46th IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 654–661.

[18] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 33–40.

[19] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. 2018. Identifying impactful service system problems via log analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 60–70.

[20] Peng Huang, Chuanxiong Guo, Jacob R Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and enhancing in situ system observability for failure detection. In *13th {USENIX} Symposium on Operating Systems Design and Implementation (OSDI)*. 1–16.

[21] Tong Jia, Pengfei Chen, Lin Yang, Ying Li, Fanjing Meng, and Jingmin Xu. 2017. An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services. In *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 25–32.

[22] Zhen Ming Jiang, Ahmed E Hassan, Parminder Flora, and Gilbert Hamann. 2008. Abstracting execution logs to execution events for enterprise applications. In *2008 The Eighth International Conference on Quality Software*. IEEE, 181–186.

[23] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, Mark D Syer, and Ahmed E Hassan. 2018. Examining the stability of logging statements. *Empirical Software Engineering* 23, 1 (2018), 290–333.

[24] Van-Hoang Le and Hongyu Zhang. 2021. Log-based anomaly detection without log parsing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 492–504.

[25] Qingwei Lin, Ken Hsieh, Yingnong Dang, Hongyu Zhang, Kaixin Sui, Yong Xu, Jian-Guang Lou, Chenggang Li, Youjiang Wu, Randolph Yao, et al. 2018. Predicting node failure in cloud service systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 480–490.

[26] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log clustering based problem identification for online service systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 102–111.

[27] Fucheng Liu, Yu Wen, Dongxue Zhang, Xihe Jiang, Xinyu Xing, and Dan Meng. 2019. Log2vec: a heterogeneous graph embedding based approach for detecting cyber threats within enterprise. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1777–1794.

[28] Yudong Liu, Xu Zhang, Shilin He, Hongyu Zhang, Liqun Li, Yu Kang, Yong Xu, Minghua Ma, Qingwei Lin, Yingnong Dang, et al. 2022. UniParser: A Unified Log Parser for Heterogeneous Log Data. In *Proceedings of the ACM Web Conference 2022*. 1893–1901.

[29] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. 2010. Mining Invariants from Console Logs for System Problem Detection.. In *USENIX Annual Technical Conference*. 1–14.

[30] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. 2018. Cloudraid: hunting concurrency bugs in the cloud via log-mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 3–14.

[31] Adetokunbo Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. 2011. A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering* 24, 11 (2011), 1921–1936.

[32] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. 2019. LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs.. In *IJCAI*, Vol. 7. 4739–4745.

[33] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. 2018. A search-based approach for accurate identification of log message formats. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 167–16710.

[34] Masayoshi Mizutani. 2013. Incremental mining of system log format. In *2013 IEEE International Conference on Services Computing*. IEEE, 595–602.

[35] Meiyappan Nagappan and Mladen A Vouk. 2010. Abstracting log lines to log event types for mining software system logs. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 114–117.

[36] Meiyappan Nagappan, Kesheng Wu, and Mladen A Vouk. 2009. Efficiently extracting operational profiles from execution logs using suffix arrays. In *2009 20th International Symposium on Software Reliability Engineering*. IEEE, 41–50.

[37] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi B Dasgupta, and Subhrajit Bhattacharya. 2016. Anomaly detection using program control flow graph mining from execution logs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 215–224.

[38] R. Real and J. M. Vargas. 1996. The Probabilistic Basis of Jaccard's Index of Similarity. *Systematic Biology* 45, 3 (1996), 380–385.

[39] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. 2008. *Introduction to information retrieval*. Vol. 39. Cambridge University Press Cambridge.

[40] Keiichi Shima. 2016. Length matters: Clustering system log messages using length of words. *arXiv preprint arXiv:1611.03213* (2016).

[41] sklearn. 2022. https://scikit-learn.org/stable/.

[42] Liang Tang, Tao Li, and Chang-Shing Perng. 2011. LogSig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*. 785–794.

[43] Risto Vaarandi. 2003. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM)*. IEEE, 119–126.

[44] Risto Vaarandi and Mauno Pihelgas. 2015. Logcluster-a data clustering and pattern mining algorithm for event logs. In *2015 11th International conference on network and service management (CNSM)*. IEEE, 1–7.

[45] Bin Xia, Yuxuan Bai, Junjie Yin, Yun Li, and Jian Xu. 2020. LogGAN: a Log-level Generative Adversarial Network for Anomaly Detection using Permutation Event Modeling. *Information Systems Frontiers* (2020), 1–14.

[46] Wei Xu. 2010. *System problem detection by mining console logs*. Ph. D. Dissertation. UC Berkeley.

[47] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 117–132.

[48] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. 2019. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM*

Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 807–817.

[49] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. 2019. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 121–130.