Logram: Efficient Log Parsing Using n-Gram Dictionaries

Hetong Dai, *Student Member, IEEE,* Heng Li, *Member, IEEE,* Che-Shao Chen, *Student Member, IEEE,* Weiyi Shang, *Member, IEEE,* Tse-Hsun (Peter) Chen, *Member, IEEE,*

Abstract—Software systems usually record important runtime information in their logs. Logs help practitioners understand system runtime behaviors and diagnose field failures. As logs are usually very large in size, automated log analysis is needed to assist practitioners in their software operation and maintenance efforts. Typically, the first step of automated log analysis is log parsing, i.e., converting unstructured raw logs into structured data. However, log parsing is challenging, because logs are produced by static templates in the source code (i.e., logging statements) yet the templates are usually inaccessible when parsing logs. Prior work proposed automated log parsing approaches that have achieved high accuracy. However, as the volume of logs grows rapidly in the era of cloud computing, efficiency becomes a major concern in log parsing. In this work, we propose an automated log parsing approach, *Logram*, which leverages n-gram dictionaries to achieve efficient log parsing. We evaluated *Logram* on 16 public log datasets and compared *Logram* with five state-of-the-art log parsing approaches. We found that *Logram* achieves a higher parsing accuracy than the best existing approaches (i.e., at least 10% higher, on average) and also outperforms these approaches in efficiency (i.e., 1.8 to 5.1 times faster than the second-fastest approaches in terms of end-to-end parsing time). Furthermore, we deployed *Logram* on *Spark* and we found that *Logram* scales out efficiently with the number of *Spark* nodes (e.g., with near-linear scalability for some logs) without sacrificing parsing accuracy. In addition, we demonstrated that *Logram* can support effective online parsing of logs, achieving similar parsing results and efficiency to the offline mode.

Index Terms—Log parsing, Log analysis, N-gram

1 Introduction

Modern software systems usually record valuable runtime information (e.g., important events and variable values) in logs. Logs play an important role for practitioners to understand the runtime behaviors of software systems and to diagnose system failures [1], [2]. However, since logs are often very large in size (e.g., tens or hundreds of gigabytes) [3], [4], prior research has proposed automated approaches to analyze logs. These automated approaches help practitioners with various software maintenance and operation activities, such as anomaly detection [5], [6], [7], [8], [9], failure diagnosis [10], [11], performance diagnosis and improvement [12], [13], and system comprehension [10], [14]. Recently, the fast-emerging AIOps (Artificial Intelligence for IT Operations) solutions also depend heavily on automated analysis of operation logs [15], [16], [17], [18], [19].

Logs are generated by logging statements in the source code. As shown in Figure 1, a logging statement is composed of log level (i.e., *info*), static text (i.e., "*Found block*" and "*locally*"), and dynamic variables (i.e., "*$blockId*"). During system runtime, the logging statement would generate raw log messages, which is a line of unstructured text that contains the static text and the values for the dynamic variables (e.g., "*rdd_42_20*") that are specified in the logging statement. The log message also contains information such as the timestamp (e.g., "*17/06/09 20:11:11*") of when the event happened. In other words, logging statements define the templates for the log messages that are generated at runtime. Automated log analysis usually has difficulties analyzing and processing the unstructured logs due to their dynamic nature [5], [10]. Instead, a log parsing step is needed to convert the unstructured logs into a structured format before the analysis. The goal of log parsing is to extract the static template, dynamic variables, and the header information (i.e., timestamp, log level, and logger name) from a raw log message to a structured format. Such structured information is then used as input for automated log analysis. He et al. [20] found that the results of log parsing are critical to the success of log analysis tasks.

Logging statement	logInfo("Found block *$blockId* locally") (From: spark/storage/BlockManager.scala)
Raw log (Unstructured)	17/06/09 20:11:11 INFO storage.BlockManager: Found block *rdd_42_20* locally
Parsed log (Structured)	**Timestamp**: 17/06/09 20:11:11; **Level**: INFO **Logger**: storage.BlockManager **Static template**: Found block <*> locally **Dynamic variable(s)**: *rdd_42_20*

Fig. 1. An illustrative example of parsing an unstructured log message into a structured format.

In practice, practitioners usually write *ad hoc* log parsing scripts that depend heavily on specially-designed regular expressions [21], [22], [23] . As modern software systems

- *Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada.*
  E-mail: (he_da, c_chesha, shang, peterc)@encs.concordia.ca
- *School of Computing, Queen's University, Kingston, Canada.*
  E-mail: hengli@cs.queensu.ca

usually contain large numbers of log templates which are constantly evolving [24], [25], [26], practitioners need to invest a significant amount of efforts to develop and maintain such regular expressions. In order to ease the pain of developing and maintaining *ad hoc* log parsing scripts, prior work proposed various approaches for automated log parsing [21]. For example, *Drain* [22] uses a fixed-depth tree to parse logs. Each layer of the tree defines a rule for grouping log messages (e.g., log message length, preceding tokens, and token similarity). At the end, log messages with the same templates are clustered into the same groups. Zhu et al. [21] proposed a benchmark and thoroughly compared prior approaches for automated log parsing.

Despite the existence of prior log parsers, as the size of logs grows rapidly [1], [2], [27] and the need for low-latency log analysis increases [19], [28], efficiency becomes an important concern for log parsing. In this work, we propose *Logram*, an automated log parsing approach that leverages n-gram dictionaries to achieve efficient log parsing. In short, *Logram* uses dictionaries to store the frequencies of n-grams in logs and leverage the n-gram dictionaries to extract the static templates and dynamic variables in logs. Our intuition is that frequent n-grams are more likely to represent the *static templates* while rare n-grams are more likely to be *dynamic variables*. The n-gram dictionaries can be constructed and queried efficiently, i.e., with a complexity of $O(n)$ and $O(1)$, respectively.

We evaluated *Logram* on 16 log datasets [21] and compared *Logram* with five state-of-the-art log parsing approaches. We found that *Logram* achieves a higher accuracy compared with the best existing approaches (i.e., at least 10% higher on average), and that *Logram* outperforms these best existing approaches in efficiency, achieving a parsing speed that is 1.8 to 5.1 times faster than the second-fastest approaches. Furthermore, as the n-gram dictionaries can be constructed in parallel and aggregated efficiently, we demonstrated that *Logram* can achieve high scalability when deployed on a multi-core environment (e.g., a *Spark* cluster), without sacrificing any parsing accuracy. Finally, we demonstrated that *Logram* can support effective online parsing, i.e., by updating the n-gram dictionaries continuously when new logs are added in a streaming manner.

In summary, the main contributions[1] of our work include:

- We present the detailed design of an innovative approach, *Logram*, for automated log parsing. *Logram* leverages n-gram dictionaries to achieve accurate and efficient log parsing.
- We compare the performance of *Logram* with other state-of-the-art log parsing approaches, based on an evaluation on 16 log datasets. The results show that *Logram* outperforms other state-of-the-art approaches in efficiency and achieves better accuracy than existing approaches.
- We deployed *Logram* on *Spark* and we show that *Logram* scales out efficiently as the number of *Spark* nodes increases (e.g., with near-linear scalability for some logs), without sacrificing paring accuracy.
- We demonstrate that *Logram* can effectively support online parsing, achieving similar parsing results and efficiency compared to the offline mode.

Our highly accurate, highly efficient, and highly scalable *Logram* can benefit future research and practices that rely on automated log parsing for log analysis on large log data. In addition, practitioners can leverage *Logram* in a log streaming environment to enable effective online log parsing for real-time log analysis.

Paper organization. The paper is organized as follows. Section 2 introduces the background of log parsing and n-grams. Section 3 surveys prior work related to log parsing. Section 4 presents a detailed description of our *Logram* approach. Section 5 shows the results of evaluating *Logram* on 16 log datasets. Section 6 discusses the effectiveness of *Logram* for online log parsing. Section 7 discusses the threats to the validity of our findings. Finally, Section 8 concludes the paper.

2 Background

In this section, we introduce the background of log parsing and n-grams that are used in our log parsing approach.

2.1 Log Parsing

In general, the goal of log parsing is to extract the static template, dynamic variables, and the header information (i.e., timestamp, level, and logger) from a raw log message. While the header information usually follows a fixed format that is easy to parse, extracting the templates and the dynamic variables is much more challenging, because 1) the static templates (i.e., logging statements) that generate logs are usually inaccessible [21], and 2) logs usually contain a large vocabulary of words [23]. Table 1 shows four simplified log messages with their header information removed. These four log messages are actually produced from two static templates (i.e., "*Found block <*> locally*" and "*Dropping block <*> from memory*"). These log messages also contain dynamic variables (i.e., "*rdd_42_20*" and "*rdd_42_22*") that vary across different log messages produced by the same template. Log parsing aims to separate the static templates and the dynamic variables from such log messages.

Traditionally, practitioners rely on *ad hoc* regular expressions to parse the logs that they are interested in. For example, two regular expressions (e.g., "*Found block [a-z0-9_]+ locally*" and "*Dropping block [a-z0-9_]+ from memory*") could be used to parse the log messages shown in Table 1. Log processing & management tools (e.g., *Splunk*[2] and *ELK* stack[3]) also enable users to define their own regular expressions to parse log data. However, modern software systems usually contain large numbers (e.g., tens of thousands) of log templates which are constantly evolving [21], [24], [25], [26], [29]. Thus, practitioners need to invest a significant amount of efforts to develop and maintain such *ad hoc* regular expressions. Therefore, recent work perform literature reviews and study various approaches to automate the log parsing process [21], [30]. In this work, we propose an automated log parsing approach that is highly accurate, highly efficient, highly scalable, and supports online parsing.

1. The source code of our tool and the data used in our study are shared at https://github.com/BlueLionLogram/Logram

2. https://www.splunk.com
3. https://www.elastic.co

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3007554, IEEE Transactions on Software Engineering

TABLE 1
Simplified log messages for illustration purposes.

1.	Found block *rdd_42_20* locally
2.	Found block *rdd_42_22* locally
3.	Dropping block *rdd_42_20* from memory
4.	Dropping block *rdd_42_22* from memory

2.2 n-grams

An n-gram is a subsequence of length n from an item sequence (e.g., text [31], speech [32], source code [33], or genome sequences [34]). Taking the word sequence in the sentence: "*The cow jumps over the moon*" as an example, there are five 2-grams (i.e., bigrams): "*The cow*", "*cow jumps*", "*jumps over*", "*over the*", and "*the moon*", and four 3-grams (i.e., trigrams): "*The cow jumps*", "*cow jumps over*", "*jumps over the*", and "*over the moon*". n-grams have been successfully used to model natural language [31], [35], [36], [37] and source code [38], [39], [40]. However, there exists no work that leverages n-grams to model log data. In this work, we propose *Logram* that leverages n-grams to parse log data in an efficient manner. Our intuition is that frequent n-grams are more likely to be static text while rare n-grams are more likely to be dynamic variables.

Logram extracts n-grams from the log data and store the frequencies of each n-gram in dictionaries (i.e., n-gram dictionaries). Finding all the n-grams in a sequence (for a limited n value) can be achieved efficiently by a single pass of the sequence (i.e., with a linear complexity) [41]. For example, to get the 2-grams and 3-grams in the sentence "*The cow jumps over the moon*", an algorithm can move one word forward each time and get a 2-gram and a 3-gram starting from that word each time. Besides, the nature of the n-gram dictionaries enables one to construct the dictionaries in parallel (e.g., by building separate dictionaries for different parts of logs in parallel and then aggregating the dictionaries). Furthermore, the n-gram dictionaries can be updated online when more logs are added (e.g., in log streaming scenarios). As a result, as shown in our experimental results, *Logram* is highly efficient, highly scalable, and supports online parsing.

3 Related Work

In this section, we discuss prior work that proposed log parsing techniques and prior work that leveraged log parsing techniques in various software engineering tasks (e.g., anomaly detection).

3.1 Prior Work on Log Parsing

In general, existing log parsing approaches could be grouped under three categories: *rule-based*, *source code-based,* and *data mining-based* approaches.

Rule-based log parsing. Traditionally, practitioners and researchers hand-craft heuristic rules (e.g., in the forms of regular expressions) to parse log data [42], [43], [44]. Modern log processing & management tools usually provide support for users to specify customized rules to parse their log data [45], [46], [47]. Rule-based approaches require substantial human effort to design the rules and maintain the rules as log formats evolve [24]. Using standardized logging formats [48], [49], [50] can ease the pain of manually designing log parsing rules. However, such standardized log formats have never been widely used in practice [23].

Source code-based log parsing. A log event is uniquely associated with a logging statement in the source code (see Section 2.1). Thus, prior studies proposed automated log parsing approaches that rely on the logging statements in the source code to derive log templates [5], [51]. Such approaches first use static program analysis to extract the log templates (i.e., from logging statements) in the source code. Based on the log templates, these approaches automatically compose regular expressions to match log messages that are associated with each of the extracted log templates. Following studies [52], [53] applied [5] on production logs (e.g., Google's production logs) and achieved a very high accuracy. However, source code is often not available for log parsing tasks, for example, when the log messages are produced by closed-source software or third-party libraries; not to mention the efforts for performing static analysis to extract log templates using different logging libraries or different programming languages.

Data mining-based log parsing. Other automated log parsing approaches do not require the source code, but instead, leverage various data mining techniques. *SLCT* [54], *LogCluster* [55], and *LFA* [55] proposed approaches that automatically parse log messages by mining the frequent tokens in the log messages. These approaches count token frequencies and use a predefined threshold to identify the static components of log messages. The intuition is that if a log event occurs many times, then the static components will occur many times, whereas the unique values of the dynamic components will occur fewer times. Prior work also formulated log parsing as a clustering problem and used various approaches to measure the similarity/distance between two log messages (e.g., *LKE* [8], *LogSig* [56], *LogMine* [57], *SHISO* [58], and *LenMa* [59]). For example, *LKE* [8] clusters log messages into event groups based on the edit distance, weighted by token positions, between each pair of log messages.

AEL [23] used heuristics based on domain knowledge to recognize dynamic components (e.g., tokens following the "=" symbol) in log messages, then group log messages into the same event group if they have the same static and dynamic components. *Spell* [60] parses log messages based on a longest common subsequence algorithm, built on the observation that the longest common subsequence of two log messages are likely to be the static components. *IPLoM* [61] iteratively partitions log messages into finer groups, firstly by the number of tokens, then by the position of tokens, and lastly by the association between token pairs. *Drain* [22] uses a fixed-depth tree to represent the hierarchical relationship between log messages. Each layer of the tree defines a rule for grouping log messages (e.g., log message length, preceding tokens, and token similarity). Zhu et al. [21] evaluated the performance of such data mining-based parsers and they found that *Drain* [22] achieved the best performance in terms of accuracy and efficiency. Our n-gram-based log parser achieves a much faster parsing speed and a better parsing accuracy compared to *Drain*.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3007554, IEEE Transactions on Software Engineering

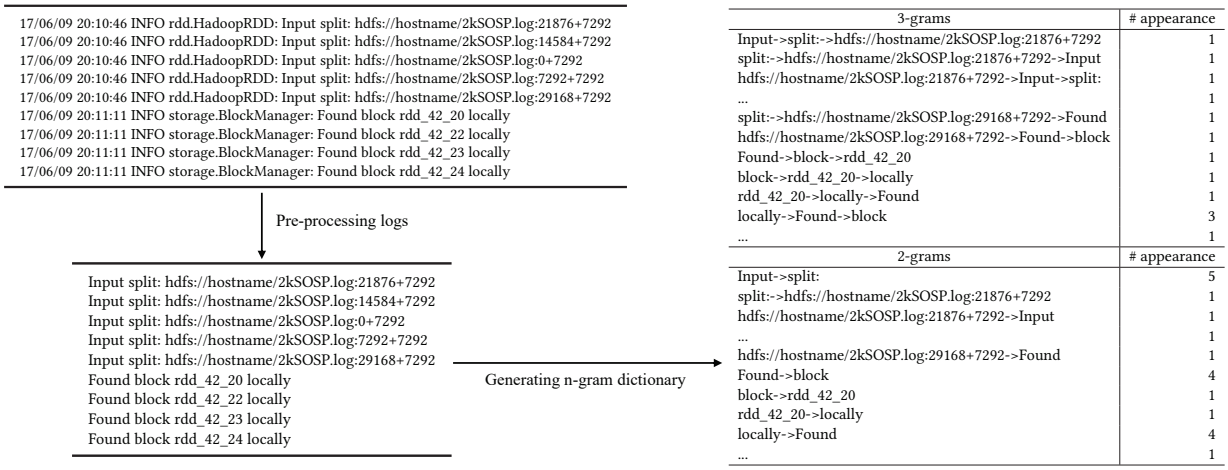


Fig. 2. A running example of generating n-gram dictionary, which will be later used for parsing each log messages as shown in Figure 3.

In addition, prior study performed a systematic literature review on automated log parsing techniques [30]. They investigated the advantages and limitations of 17 log parsing techniques in terms of seven aspects, such as efficiency and required parameter tuning effort, which can help practitioners choose the right log parsers for their specific scenarios and provide insights for future research directions.

3.2 Applications of Log Parsing

Log parsing is usually a prerequisite for various log analysis tasks, such as anomaly detection [5], [6], [7], [8], [9], failure diagnosis [10], [11], performance diagnosis and improvement [12], [13], and system comprehension [10], [14]. For example, Fu et al. [8] first parse the raw log messages to extract log events. Based on the extracted event sequences, they then learn a Finite State Automaton (FSA) to represent the normal work flow, which is in turn used to detect anomalies in new log files. Prior work [20] shows that the accuracy of log parsing is critical to the success of log analysis tasks. Besides, as the size of log files grows fast [1], [2], [27], a highly efficient log parser is important to ensure that the log analysis tasks can be performed in a timely manner. In this work, we propose a log parsing approach that is not only accurate but also efficient, which can benefit future log analysis research and practices.

4 Approach

In this section, we present our automated log parsing approach that is designed using n-gram dictionaries.

4.1 Overview of *Logram*

Our approach consists of two steps: 1) generating n-gram dictionaries and 2) parsing log messages using n-gram dictionaries. In particular, the first step generates n-grams from log messages and calculate the number of appearances of each n-gram. In the second step, each log message is transformed into n-grams. By checking the number of appearance of each n-gram, we can automatically parse the log message into static text and dynamic variables. Figure 2 and 3 show the overview of our approach with a running example.

4.2 Generating an n-gram dictionary

4.2.1 Pre-processing logs

In this step, we extract a list of tokens (i.e., separated words) from each log message. First of all, we extract the content of a log message by using a pre-defined regular expression. For example, a log message often starts with the time stamp , the log level, and the logger name. Since these parts of logs often follow a common format in the same software system (specified by the configuration of logging libraries), we can directly parse and obtain these information. For example, a log message from the running example in Figure 2, i.e., "*17/06/09 20:11:11 INFO storage.BlockManager: Found block rdd_42_24 locally*", "*17/06/09 20:11:11*" is automatically identified as time stamp, "*INFO*" is identified as the log level and "*Storage.BlockManager:*" is identified as the logger name; while the content of the log is "*Found block rdd_42_24 locally*". After getting the content of each log message, we split the log message into tokens. The log message is split with white-space characters (e.g., *space* and *tab*). Finally, there exist common formats for some special dynamic information in logs, such as IP address and email address.

In order to have a unbiased comparison with other existing log parsers in the *LogPai* benchmark (cf. Section 5), we leverage the openly defined regular expressions that are available from the *LogPai* benchmark to identify such dynamic information.

4.2.2 Generating an n-gram dictionary

We use the log tokens extracted from each log message to create an n-gram dictionary. Naively, for a log message with m tokens, one may create an n-gram where $n \leq m$. However, when m has the same value as n, the phrases with n-grams are exactly all log messages. Such a dictionary is not useful since almost all log messages have tokens that are generated from dynamic variables. On the other hand, a small value of n may increase the chance that the text generated by a dynamic variable has multiple appearances. A prior study [62] finds that the repetitiveness of an n-gram in logs starts to become stable when $n \leq 3$. Therefore, in our approach, we generate the dictionary using

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3007554, IEEE Transactions on Software Engineering

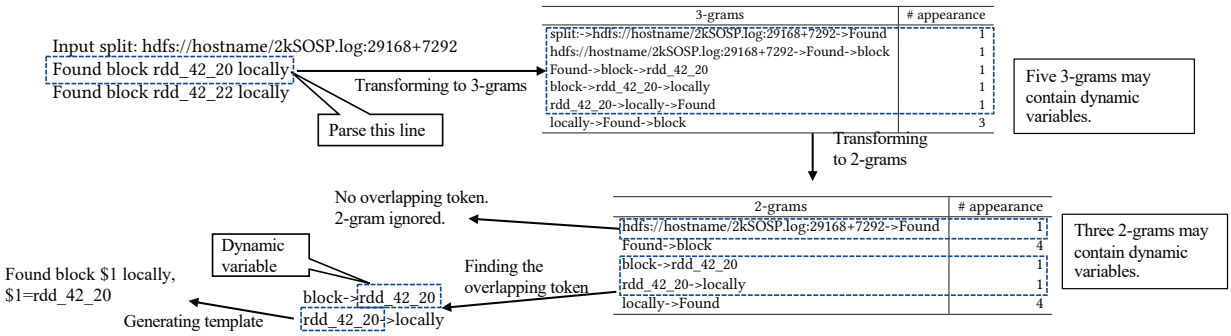


Fig. 3. A running example of parsing one log message using the dictionary built from Figure 2.

phrases with two or three words (i.e., 2-grams and 3-grams). Naively, one may generate the dictionary by processing every single log message independently. However, such a naive approach has two limitations: 1) some log events may span across multiple lines and 2) the beginning and the ending tokens of a log message may not reside in the same number of n-grams like other tokens (c.f. our parsing step "Identifying dynamically and statically generated tokens" in Section 4.3.2). For example, the first token of each log message cannot be put in a 2-gram nor 3-gram and the second token of each log line cannot be put in a 3-gram. This limitation may lead to potential bias of the tokens being considered as dynamic variables. In order to ensure that all the tokens are considered equally when generating the n-gram dictionary for *Logram*, at the beginning and the ending tokens of a log message, we also include the end of the prior log message and the beginning of the following log message, respectively, to create n-grams. For example, if our highest n in the n-gram is 3, we would check two more tokens at the end of the prior log message and the beginning of the following log message. In addition, we calculate the number of occurrences of each n-gram in our dictionary.

As shown in a running example in Figure 2, a dictionary from the nine lines of logs is generated consisting of 3-grams and 2-grams. Only one 3-grams, "*locally->Found->block*", in the example have multiple appearance. Three 2-grams, "*Found->block*", "*Input->split:*" and "*locally->Found*", have four to five appearances. In particular, there exists n-grams, such as the 3-gram "*locally->Found->block*", that are generated by combining the end and beginning of two log messages. Without such combination, tokens like "*input*", "*Found*" and "*locally*" will have lower appearance in the dictionary.

4.3 Parsing log messages using an n-gram dictionary

In this step of our approach, we parse log messages using the dictionary that is generated from the last step.

4.3.1 Identifying n-grams that may contain dynamic variables

Similar to the last step, each log message is transformed into n-grams. For each n-gram from the log message, we check its number of appearances in the dictionary. If the number of occurrence of a n-gram is smaller than a automatically determined threshold (see Section 4.3.3), we consider that the n-gram may contain a token that is generated from dynamic variables. In order to scope down to identify the dynamically generated tokens, after collecting all low-appearing n-grams, we transform each of these n-grams into $n-1$-grams, and check the number of appearance of each $n-1$-gram. We recursively apply this step until we have a list of low-appearing 2-grams, where each of them may contain one or two tokens generated from dynamic variables. For our running example shown in Figure 3, we first transform the log message into two 3-grams, while both only have one appearance in the dictionary. Hence, both 3-grams from the running example in Figure 3 may contain dynamic variables. Afterwards, we transform the 3-grams into three 2-grams. One of the 2-grams ("*Found->block*") has four appearances; while the other two 2-grams ("*block->rdd_42_20*" and "*rdd_42_20->locally*") only have one appearance. Therefore, we keep the two 2-grams to identify the dynamic variables.

4.3.2 Identifying dynamically and statically generated tokens

From the last step, we obtain a list of low-appearing 2-grams. However, not all tokens in the 2-grams are dynamic variables. There may be 2-grams that have only one dynamically generated token while the other token is static text. In such cases, the token from the dynamic variable must reside in two consecutive low-appearing 2-grams (i.e., one ends with the dynamic variable and one starts with the dynamic variable). For all other tokens, including the ones that are now selected in either this step or the last step, we consider them as generated from static text.

However, a special case of this step is the beginning and ending tokens of each log message (c.f. the previous step "Generating an n-gram dictionary" in Section 4.2.2). Each of these tokens would only appear in smaller number of n-grams. For example, the first token of a log message would only appear in one 2-gram. If these tokens of the log message are from static text, they may be under-counted to be considered as potential dynamic variables. If these tokens of the log message are dynamically generated, they would never appear in two 2-grams to be identified as dynamic variable. To address this issue, for the beginning and ending tokens of each log message, we generate additional n-grams by considering the ending tokens from the prior log message; and for the ending tokens of each log message, we generate additional n-grams by considering the beginning tokens from the next log message.

For our running example shown in Figure 3, "*rdd_42_20*" is generated from dynamic variable and it reside in two 2-grams ("*block->rdd_42_20*" and "*rdd_42_20->locally*". Therefore, we can identify token "*rdd_42_20*" as a dynamic variable, while "*block*" and "*locally*" are static text. On the other hand, since "*hdfs://hostname/2kSOSP.log :29168+7292->Found*" only appear without overlapping tokens with others, we ignore this 2-gram for identifying dynamic variables.

Another special case happens when a static token appears in between two dynamic tokens. For example, if we have three tokens that form a 3-gram *dynamic->static->dynamic*, when transforming the 3-gram, we end up with two 2-grams: *dynamic->static* and *static->dynamic*. Both 2-grams would be low-appearing 2-grams as both of them include a dynamic token. As the token *static* appears in two low-appearing 2-grams, it would be falsely identified as a dynamic token based on our earlier rule. In order to cope with such special cases, when a token is surrounded by two dynamic tokens, we create a pseudo bi-gram using a wildcard (e.g., *?->static*), and use its frequency to determine whether the middle token (e.g., *static*) is static or dynamic.

4.3.3 Automatically determining the threshold of n-gram occurrences

The above identification of dynamically and statically generated tokens depends on a threshold of the occurrences of n-grams. In order to save practitioners' effort for manually searching the threshold, we use an automated approach to estimate the appropriate threshold. Our intuition is that most of the static n-grams would have more occurrences while the dynamic n-grams would have fewer occurrences. Therefore, there may exist a gap between the occurrences of the static n-grams and the occurrences of the dynamic n-grams, i.e., such a gap helps us identify a proper threshold automatically.

In particular, first, we measure the occurrences of each n-gram. Then, for each occurrence value, we calculate the number of n-grams that have the exact occurrence value. We use a two-dimensional coordinate to represent the occurrence values (i.e., the X values) and the number of n-grams that have the exact occurrence values (i.e., the Y values). Then we use the *loess* function [63] to smooth the Y values and calculate the derivative of the Y values against the X values. After getting the derivatives, we use Ckmeans.1d.dp [64], a one-dimensional clustering method, to find a break point to separate the derivatives into two groups, i.e., a group for static n-grams and a group for dynamic n-grams. The breaking point would be automatically determined as the threshold.

4.3.4 Generating log templates

Finally, we generate log templates based on the tokens that are identified as dynamically or statically generated. We follow the same log template format as the *LogPai* benchmark [21], in order to assist in further research. For our running example shown in Figure 3, our approach parses the log message "*Found block rdd_42_20 locally*" into "*Found block $1 locally, $1=rdd_42_20*".

5 Evaluation

In this section, we present the evaluation of our approach. We evaluate our approach by parsing logs from the *LogPai* benchmark [21]. We compare *Logram* with five automated log parsing approaches, including *Drain* [22], *Spell* [60], *AEL* [23], *Lenma* [59] and *IPLoM* [61] that are from prior research and all have been included in the *LogPai* benchmark. We choose these five approaches since a prior study [21] finds that these approaches have the highest accuracy and efficiency among all of the evaluated log parsers. In particular, we evaluate our approach on four aspects:

Accuracy. The accuracy of a log parser measures whether it can correctly identify the static text and dynamic variables in log messages, in order to match log messages with the correct log events. A prior study [20] demonstrates the importance of high accuracy of log parsing, and low accuracy of log parsing can cause incorrect results (such as false positives) in log analysis.

Efficiency. Large software systems often generate a large amount of logs during run time [65]. Since log parsing is typically the first step of analyzing logs, low efficiency in log parsing may introduce additional costs to practitioners when doing log analysis and cause delays to uncover important knowledge from logs.

Ease of stabilisation. Log parsers typically learn knowledge from existing logs in order to determine the static and dynamic components in a log message. The more logs seen, the better results a log parser can provide. It is desired for a log parser to have a stable result with learning knowledge from a small amount of existing logs, such that parsing the log can be done in a real-time manner without the need of updating knowledge while parsing logs.

Scalability. Due to the large amounts of log data, one may consider leveraging parallel processing frameworks, such as *Hadoop* and *Spark*, to support the parsing of logs [66]. However, if the approach of a log parser is difficult to scale, it may not be adopted in practice.

5.1 Subject log data

We use the data set from the *LogPai* benchmark [21]. The data sets and their descriptions are presented in Table 2. The benchmark includes logs produced by both open source and industrial systems from various domains. These logs are typically used as subject data for prior log analysis research, such as system anomaly detection [67], [68], system issue diagnosis [69] and system understanding [70]. To assist in automatically calculating accuracy on log parsing (c.f., Section 5.2), each data set in the benchmark includes a subset of 2,000 log messages that are already manually labeled with log event. Such manually labeled data are used in evaluating the accuracy of our log parser. For the other three aspects of the evaluation, we use the entire logs of each log data set.

5.2 Accuracy

In this subsection, we present the evaluation results on the accuracy of *Logram*.

Prior approach by Zhu et al. [21] defines an accuracy metric as the ratio of correctly parsed log messages over the total number of log messages. In order to calculate the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3007554, IEEE Transactions on Software Engineering

TABLE 2
The subject log data used in our evaluation.

Dataset	Description	Size
Android	Android framework log	183.37MB
Apache	Apache server error log	4.90MB
BGL	Blue Gene/L supercomputer log	708.76MB
Hadoop	Hadoop mapreduce job log	48.61MB
HDFS	Hadoop distributed file system log	1.47GB
HealthApp	Health app log	22.44MB
HPC	High performance cluster log	32.00MB
Linux	Linux system log	2.25MB
Mac	Mac OS log	16.09MB
OpenSSH	OpenSSH server log	70.02MB
OpenStack	OpenStack software log	58.61MB
Proxifier	Proxifier software log	2.42MB
Spark	Spark job log	2.75GB
Thunderbird	Thunderbird supercomputer log	29.60GB
Windows	Windows event log	26.09GB
Zookeeper	ZooKeeper service log	9.95MB

TABLE 3
Accuracy of *Logram* compared with other log parsers. The results that are the highest among the parsers are annotaetd with * and the results that are higher than 0.9 are highlighted in bold.

Dataset	*Drain*	*AEL*	*Lenma*	*Spell*	*IPLoM*	*Logram*
Android	**0.933**	0.867	**0.976***	**0.933**	0.716	0.848
Apache	0.693	0.693	0.693	0.693	0.693	0.699*
BGL	0.822*	0.818	0.577	0.639	0.792	0.740
Hadoop	0.545	0.539	0.535	0.192	0.373	**0.965***
HDFS	**0.999***	**0.999***	**0.998**	**0.999***	**0.998**	**0.981**
HealthApp	0.609	0.615	0.141	0.602	0.651	**0.969***
HPC	**0.929**	**0.990***	**0.915**	0.694	**0.979**	**0.959**
Linux	0.250	0.241	0.251	0.131	0.235	0.460*
Mac	0.515	0.579	0.551	0.434	0.503	0.666*
OpenSSH	0.507	0.247	0.522	0.507	0.508	0.847*
Openstack	0.538	0.718	0.759*	0.592	0.697	0.545
Proxifier	**0.973***	**0.968**	**0.955**	0.785	**0.975**	**0.951**
Spark	**0.902**	**0.965***	**0.943**	0.865	0.883	**0.903**
Thunderbird	0.803	0.782	0.814*	0.773	0.505	0.761
Windows	**0.983***	**0.983***	0.277	**0.978**	0.554	**0.957**
Zookeeper	**0.962**	**0.922**	0.842	**0.955**	**0.967***	**0.955**
Average	0.748	0.745	0.672	0.669	0.689	0.825*

parsing accuracy, a log event template is generated for each log message and log messages with the same template will be grouped together. If all the log messages that are grouped together indeed belong to the same log template, and all the log messages that indeed belong to this log template are in this group, the log messages are considered parsed correctly. However, the grouping accuracy has a limitation that it only determines whether the logs from the same events are grouped together; while the static text and dynamic variables in the logs may not be correctly identified. We discuss the limitation of using the grouping accuracy in detail in the *Results* subsection.

On the other hand, correctly identifying the static text and dynamic variables are indeed important for various log analysis. For example, Xu et al. [5] consider the variation of the dynamic variables to detect run-time anomalies. Therefore, we manually check the parsing results of each log message and determine whether the static text and dynamic variables are correctly parsed, i.e., parsing accuracy. In other words, a log message is considered correctly parsed if and only if all its static text and dynamic variables are correctly identified.

Results

***Logram* achieves an over 0.9 accuracy or the best accuracy in parsing 12 out of the 16 log datasets.** Table 3 shows the accuracy on 16 datasets. Following the prior log-parsing benchmark [21], we highlight the accuracy values that are higher than 0.9 or that are the highest for each log dataset. *Logram* achieves an average parsing accuracy of 0.825, which is at least 10% higher than the average accuracy achieved by any other log parsers. The second most accurate approach, i.e., *Drain*, achieves an average parsing accuracy of 0.748. For eight log datasets, *Logram* has an parsing accuracy higher than 0.9; for four other datasets, *Logram* has the highest parsing accuracy among all parsers. Since *Logram* is designed based on processing every token in a log instead of comparing each line of logs with others, *Logram* exceeds other approaches in terms of parsing accuracy.

Even though prior approaches may often correctly group log messages together, the static text and dynamic variables of these log messages may not be correctly identified. For example, when parsing the Hadoop logs, the parser *Spell* achieves a grouping accuracy of 0.778 [21] but the parsing accuracy is only 0.192. By manually checking the results, we find that some groups of logs share the same strings of host names that are generated from dynamic variables but are parsed as static tokens. For example, a log message "*Address change detected. Old: msra-sa-41/10.190.173.170:9000 New: msra-sa-41:9000*" is parsed into a log template "*Address change detected. Old msra-sa-41/<*> <*> New msra-sa-41 <*>*". We can see that the strings "*msra-sa*" and "*msra-sa-41*" in the host names are not correctly detected as dynamic variables. However, since all the log messages in this category have such strings in the host names, even though *Spell* cannot identify the dynamic variables, the log messages are still grouped together. In particular, we manually examined the parsing results of *Logram* for all the log lines that are considered to be correctly parsed in terms of the grouping accuracy [21]. In total, we found that 17.8% (4641 out of 26017) of the log lines that are considered to be correctly parsed in terms of the grouping accuracy are actually incorrectly parsed based on our manual examination. For each log file, 0.2% to 79.2% of the log lines that are considered correctly parsed in terms of the grouping accuracy are false positives.

Finally, we manually study the incorrectly parsed log messages by *Logram*. Such incorrectly parsed log messages can be due to false-dynamic, i.e., a static token that is mis-identified as a dynamic token, or false-static, i.e., a dynamic token that is mis-identified as a static token. For all the lines that are incorrectly parsed, 41.1% contain false-dynamic tokens and 61.8% contain false-static tokens. Through further manual investigation, we identify the following three reasons of incorrectly parsing some of the log messages:

1) **Mis-split tokens.** Some log messages contain special tokens such as $+$ and $\{$ to separate two tokens. In addition, sometimes static text and dynamic variables are printed into one single token without any separator (like a white space). It is difficult for a token-based approach to address such cases. This

reason may contribute to both false-static and false-dynamic results.

2) **Pre-processing errors.** The pre-processing of common log formats may introduce mistakes to the log parsing. For example, text in a common format (e.g., a date format) may be part of a long dynamic variable (a task id with its date as part of it). However, the pre-processing step extracts only the text in the common format, causing the rest of the text in the dynamic variable being parsed into a wrong format. This reason may contribute to false-static results.
3) **Frequently-appearing dynamic variables.** Some dynamic variables contain contextual information of system environment and can appear frequently in logs. For example, in Apache logs, the path to a property file often appears in log messages such as: "*workerEnv.init() ok /etc/httpd/conf/-workers2.properties*". Although the path is a dynamic variable, in fact, the value of the dynamic variable never changes in the logs, preventing *Logram* from identifying it as a dynamic variable. On the other hand, such an issue is not challenging to address in practice. Practitioners can consider such contextual information in the pre-processing step.

5.3 Efficiency

To measure the efficiency of a log parser, similar to prior research [21], [71], we record the elapsed time to finish the entire end-to-end parsing process on different log data with varying log sizes. We randomly extract data chunks of different sizes, i.e., 300KB, 1MB, 10MB, 100MB, 500MB and 1GB. Specifically, we choose to evaluate the efficiency from the Android, BGL, HDFS, Windows and Spark datasets, due to their proper sizes for such evaluation. From each log dataset, we randomly pick a point in the file and select a data chunk of the given size (e.g., 1MB or 10MB). We ensure that the size from the randomly picked point to the end of the file is not smaller than the given data size to be extracted. We measure the elapsed time for the log parsers on a desktop computer with Inter Core i5-2400 CPU 3.10GHz CPU, 8GB memory and 7,200rpm SATA hard drive running Ubuntu 18.04.2. We compare *Logram* with three other parsers, i.e., *Drain*, *Spell* and *AEL*, that all have high accuracy in our evaluation and more importantly, have the highest efficiency on log parsing based on the prior benchmark study [21].

In addition, we calculate the size of the dictionaries that are generated by *Logram* (cf. Section 4.2). These dictionaries are meant to be kept in memory to assist in a fast parsing of logs. The large size of the dictionaries may become a memory overhead to the efficiency of the approach.

Results

***Logram* outperforms the fastest state-of-the-art log parsers in efficiency by 1.8 to 5.1 times.** Figure 4 shows that time needed to parse five different log data with various sizes using *Logram* and five other log parsers. We note that for *Logram*, the time to construct the dictionaries is already included in our end-to-end elapsed time, in order to fairly compare log parsing approaches. The dictionary construction (including pre-processing log messages and generating n-gram dictionaries) accounts for 77% to 92% of the overall parsing time, while parsing the log messages using the constructed dictionaries only takes 8% to 23% of the overall parsing time. We find that *Logram* drastically outperform all other existing parsers. In particular, *Logram* is 1.8 to 5.1 times faster than the second fastest approaches when parsing the five log datasets along different sizes.

The efficiency of *Logram* is stable when increasing the sizes of logs. From our results, the efficiency of *Logram* is not observed to be negatively impacted when the size of logs increases. For example, the running time only increases by a factor of 773 to 1039 when we increase the sizes of logs from 1 MB to 1GB (i.e., by a factor of 1,000). *Logram* keeps a dictionary that is built from the n-grams in log messages. With larger size of logs, the size of dictionary may drastically increasing. However, our results indicate that up to the size of 1GB of the studied logs, the efficiency keeps stable. We consider the reason is that when paring a new log message using the dictionary, the look-up time for an n-gram in our dictionary is consistent, despite the size of the dictionary. Hence even with larger logs, the size of the dictionary do not drastically change.

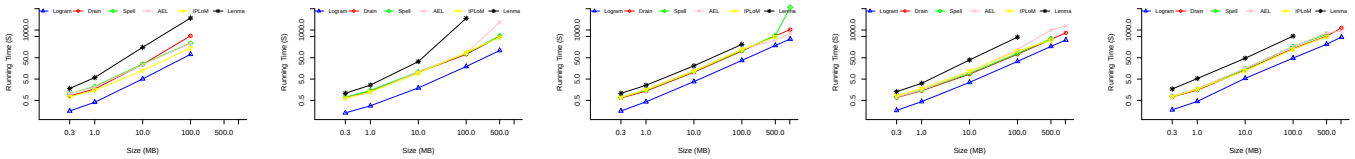
On the other hand, the efficiency of other log parsers may deteriorate with larger logs. In particular, *Lenma* has the lowest efficiency among all studied log parsers. *Lenma* cannot finish finish parsing any 500MB and 1GB log dataset within hours. In addition, *Spell* would crash on Windows and Spark log files with 1G size due to memory issues. *AEL* shows a lower efficiency when parsing large Windows and BGL logs. Finally, *Drain*, although not as efficient as *Logram*, does not have a lower efficiency when parsing larger sizes of logs, which agrees with the finding in prior research on the log parsing benchmark [21].

***Logram* does not generate large size of dictionaries as memory overhead.** Table 4 shows the sizes of both 2-gram and 3-gram dictionaries that are generated by each log dataset. We found that, a dataset with a large dictionary, e.g, BGL, has less than 300K keys in total, which only takes 11.2 MB (4.1MB for 2-grams and 7.1MB for 3-grams) of memory. The largest dataset, i.e., the Thunderbird logs, has a total of nearly 9 million keys. However, the dictionary only takes 412 MB (150 MB for 2-grams and 262 MB for 3-grams) of memory. On the other hand, we also found that a large log dataset may not have as large a dictionary. For example, the Windows logs have almost a similar size as the Thunderbird logs but the generated dictionary only has 600K keys, which is only 7% of the size of the dictionary generated from the Thunderbird logs. Our results show that the size of the dictionaries may be influenced not only by the size of the logs, but also by other characteristics of the logs (e.g., the number of unique 2-grams and 3-grams).

5.4 Ease of stabilisation

We evaluate the ease of stabilisation by running *Logram* based on the dictionary from a subset of the logs. In other words, we would like to answer the following question:
Can we generate a dictionary from a small size of log data and correctly parse the rest of the logs without updating the dictionary?

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3007554, IEEE Transactions on Software Engineering



(a) Android (b) BGL (c) HDFS (d) Windows (e) Spark

Fig. 4. The elapsed time of parsing five different log data with various sizes. The x and y axes are in log scale.

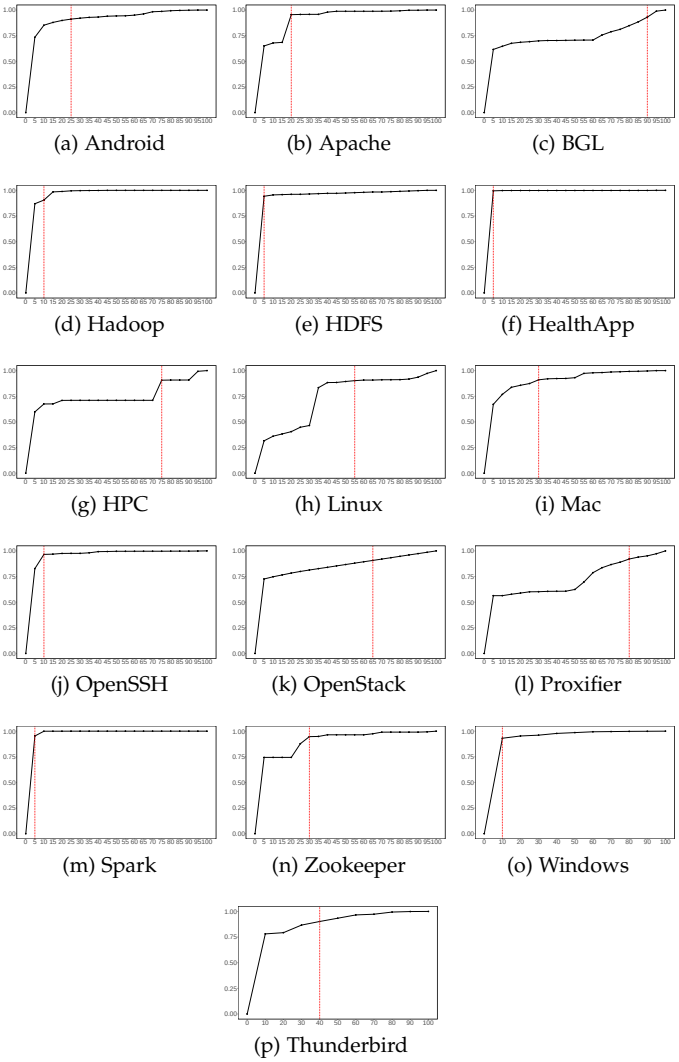
TABLE 4
The number of 2-grams and 3-grams in the dictionaries generated by *Logram*

Dataset	2-gram	3-gram	Dataset	2-gram	3-gram
Android	53,719	57,298	Mac	18,460	24,762
Apache	296	281	OpenSSH	5,718	9,257
BGL	110,237	174,336	OpenStack	3,960	7,865
Hadoop	2,363	2,337	Proxifier	75	82
HDFS	494	516	Spark	9,169	10,157
HealthApp	755	624	Zookeeper	9,222	12,933
HPC	1,813	2,711	Windows	230,639	380,682
Linux	2,880	3,016	Thunderbird	3,747,372	5,136,271

If so, in practice, one may choose to generate the dictionary with a small amount of logs without the need of always updating the dictionary while parsing logs, in order to achieve even higher efficiency and scalability. In particular, for each subject log data set, we first build the dictionary based on the first 5% (or 10% only for large datasets like Thunderbird and Windows) of the entire logs. Then we use the dictionary to parse the entire log data set. Due to the limitation of grouping accuracy found from the last subsection and the limitation of the high human effort needed to manually calculate the parsing accuracy, we do not calculate any accuracy for the parsing results. Instead, we automatically measure the agreement between the parsing result using the dictionary generated from the first 5% (or 10%) lines of logs and the entire logs. For each log message, we only consider that the two parsing results agree to each other if they are exactly the same. We then gradually increase the size of logs to build a dictionary by appending another 5% (or 10%) of logs. We keep calculating the agreement until the agreement is 100%.

Results

***Logram* achieves stable parsing results using a dictionary generated from a small portion of log data.** Figure 5 shows agreement ratio between parsing results from using partial log data to generate an n-gram dictionary and using all log data. The red line in the figures indicates that the agreement ratio is over 90%. In 10 out of 16 studied log data sets, our log parser can generate an n-gram dictionary from less than 30% of the entire log data, while having over 90% of the log parsing results the same as using all the logs to generate a dictionary. In particular, one of the large log datasets (the Spark logs) gains a 95.5% agreement ratio using only the first 5% of the log data to build the dictionary. Our results show that the log data is indeed repetitive. Such results demonstrate that practitioners can consider leveraging the two parts of *Logram* separately, i.e., generating the n-gram dictionary (i.e., Figure 2) may not be needed for every log message, while the parsing of each log message (i.e., Figure 3) can depend on a dictionary generated from existing logs.



(a) Android (b) Apache (c) BGL
(d) Hadoop (e) HDFS (f) HealthApp
(g) HPC (h) Linux (i) Mac
(j) OpenSSH (k) OpenStack (l) Proxifier
(m) Spark (n) Zookeeper (o) Windows
(p) Thunderbird

Fig. 5. The agreement ratio of log parsing results between using a part of log to generate dictionary and using all logs to generate dictionary. The red vertical lines indicate that the agreement ratios reach 90%.

We manually check the other six log datasets and we find three reasons that that may cause the instability. 1) Some parts of the log datasets have drastically different log events than others. For example, between the first 30% and 35% of the log data in Linux, a large number of log messages are associated with new events for Bluetooth connections and memory issues. Such events do not exist in the logs in

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3007554, IEEE Transactions on Software Engineering

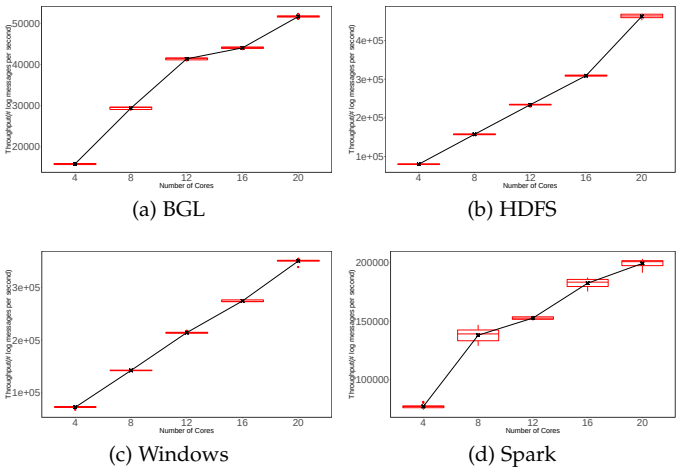
the beginning of the dataset. The unseen logs cause parsing results using dictionary from the beginning of the log data to be less agreed with the parsing results using the the entire logs. However, it is interesting to see that after our dictionary learns the n-grams in that period, the log parsing results become stable. Therefore, in practice, developers may need to monitor the parsing results to determine the need of updating the n-gram dictionary from logs. 2) Log data is too small to reach stabilisation. For example, Proxifier has a very small dataset of logs (only 2.42MB). By using even a smaller part of logs (as small as 5%) to generate the dictionary, the dictionary may have a low coverage of the n-grams in the entire dataset, making it difficult to reach stabilisation when the size of the logs used to generated the dictionary is small. 3) The inaccurate parsing results resulted from mis-split tokens may also lead to instability in our results. When tokens are not split correctly, it may lead to the gradual increase of new n-grams in the dictionary. For example, for OpenStack, we observed a steady increase of the number of n-grams when we increase the size of the dataset to build the dictionaries, which is due to mis-split tokens. In fact, the parsing accuracy of OpenStack is almost the worst among all the datasets (shown in Table 3). As using a smaller part of the logs would generate dictionaries that cannot represent a sufficient coverage of all the n-grams in the dataset, it is difficult to achieve an early stabilisation.

5.5 Scalability

In order to achieve a high-scalability of log parsing, we migrate *Logram* to *Spark*. *Spark* [72] is an open-source distributed data processing engine, with high-level API in several program languages such as Java, Scala, Python, and R. *Spark* has been adopted widely in practice for analyzing large-scale data including log analysis. We migrate each step of *Logram*, i.e., 1) generating n-gram model based dictionaries and 2) parsing log messages using dictionaries, separately to *Spark*. In particular, the first step of generating dictionary is written similar as a typically *wordcount* example program, where each item in the dictionary is a n-gram from a log message. In addition, the second step of parsing log messages is trivial to run in parallel where each log message is parsed independently[4].

We evaluate the scalability of *Logram* on a clustering with one master node and five worker nodes running *Spark* 2.43. Each node is deployed on a desktop machine with the same specifications as used in our efficiency evaluation (cf. Section 5.3). In total, our cluster has four cores for each worker, leading to a total of 20 cores for processing logs. We first store the log data into HDFS that are deployed on the cluster with a default option of three replications. We then run the *Spark* based *Logram* on the *Spark* cluster with one worker (four cores) to five workers (20 cores) enabled. We evaluate the scalability on the same log datasets used for evaluating the efficiency, except for Android due to its relatively small size. We measure the throughput for parsing each log data to assess scalability. Due to the possible noise in a local network environment and the indeterministic nature of the parallel processing framework, we independently repeat each run 10 times when measuring throughput, i.e., number of log messages parsed per second.

4. Due to the limited space, the detail of our implementation of the Spark based *Logram* is available in our replication package.



(a) BGL (b) HDFS

(c) Windows (d) Spark

Fig. 6. Box plots of running time of *Logram* with different number of cores.

Results

***Logram* scales out efficiently with the number of *Spark* nodes without sacrificing parsing accuracy.** Figure 6 uses boxplots to present the throughput (i.e., number of log messages parsed per second) of *Logram* when we increase the number of nodes from one (i.e., four cores) to five (i.e., 20 cores).As shown in Figure 6, the throughput increases nearly linearly, achieving up to 5.7 times speedup as we increase the number of nodes by a factor of five. In addition, Figure 6 shows that the throughput of *Logram* has low variance when we repeat the parsing of each log dataset 10 times. We would like to note that the parsing accuracy always keeps the same as we increase the number of nodes. When the volume of the parsed logs is very large (e.g., the Windows log data), *Logram* allows practitioners to increase the speed of log parsing efficiently by adding more nodes without sacrificing any accuracy.

***Logram* achieves near-linear scalability for some logs but less scalabiltiy on other logs.** A linear scalability means the throughput increases K times when we increase the number of nodes by a factor of K, which is usually the best one usually expects to achieve when scaling an application [73], [74]. The throughput of *Logram* when parsing the HDFS and Windows logs increases by 5.7 to 4.8 times when we increase the number of nodes from one to five, indicating a near-linear or even super-linear scalability. However, *Logram* achieves less scalability when parsing the BGL and Spark logs. Specifically, the throughput of *Logram* when parsing the BGL and Spark logs increases 3.3 and 2.7 times when we increase the number of nodes by a factor of five. We consider that the less promising scalability on BGL and Spark logs may be caused by the imbalanced distribution of unique n-grams. Through examining the BGL and Spark logs, we found that in both logs, some log blocks have much more unique n-grams than other log blocks. *Logram* spends 77% to 92% of its overall parsing time on constructing the n-gram dictionaries (cf. Section 5.3). When *Logram* is deployed in

the Spark environment, each Spark node constructs a sub-dictionary for a block of logs, then the sub-dictionaries are merged into a single dictionary. As the log blocks with more unique n-grams take longer time to build the dictionaries (i.e., the size of the dictionaries is larger), the imbalanced distribution of unique n-grams in the BGL and Spark logs may impair the scalability of *Logram* on these logs.

6 Migrating *Logram* to an Online Parser

Logram parses logs in two steps: 1) generating n-gram dictionaries from logs, and 2) using the n-gram dictionaries to parse the logs line by line. Section 4 describes an offline implementation of *Logram*, in which the step for generating the n-gram dictionaries is completely done before the step of parsing logs using the n-gram dictionaries (even when we evaluate the ease of stabilisation in Section 5.4). Therefore, the offline implementation requires all the log data used to generate the n-gram dictionaries to be available before parsing. On the contrary, an online parser parses logs line by line, without an offline training step. An online parser is especially helpful in a log-streaming scenario, i.e., to parse incoming logs in a real-time manner.

Logram naturally supports online parsing, as the n-gram dictionaries can be updated efficiently when more logs are continuously added (e.g., in log streaming scenarios). In our online implementation of *Logram*, we feed logs in a streaming manner (i.e., feeding one log message each time). When reading the first log message, the dictionary is empty (i.e., all the n-grams have zero occurrence), so *Logram* parses all the tokens as dynamic variables. *Logram* then creates a dictionary using the n-grams extracted from the first log message. After that, when reading each log message that follows, *Logram* parses the log message using the existing n-gram dictionary. Then, *Logram* updates the existing n-gram dictionary on-the-fly using the tokens in the log message. In this way, *Logram* updates the n-gram dictionary and parses incoming logs continuously until all the logs are processed. Similar to Section 5.4, we measure the ratio of agreement between the parsing results of the online implementation and the offline implementation. For each log message, we only consider the two parsing results agreeing to each other if they are exactly the same. We also measure the efficiency of *Logram* when parsing logs in an online manner relative to the offline mode. Specifically, we measure the efficiency difference ratio, which is calculated as $\frac{T_{\text{online}} - T_{\text{offline}}}{T_{\text{offline}}}$ where T_{online} and T_{offline} are the time taken by the online *Logram* and offline *Logram* to parse the same log data, respectively.

Results

The online mode of *Logram* achieves nearly the same parsing results as the offline *Logram*. Table 5 compares the parsing results of *Logram* between the online and offline modes. We considered the same five large log datasets as the ones used for evaluating the efficiency of *Logram* (cf. Section 5.3). The agreement ratio between the online and offline modes of *Logram* range from 95.0% to 100.0%, indicating that the parsing results of the online *Logram* are almost identical to the parsing results of the offline *Logram*.

The online mode of *Logram* reaches a parsing efficiency similar to the offline *Logram*. Table 5 also compares the efficiency between the online and offline*Logram*, for the five considered log datasets with sizes varying from 300KB to 1GB. A positive value of the efficiency difference ratio indicates the online mode is slower (i.e., taking longer time), while a negative value indicates the online mode is even faster. Table 5 shows that the efficiency difference ratio ranges from -3.0% to 8.9%. Overall, the online mode of *Logram* is as efficient as the offline model. In some cases, the online mode is even faster, because the online mode parses logs with smaller incomplete dictionaries – thus being queried faster – compared to the full dictionaries used in the offline mode.

TABLE 5
Comparing the parsing results of *Logram* between the online and offline modes.

Subject log dataset	Efficiency Difference Ratio 300k	1M	10M	100M	500M	1G	Agreement with offline results
HDFS	5.9%	0.0%	-2.4%	-1.5%	-3.0%	-0.8%	100.0%
Spark	0.0%	-0.3%	-0.6%	0.3%	-3.0%	-1.1%	99.9%
Windows	0.0%	0.0%	1.1%	-0.0%	-0.2%	0.6%	96.8%
BGL	7.1%	6.7%	7.2%	5.9%	7.4%	N/A	98.7%
Android	5.9%	8.9%	6.6%	6.5%	N/A	N/A	95.0%

Note: a positive value means that *Logram* is slower in online parsing than in offline parsing.

In summary, as the online mode of *Logram* achieves similar parsing results and efficiency compared to the offline mode, *Logram* can be effectively used in an online parsing scenario. For example, *Logram* can be used to parse stream logs in a real-time manner.

7 Threats to Validity

In this section, we discuss the threat to the validity of our paper.

External validity. In this work, we evaluate *Logram* on 16 log datasets from an existing benchmark [21]. *Logram* achieves a parsing accuracy higher than 0.9 on about half of the datasets. We cannot ensure that *Logram* can achieve high accuracy on other log datasets not tested in this work. Nevertheless, through an evaluation on logs produced by 16 different systems from different domains (e.g., big data applications and operation systems), we show that *Logram* achieves a higher accuracy and a much faster speed than the best existing log parsing approaches. Future work can improve our approach to achieve high accuracy on more types of log data.

When deploying *Logram* on Spark, we demonstrate that *Logram* achieves near-linear scalability for some logs. However, the scalability is less promising for other logs (i.e., the BGL and Spark logs). Through examining the BGL and Spark logs, we observed imbalanced distribution of unique n-grams in the log blocks. As the log blocks with more unique n-grams take longer time to build the dictionaries (i.e., the size of the dictionaries is larger), we suspect that the imbalanced distribution of unique n-grams in the BGL and Spark logs causes the less promising scalability of *Logram* on these logs. Future work may further improve the scalability of parsing logs based on our approach.

Internal validity. *Logram* leverages n-grams to parse log data. n-grams are typically used to model natural languages or source code that are written by humans. However, logs are different from natural languages or source code as logs are produced by machines and logs contain static

and dynamic information. Nevertheless, we show that n-grams can help us effectively distinguish static and dynamic information in log parsing. Future work may use n-grams to model log messages in other log-related analysis. We use an automated approach to determine the threshold for identifying statically and dynamically generated tokens. Such automatically generated thresholds may not be optimal, i.e., by further optimizing the thresholds, our approach may achieve even higher accuracy; while our currently reported accuracy may not be the highest that our approach can achieve.

Construct validity. In the evaluation of this work, we compare *Logram* with six other log parsing approaches. There exists other log parsing approaches (e.g., *LKE* [8]) that are not evaluated in this work. We only consider five existing approaches as we need to manually verify the parsing accuracy of each approach which takes significant human efforts. Besides, the purpose of the work is not to provide a benchmark, but rather to propose and evaluate an innovative and promising log parsing approach. Nevertheless, we compare *Logram* with the best-performing log parsing approaches evaluated in a recent benchmark [21]. Our results show that *Logram* achieves better parsing accuracy and much faster parsing speed compared to existing state-of-the-art approaches.

8 Conclusion

In this work, we propose *Logram*, an automated log parsing approach that leverages n-grams dictionaries to parse log data in an efficient manner. The nature of the n-gram dictionaries also enables one to construct the dictionaries in parallel without sacrificing any parsing accuracy and update the dictionaries online when more logs are added (e.g., in log streaming scenarios). Through an evaluation of *Logram* on 16 public log datasets, we demonstrated that *Logram* can achieve high accuracy and efficiency while parsing logs in a stable and scalable manner. In particular, *Logram* outperforms the state-of-the-art log parsing approaches in efficiency and achieves better parsing accuracy than existing approaches. Finally, we demonstrate that *Logram* can effectively support online parsing (i.e., when logs are continuously generated as a stream) with similar parsing results and efficiency as the offline mode. This is the first work that uses n-grams in log analysis, which demonstrates the success of leveraging a mix of the (un)natural characteristics of logs. *Logram* can benefit future research and practices that rely on automated log parsing to achieve their log analysis goals.

References

[1] 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, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, 2016, pp. 92–101.

[2] 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, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 393–403.

[3] A. J. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings*, 2007, pp. 575–584.

[4] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an MTTF of 1, 000, 000 hours mean to you?" in *5th USENIX Conference on File and Storage Technologies, FAST 2007, February 13-16, 2007, San Jose, CA, USA*, 2007, pp. 1–16.

[5] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, 2009, pp. 117–132.

[6] ——, "Online system problem detection by mining patterns of console logs," in *ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6-9 December 2009*, 2009, pp. 588–597.

[7] J. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection," in *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, 2010.

[8] Q. Fu, J. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6-9 December 2009*, 2009, pp. 149–158.

[9] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008, Beijing, China*, 2008, pp. 307–316.

[10] Q. Fu, J. 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, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 397–400.

[11] "Automated root cause analysis for spark application failures - o'reilly media," https://www.oreilly.com/ideas/automated-root-cause-analysis-for-spark-application-failures, (Accessed on 08/13/2019).

[12] K. Nagaraj, C. E. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, 2012, pp. 353–366.

[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 *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, 2014, pp. 217–231.

[14] 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 *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 402–411.

[15] Y. Dang, Q. Lin, and P. Huang, "Aiops: real-world challenges and research innovations," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019.*, 2019, pp. 4–5.

[16] Q. Lin, K. Hsieh, Y. Dang, H. Zhang, K. Sui, Y. Xu, J. Lou, C. Li, Y. Wu, R. Yao, M. 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, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 480–490.

[17] 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, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 60–70.

[18] N. El-Sayed, H. Zhu, and B. Schroeder, "Learning from failure across multiple clusters: A trace-driven approach to understanding, predicting, and mitigating job terminations," in *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, 2017, pp. 1333–1344.

[19] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang, "Capturing and enhancing in situ system observability for failure detection,"

in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, 2018, pp. 1–16.

[20] 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)*, June 2016, pp. 654–661.

[21] 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, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 121–130.

[22] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25-30, 2017*, 2017, pp. 33–40.

[23] 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, pp. 249–267, 2008.

[24] 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," *Journal of Software: Evolution and Process*, vol. 26, no. 1, pp. 3–26, 2014.

[25] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 102–112.

[26] B. Chen and Z. M. J. Jiang, "Characterizing logging practices in java-based open source software projects - a replication study in apache software foundation," *Empirical Software Engineering*, vol. 22, no. 1, pp. 330–374, 2017.

[27] M. Lemoudden and B. E. Ouahidi, "Managing cloud-generated logs using big data technologies," in *International Conference on Wireless Networks and Mobile Communications, WINCOM 2015, Marrakech, Morocco, October 20-23, 2015*, 2015, pp. 1–7.

[28] H. Li, T. P. Chen, A. E. Hassan, M. N. Nasser, and P. Flora, "Adopting autonomic computing capabilities in existing large-scale systems: an industrial experience report," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 1–10.

[29] H. Li, W. Shang, Y. Zou, and A. E. Hassan, "Towards just-in-time suggestions for log changes," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1831–1865, 2017.

[30] D. El-Masri, F. Petrillo, Y.-G. Guéhéneuc, A. Hamou-Lhadj, and A. Bouziane, "A systematic literature review on automated log abstraction techniques," *Information and Software Technology*, vol. 122, p. 106276, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584920300264

[31] W. B. Cavnar, J. M. Trenkle *et al.*, "N-gram-based text categorization," in *Proceedings of the 3rd annual symposium on document analysis and information retrieval, SDAIR '94*, vol. 161175. Citeseer, 1994.

[32] M. Siu and M. Ostendorf, "Variable n-grams and extensions for conversational speech language modeling," *IEEE Trans. Speech and Audio Processing*, vol. 8, no. 1, pp. 63–75, 2000.

[33] S. Nessa, M. Abedin, W. E. Wong, L. Khan, and Y. Qi, "Software fault localization using n-gram analysis," in *Wireless Algorithms, Systems, and Applications, Third International Conference, WASA 2008, Dallas, TX, USA, October 26-28, 2008. Proceedings*, 2008, pp. 548–559.

[34] A. Tomovic, P. Janicic, and V. Keselj, "n-gram-based classification and unsupervised hierarchical clustering of genome sequences," *Computer Methods and Programs in Biomedicine*, vol. 81, no. 2, pp. 137–153, 2006.

[35] C. Lin and E. H. Hovy, "Automatic evaluation of summaries using n-gram co-occurrence statistics," in *Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics, HLT-NAACL 2003, Edmonton, Canada, May 27 - June 1, 2003*, 2003.

[36] P. F. Brown, V. J. D. Pietra, P. V. de Souza, J. C. Lai, and R. L. Mercer, "Class-based n-gram models of natural language," *Computational Linguistics*, vol. 18, no. 4, pp. 467–479, 1992.

[37] E. Charniak, *Statistical language learning*. MIT press, 1996.

[38] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE'12, 2012, pp. 837–847.

[39] M. Rahman, D. Palani, and P. C. Rigby, "Natural software revisited," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 37–48.

[40] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 532–542.

[41] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*. O'Reilly, 2009.

[42] R. Vaarandi, "Simple event correlator for real-time security log monitoring," *Hakin9 Magazine*, vol. 1, no. 6, pp. 28–39, 2006.

[43] C. V. Damasio, P. Fröhlich, W. Nejdl, L. M. Pereira, and M. Schroeder, "Using extended logic programming for alarm-correlation in cellular phone networks," *Applied Intelligence*, vol. 17, no. 2, pp. 187–202, 2002.

[44] S. E. Hansen and E. T. Atkins, "Automated system monitoring and notification with swatch." in *LISA*, vol. 93, 1993, pp. 145–152.

[45] R. Ramati, "A beginner's guide to logstash grok," https://logz.io/blog/logstash-grok, (Accessed on 08/14/2019).

[46] L. Bennett, "Lessons learned from using regexes at scale," https://www.loggly.com/blog/lessons-learned-from-using-regexes-at-scale/, (Accessed on 08/14/2019).

[47] S. Documentation, "About splunk regular expressions," https://docs.splunk.com/Documentation/Splunk/7.3.1/Knowledge/AboutSplunkregularexpressions, (Accessed on 08/14/2019).

[48] M. Documentation, "W3c logging," https://docs.microsoft.com/en-us/windows/win32/http/w3c-logging, (Accessed on 08/14/2019).

[49] "Apache/ncsa custom log format," https://www.loganalyzer.net/log-analyzer/apache-custom-log.html, (Accessed on 08/13/2019).

[50] M. Documentation, "Iis log file formats," https://docs.microsoft.com/en-us/previous-versions/iis/6.0-sdk/ms525807(v=vs.90), (Accessed on 08/14/2019).

[51] M. Nagappan, K. Wu, and M. A. Vouk, "Efficiently extracting operational profiles from execution logs using suffix arrays," in *ISSRE 2009, 20th International Symposium on Software Reliability Engineering, Mysuru, Karnataka, India, 16-19 November 2009*, 2009, pp. 41–50.

[52] W. Xu, L. Huang, and M. I. Jordan, "Experience mining google's production console logs." in *SLAML*, 2010.

[53] D. Schipper, M. F. Aniche, and A. van Deursen, "Tracing back log data to its log statement: from research to practice," in *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada.*, 2019, pp. 545–549.

[54] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*. IEEE, 2003, pp. 119–126.

[55] M. Nagappan and M. A. Vouk, "Abstracting log lines to log event types for mining software system logs," in *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*, 2010, pp. 114–117.

[56] L. Tang, T. Li, and C. Perng, "Logsig: generating system events from raw textual logs," in *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, 2011, pp. 785–794.

[57] 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*, ser. CIKM '16. New York, NY, USA: ACM, 2016, pp. 1573–1582.

[58] M. Mizutani, "Incremental mining of system log format," in *2013 IEEE International Conference on Services Computing, Santa Clara, CA, USA, June 28 - July 3, 2013*, 2013, pp. 595–602.

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

[60] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *IEEE 16th International Conference on Data Mining, ICDM 2016, December 12-15, 2016, Barcelona, Spain*, 2016, pp. 859–864.

[61] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France, June 28 - July 1, 2009*, 2009, pp. 1255–1264.

[62] P. He, Z. Chen, S. He, and M. R. Lyu, "Characterizing the natural language descriptions in software logging statements," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 178–189.

[63] "loess function — r documentation," https://www.rdocumentation.org/packages/stats/versions/3.6.1/topics/loess, (Accessed on 01/02/2020).

[64] "Ckmeans.1d.dp function — r documentation," https://www.rdocumentation.org/packages/Ckmeans.1d.dp/versions/3.4.0-1/topics/Ckmeans.1d.dp, (Accessed on 01/02/2020).

[65] A. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Commun. ACM*, vol. 55, no. 2, pp. 55–61, Feb. 2012.

[66] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "Towards automated log parsing for large-scale log data analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 6, pp. 931–944, 2017.

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

[68] 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*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 102–111.

[69] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 1285–1298.

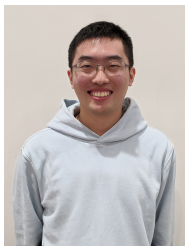
[70] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 575–584.

[71] P. He, J. Zhu, P. Xu, Z. Zheng, and M. R. Lyu, "A directed acyclic graph approach to online log parsing," *arXiv preprint arXiv:1806.04356*, 2018.

[72] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[73] N. J. Gunther, P. Puglia, and K. Tomasette, "Hadoop superlinear scalability," *ACM Queue*, vol. 13, no. 5, p. 20, 2015.

[74] H. Zhu, Z. Bai, J. Li, E. Michael, D. R. K. Ports, I. Stoica, and X. Jin, "Harmonia: Near-linear scalability for replicated storage with in-network conflict detection," *CoRR*, vol. abs/1904.08964, 2019.



Hetong Dai is a Master student in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada, supervised by Weiyi Shang. His research lies within Software Engineering, with special interests in software engineering for ultra-large-scale systems, software log mining and mining software repositories. He obtained his BS from Nanjing University. Contact him at he_da@encs.concordia.ca

Heng Li is a postdoctoral fellow in the School of Computing at Queen's University, Canada. His research lies within Software Engineering and Computer Systems, with special interests in Artificial Intelligence for DevOps, software log mining, software performance engineering, mining software repositories, and qualitative studies of software engineering data. He obtained his BE from Sun Yat-sen University, MSc from Fudan University, and PhD from Queen's University, Canada. He worked at Synopsys as a full-time R&D Engineer before starting his PhD. Contact him at hengli@cs.queensu.ca

Che Shao Chen is studying Master of Software Engineering at Concordia University, Montreal, Canada. His interests and research areas are Software Engineering with special interests in software engineering for ultra-large-scale systems related to Software Refactoring, Data Mining, software log mining, and mining software repositories. He is supervised by Professor Weiyi Shang, who is an assistant professor and research chair at Concordia University. He obtained his BS from Tamkang University. Contact him at c_chesha@encs.concordia.ca

Weiyi Shang is a Concordia University Research Chair at the Department of Computer Science. His research interests include AIOps, big bata software engineering, software log analytics and software performance engineering. He is a recipient of various premium awards, including the SIGSOFT Distinguished paper award at ICSE 2013, best paper award at WCRE 2011 and the Distinguished reviewer award for the Empirical Software Engineering journal. His research has been adopted by industrial collaborators (e.g., BlackBerry and Ericsson) to improve the quality and performance of their software systems that are used by millions of users worldwide. Contact him at shang@encs.concordia.ca; http://users.encs.concordia.ca/ shang.

Tse-Hsun (Peter) Chen is an Assistant Professor in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. He leads the Software PErformance, Analysis, and Reliability (SPEAR) Lab, which focuses on conducting research on performance engineering, program analysis, log analysis, production debugging, and mining software repositories. His work has been published in flagship conferences and journals such as ICSE, FSE, TSE, EMSE, and MSR. He serves regularly as a program committee member of international conferences in the field of software engineering, such as ASE, ICSME, SANER, and ICPC, and he is a regular reviewer for software engineering journals such as JSS, EMSE, and TSE. Dr. Chen obtained his BSc from the University of British Columbia, and MSc and PhD from Queen's University. Besides his academic career, Dr. Chen also worked as a software performance engineer at BlackBerry for over four years. Early tools developed by Dr. Chen were integrated into industrial practice for ensuring the quality of large-scale enterprise systems. More information at: http://petertsehsun.github.io/.