Analysing the current log parsing techniques in large-scale software systems

Stefan Petrescu
TU Delft
Delft, Netherlands
petrescu-1@student.tudelft.nl

Abstract

Due to the complexity of current software systems, the amount of logs generated is tremendous. As a result, it is infeasible to manually investigate this data in reasonable time, thereby missing important information that could help making systems more efficient and more secure. Consequently, effectively using logs requires automating the log analysis process. By using automated log analysis techniques, arbitrary system log sizes can be tackled, thus complementing manual investigations of logs. However, although automated log analysis techniques have proven to be of high utility, their effectiveness depends on the output of a process known as 'log parsing'. Thus, as this is of utmost importance, we examined the state-of-the-art log parsing approaches. For this, we conducted a literature survey in which we identified 34 distinct log parsing approaches. We evaluated the scalability and accuracy for the 14 most representative in the literature. Our findings indicate that, out of these, only four are realistically scalable, namely Drain, IPLoM, LFA, LogCluster. With regards to their accuracy, the best results are produced by NuLog, followed by Drain.

Keywords: log parsing, log abstraction techniques, log mining.

1 Introduction

Logs record runtime information of large software systems to provide an audit trail for monitoring, understanding the activity, or diagnosing problems [37]. For instance, system administrators can follow the logs and conduct investigations in case of any incidents. Furthermore, in case of errors or anomalies, log messages can facilitate the process of generating alerts [41].

Unfortunately, the complexity of modern software ecosystems produces a deluge of log information that, due to the size and the low signal to noise ratio, often remains unanalyzed even though critical information could be destilled from the log content. For example, some systems can produce 30-50 gigabytes of logs per hour [20]. As a result of this, traditional log analysis approaches are not suitable. Thus, it has become infeasible for humans to manually investigate this amount of information in a reasonable amount of time. Automated log analysis techniques are required, in order to realistically utilize system logs.

Most automated log analysis methods require logs to undergo a specific type of transformation in their initial step. In the literature, this process is known as 'log parsing'[1]. Log parsing transforms a raw log message into structured information. More specifically, the goal is to separate the static parts of the original logging statement from the dynamic parts (information available at runtime) of a log message in order to create usable input for downstream tasks. Distinguishing between constants and variables means finding the static template of the raw log message, as displayed in Figure 1. Conducting this type of transformation can be problematic if the number of log messages is large and accuracy needs to be high.

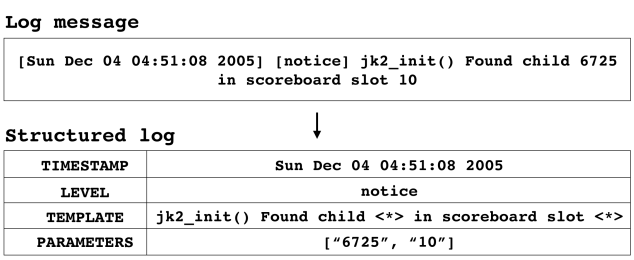

Figure 1. Log parsing example: a raw log message gets transformed into structured information. In other words, the main goal of running a log parser is to distinguish between constants and variables [11].

As the performance of automated log analysis techniques can be directly influenced by the output quality of the log parsing methods [9], it is crucial to find which of these are most appropriate. Hence, this study focuses on the state-of-the-art (SOTA) log parsing approaches in terms of their accuracy and scalability. Specifically, we tackle the following research questions:

RQ1: How has log parsing been approached so far?
RQ2: How do log parsing methods perform in terms of their scalability?
RQ3: How do log parsing methods perform in terms of their accuracy?

[1] 'Log parsing' is sometimes used interchangeably with 'event template extraction' or 'log abstraction'.

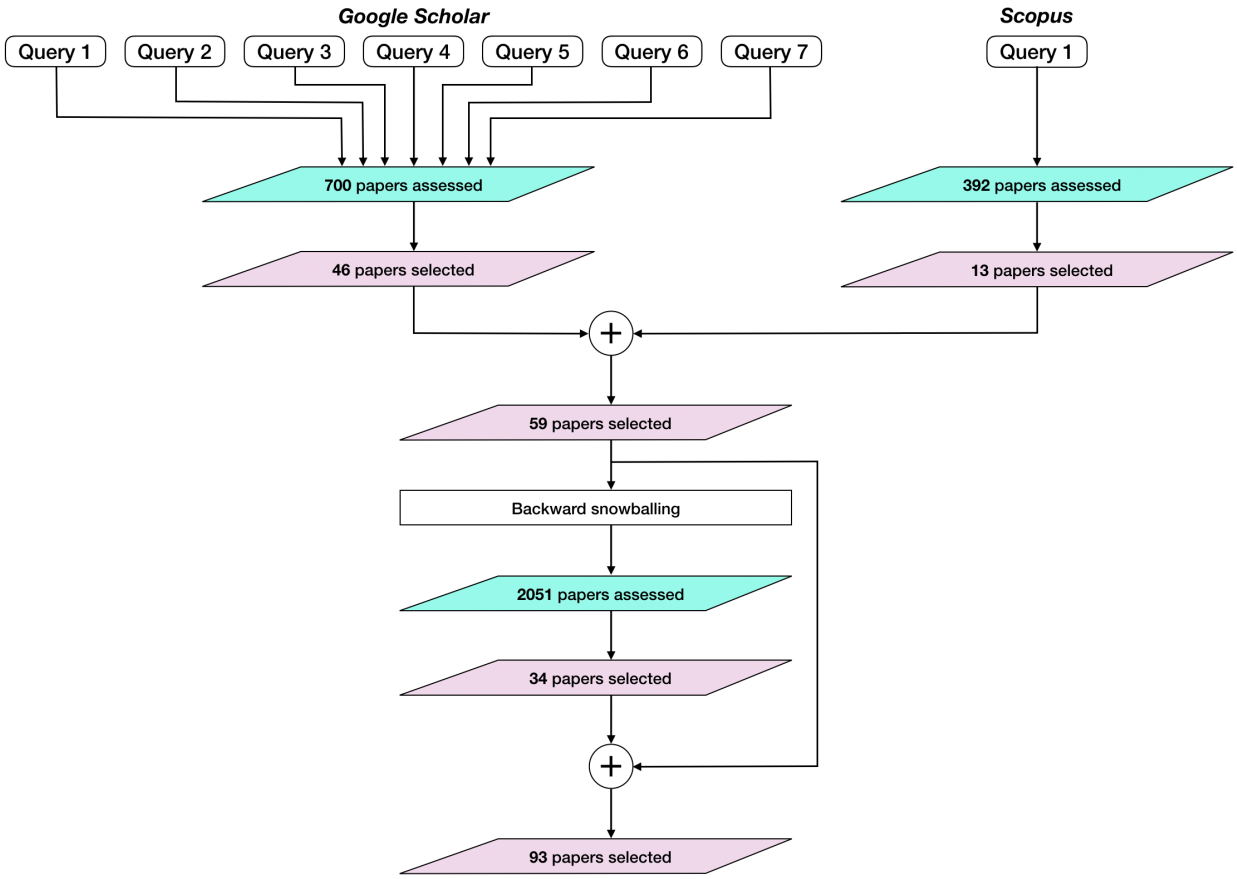

Figure 2. Selection process for the papers chosen for further investigation.

By answering these questions, we bring more structure and clarity to the discussion in order to guide future researchers in identifying current challenges and opportunities. Furthermore, to facilitate future works and reproducibility, we make the code and datasets associated with this paper publicly avaiable[2].

This paper is organized as follows. Section 2 gives an overview of the related work. Section 3 presents the search methodology. Section 4 contains the answers to the research questions. Finally, Section 5 concludes this article and discusses the most important findings.

2 Related work

Works that survey log parsing approaches have been published recently. These outline the various methods present in the literature, and evaluate them based on specific criteria.

The authors of [41] evaluate 13 representative log parsers. They analyze the methods with regards to three aspects, *Accuracy*[3], *Robustness*[4], and *Efficiency*[5]. To do so, the authors run experiments on 16 different log datasets. They make the code and labeled datasets publicly available, with the aim of providing a basis for further developments.

In [11] the authors survey automated log analysis methods in the area of reliability engineering. Within this broader context, they include a chapter that describes log parsing methods. Their contribution is to characterize them based on four aspects, namely *mode*[6], *coverage*[7], *preprocessing*[8], *technique*[9].

To the best of our knowledge, [6] is the only literature survey solely intended to analyze log parsing approaches. This

[2] https://github.com/spetrescu/literature-survey-log-parsing

[3] Accuracy is defined as "the ratio of correctly parsed log messages over the total number of log messages".
[4] Robustness of a log parser is defined as "the consistency of its accuracy under log datasets of different sizes or from different systems".
[5] Efficiency is defined by the processing speed of a log parser.
[6] The mode refers to the ability of an approach to parse logs online or offline.
[7] The coverage refers to how well a log parser can discover as many log templates as possbile for a given log dataset.
[8] The preprocessing refers to having user interventions prior to running a method, for e.g., having users input regular expressions.
[9] The technique refers to the algorithmic approach used by the method, for e.g., clustering or frequent pattern mining.

work aims to bridge the gap between industry and academia, and to do so, the authors use seven criteria that describe log parsing methods, namely *mode*[6], *coverage*[7], *efficiency*[10], *scalability*[11], *system knowledge independence*[12], *delimiter independence*[13], *parameters tuning effort*[14].

In contrast to the related work, in our study we highlight important aspects that have not been tackled before, or that have been tackled incompletely. Firstly, different from other studies, apart from providing an outline of existing methods, we visualise how they cluster together based on their algorithmic approach. This way, we identify potential opportunities for further exploration. Secondly, compared to existing surveys, we conduct experiments that test the scalability of the methods, by taking into account both runtime and memory consumption for various datasets and dataset sizes.

3 Method

A literature search inspired by the methodology proposed in [15] was conducted. In Figure 2, an overview of the initial selection process can be found. Our goal here was to find all possible papers that contained information about log parsing. Papers were included if they (i) proposed a log parsing method, (ii) implemented a log parser in their pipeline, or (iii) referenced other log parsers/log parsing methods. In doing so, only papers on software logs (logfiles), in contrast to mathematical logarithms, were selected. Papers were first assessed based on title and abstract. For problematic cases, the full text was consulted.

The followed process yielded 93 papers. For these, their full text was read in order to decide if they proposed/introduced a log parsing approach. This resulted in the inclusion of 32 papers for the final selection. This process is visualised in Figure 3. The selected papers were split in two main categories, namely online and offline approaches (details in Section 4). Last but not least, during the writing of this document, we applied an additional filtering process, based on two aspects. Firstly, we excluded papers for which the authors explicitly mentioned that their method was preliminary (we only considered work that was feature complete). Secondly, we did not consider papers that had their technical description insufficient to reproduce.

[10]Here, efficiency refers to how well a method performs in terms of runtime/memory complexity. It can be of two types, namely high efficiency or low efficiency.
[11]A method is considered to be scalable if it allows parallelization.
[12]System knowledge independence refers to how a method makes use of heuristics, hard-coded rules etc.
[13]Delimiter independence refers to a log parsing algorithm requiring (or not) pre-defined rules for delimiters.
[14]Parameters tuning effort refers to how much parameter tuning is required for the method to perform correctly.

Practically, we started the selection process by querying two well-known databases, Google Scholar[15] and Scopus[16]. For Google Scholar, we consulted the first 10 pages of results (first 100 results, any time, sort by relevance, any type). For Scopus, we consulted all of the 392 returned results. The queries used can be found in Appendix A.

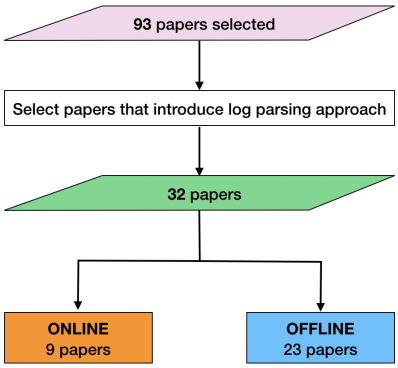

Figure 3. Final selection process of papers that proposed log parsing approaches.

4 Results

In this section we briefly present each log parsing approach found and mention its main contribution(s) and biggest disadvantage(s). Furthermore, two representations of how the methods are clustered are visualised. Each subsection tackles one corresponding research question.

4.1 Log parsing approaches

Inspired by similar works [6] [11], we distinguish between the methods by using their *mode*. This leads us into categorising the approaches in two main branches, *offline* and *online*. The distinction between these two modes is very important, and it has to do with the manner in which the log data is being processed.

Offline approaches process log data in batches, and are able to discover templates given a static set of log messages. They require a training phase, during which the templates are discovered. After this, they are able to parse incoming logs by matching with the templates found previously, either in batch or stream [6]. As changes/updates in software can generate new log templates, one drawback of using this type of approach is that it requires the training phase to be re-run periodically in order to discover these.

Online approaches process log data item by item in a streaming manner, and do not require a batch of data to be available prior to executing. More specifically, they are

[15]https://scholar.google.com
[16]https://www.scopus.com

able to discover event templates without requiring an offline training phase. Furthermore, as event templates are being updated dynamically, such methods can be integrated seamlessly for downstream tasks [11]. Online parsers are recommended for use cases where the decision time for certain actions needs to be really short (e.g., trying to predict incidents in a software system) and logs need to be processed on the fly.

As a general overview, Table 1 contains a list of the selected offline log parsing approaches. In turn, Table 2 contains a list of the selected online log parsers.

Table 1. Overview of all offline log parsing approaches found in the literature.

Year	Method used
2003	SLCT [35]
2008	AEL [14]
2009	LKE [7]
2010	LFA [23]
2011	LogSig [32]
2012	IPLoM [18]
2013	HLAer [26]
2014	NLP-LTG [16]
2015	LogCluster [36]
2016	LogMine [8]
2017	NLM-FSE [34]
2017	POP [9]
2018	MoLFI [19]
2020	LPV [40]
2020	NuLog [25]
2020	ELA [31]
2021	AWSOM-LP [28]
2021	LogStamp [33]

4.1.1 Offline log parsing approaches. Figure 4 shows a clustering of the different offline methods based on their algorithmic approach. The connection between two methods is represented by an arrow. More specifically, we considered two methods connected, if either one of them evaluated against the other, or if one of them inherited its approach from the other. This way, we observe which types of algorithms test against which types – it can be noticed that most authors take into account others' methods when evaluating their own. Notable exceptions include the work on LPV, which renders the claimed benefits at least questionable. Furthermore, it can be seen that the biggest group is the one that uses clustering algorithms, whereas approaches that use heuristics or evolutionary algorithms have only one method present in their cluster. Lastly, the figure suggests that the log parsing community is very active (indicated by the various connections between the methods/clusters).

SLCT (Simple Logfile Clustering Tool) [35] uses a data clustering algorithm to search for 'event types' (log templates). Finding these means discovering the constant parts of a log message – the same as log parsing. Practically, a frequent pattern mining algorithm is applied, consisting of three steps. During the first data pass, a table of frequent words is constructed (a word is considered to be frequent if it appears in the log data at least N times – a user specified threshold). For the second step, cluster candidates are formed. More specifically, log lines that contain one or more frequent words (found previously) are added to a candidate table (initially empty). The third step involves inspecting the candidate table previously generated, and, based on the user specified threshold N, clusters are reported – the log templates.

AEL (Abstracting Execution Logs) [14] proposes a rule-based approach that consists of three steps. The first step requires the use of heuristics (rules tailored for the specifics of a particular log dataset) in order to replace the dynamic parts of a log message with generic tokens[17]. One of the two heurstics regards all "word=value" pairs present in a log message as containing dynamic information, thus replacing *value* with a generic token. For example, the log message: `"Data points amount to d=20"` gets transformed into `"Data points amount to d=$v"`. The second step clusters logs that are similar to each other in different groups (bins). For this step, the authors consider logs to be similar in terms of two aspects, namely the number of words and parameters (generic tokens) of a log line. The third step of the method iterates through all the previously created groups and returns the log templates. More specifically, by using a similarity metric, every log line within a specific group is compared against all the others. Logs that are similar to each other are considered to belong to the same template. Unfortunately, the authors do not specify how the similarity metric is computed. Finally, the log templates are returned. One of its biggest disadvantages is that, if the first step cannot be followed, the method cannot be used as the similarity comparisons cannot be made anymore. For example, the authors of LFA were unable to use AEL due to being unable to follow AEL's first step.

LKE (Log Key Extraction) [7] proposes a three-step clustering approach. During the first step, logs undertake a preprocessing transformation, in which parameters are removed by regular expressions. Parameters are defined by the user by leveraging domain specific knowledge (for example, IP addresses etc.). During the second step, log messages are clustered based on a similarity metric, namely a weighted edit string distance. During the third step, the clusters are refined by means of additional heuristics (for example, authors

[17] AEL replaces the dynamic parts of a log message with "$v".

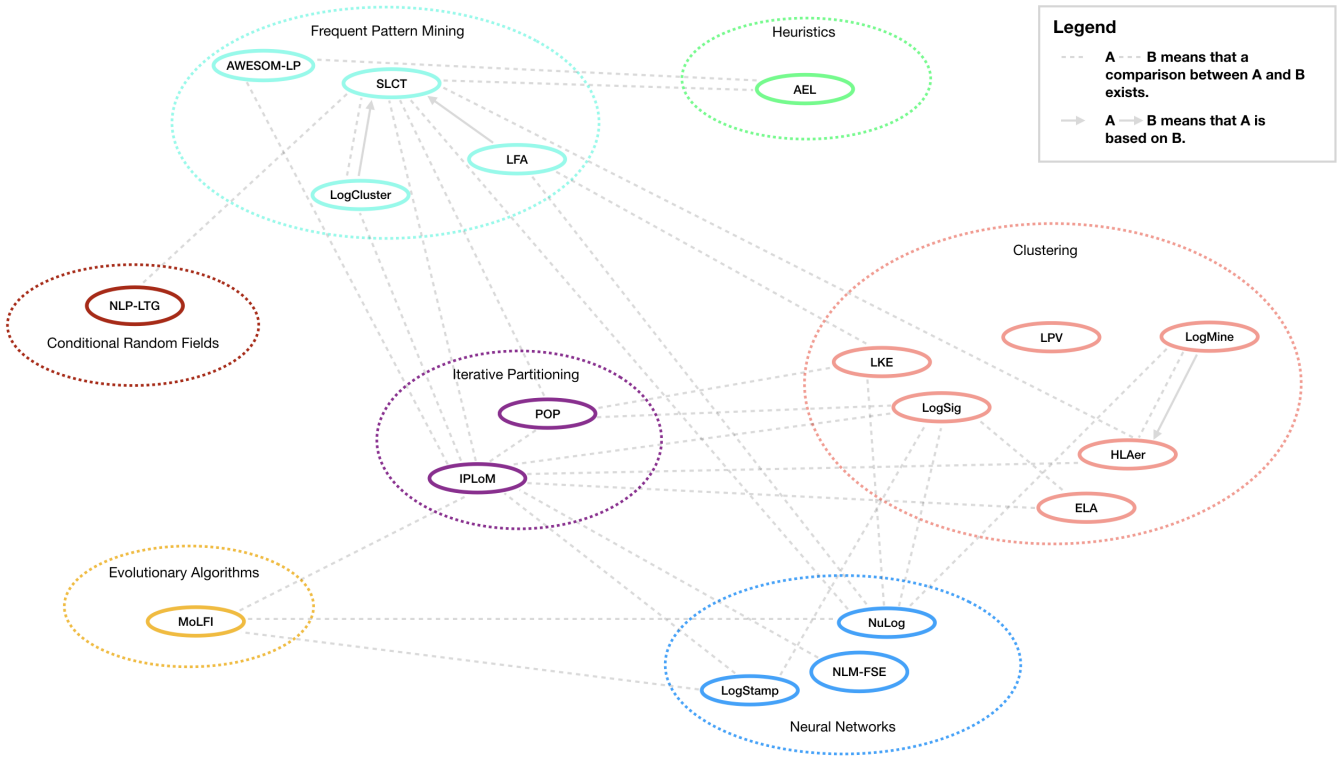

Figure 4. Graphical representation of how the offline methods are clustered together based on their algorithmic approach. The arrows represent connections between various methods. More specifically, an arrow means that the method from which it originates, evaluated against the method towards which it points.

consider two logs having different templates if the frequency of a word at a specific position is lower than a certain threshold q). The biggest disadvantage of using this method is that it involves hand-crafted rules, such as regular expressions or thresholds. However, these might not be discoverable, valid or might not hold for different datasets.

LogSig [32] proposes a novel clustering algorithm, consisting of 3 main steps. It first splits log lines into pairs of tokens by extracting each pairwise of terms[18]. During its second stage, log messages are partitioned into k groups, k being a support value parameter. The third and final step implies creating the message signatures from the previously partitioned groups. Compared to LKE, instead of directly clustering log messages, LogSig transformed each log message into a set of word pairs and clustered logs based on the corresponding pairs. In terms of its results, LogSig displays poor scalability and accuracy results.

IPLoM (Iterative Partitioning Log Mining) [18] tackles log parsing by proposing a 4-step hierachical partitioning algorithm. During the first stage, log lines are partitioned by their token[19] count. This step assumes that log messages that have the same template are more likely to have the same number of tokens. The second step involves partitioning by the token position. Here, the authors assume that, for a log that has a length of n tokens, the column with the least variability is most likely to contain static parts (parts of the template). The third step represents partitioning by "search for bijection" – a mapping between the set of unique tokens (suspected to be apart of the log template). During the fourth and final step, log templates are returned.

HLAer (Heterogeneous Log Analyzer) [26] proposes a 3-step log parsing approach. During its first stage, logs are tokenized (using the whitespace character as the delimiter). More specifically, all words and special symbols (except numbers) are separated by an empty space. For example, the log message: `"GET /images/header/nav.gif"` gets transformed into `"GET / images / header / nav . gif"`. For the second step, the authors cluster logs using the *OPTICS*[20]

[18]For example, `"Component 042alt0042 state HWID=3180"` is split in the following pairs: `{Component, 042alt0042}`, `{Component, state}`, `{Component, HWID=3180}`, `{042alt0042, state}`, `{042alt0042, HWID=3180}`, `{state, HWID=3180}`

[19]Any sequence of characters separated by whitespaces is considered to be a token – a word, a number, an IP address etc. For example, *"Connection from 120.0.0.1"* has 3 tokens.

[20]Although the *OPTICS* algorithm has an $O(n^2)$ memory complexity [6], the authors HLAer still claim the method to be scalable as it can be parallelized.

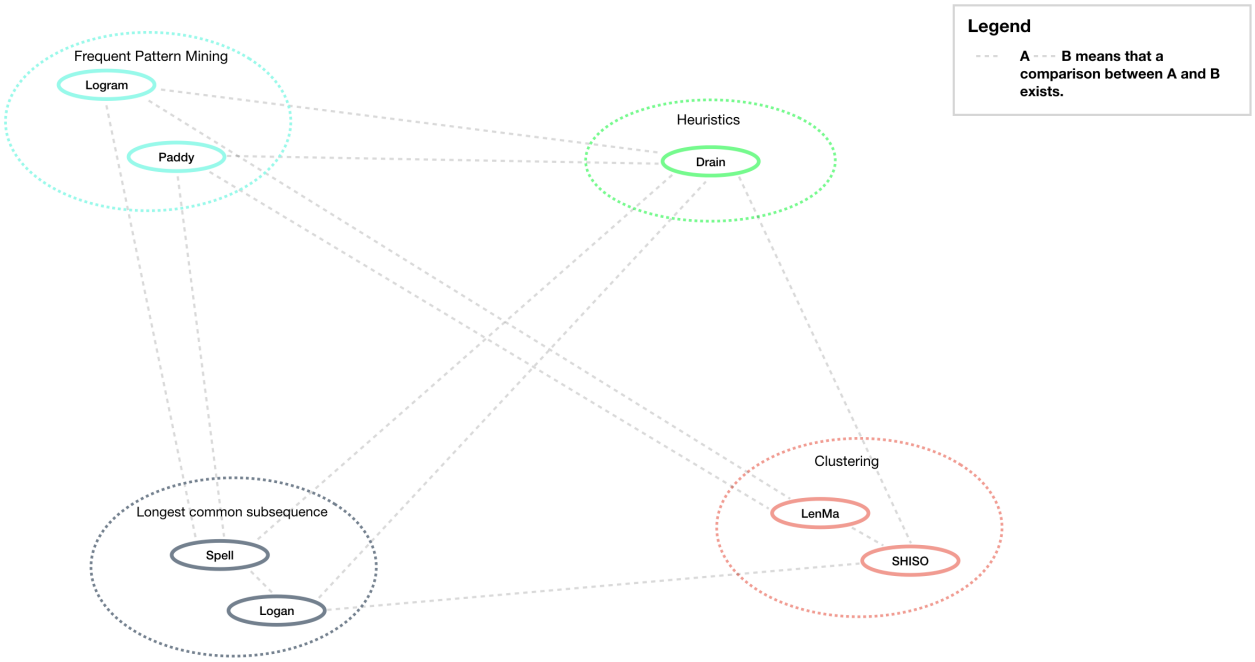

Figure 5. Graphical representation of how the online methods are clustered together based on their algorithmic approach. The arrows represent connections between various methods. More specifically, an arrow means that the method from which it originates, evaluated against the method towards which it points.

[2] algorithm. During the third stage, for each of the previously found clusters, a pattern recognition algorithm is applied such that event templates are found and returned.

NLP-LTG (Natural Language Processing–Log Template Generation) [16] proposes an approach based on a statistical modelling technique, namely Conditional random field [39] (CRF). To leverage this technique, the authors construct a labeled dataset where words in log messages are annotated as either being a 'Description' or a 'Variable'. The former corresponds to the content of the log template. The latter corresponds to the dynamic parts of a log messages generated at runtime, intended to be removed.

LogCluster [36] was designed to overcome the shortcomings of SLCT and IPLoM by introducing a frequent pattern mining algorithm, consisting of 3 steps. In contrast to SLCT and IPLoM, LogCluster does not take the words' positions into account. During its first step, it discovers frequent words in the log dataset. During its second stage, it generates a set of candidate clusters. During its third step, it drops all cluster candidates that have a counter value lower than the selected support threshold, and subsequently reports the remaining candidates as log templates. An advantage of this method is that it also finds templates for log messages with a variable parameter value length (which proved to be a problem for SLCT and IPLoM). For example, "`user Fiona workerEnv in error state 7`" and "`user Peter Pan workerEnv in error state 9`" have the same event template "`user <*> workerEnv in error state <*>`", while the length of the parameter of the values "Fiona" and "Peter pan" vary. An important observation is that if LogCluster is run with low support threshold values, the results are similar to the ones returned by SLCT.

LogMine [8] introduces a clustering approach for log parsing, consisting of 4 steps. The first step of the algorithm involves, tokenization and type detection (for e.g., a type can be a date, timestamp, IP address, number etc.). LogMine replaces detected variables, such as numbers or dates with the name of the field. For example, after the first step, the log message "`session opened for user anna uid = 10`" becomes "`session opened for user name uid = number`". As the second step, a clustering of the logs is done (with an approach-specific distance metric). During the third step, an algorithm is used to recognize patterns in logs. For each of the previously found clusters, a single pattern is representative of all other instances of a cluster. As their final step, the authors introduce an algorithm that generates a hierarchy of patterns, and returns log templates.

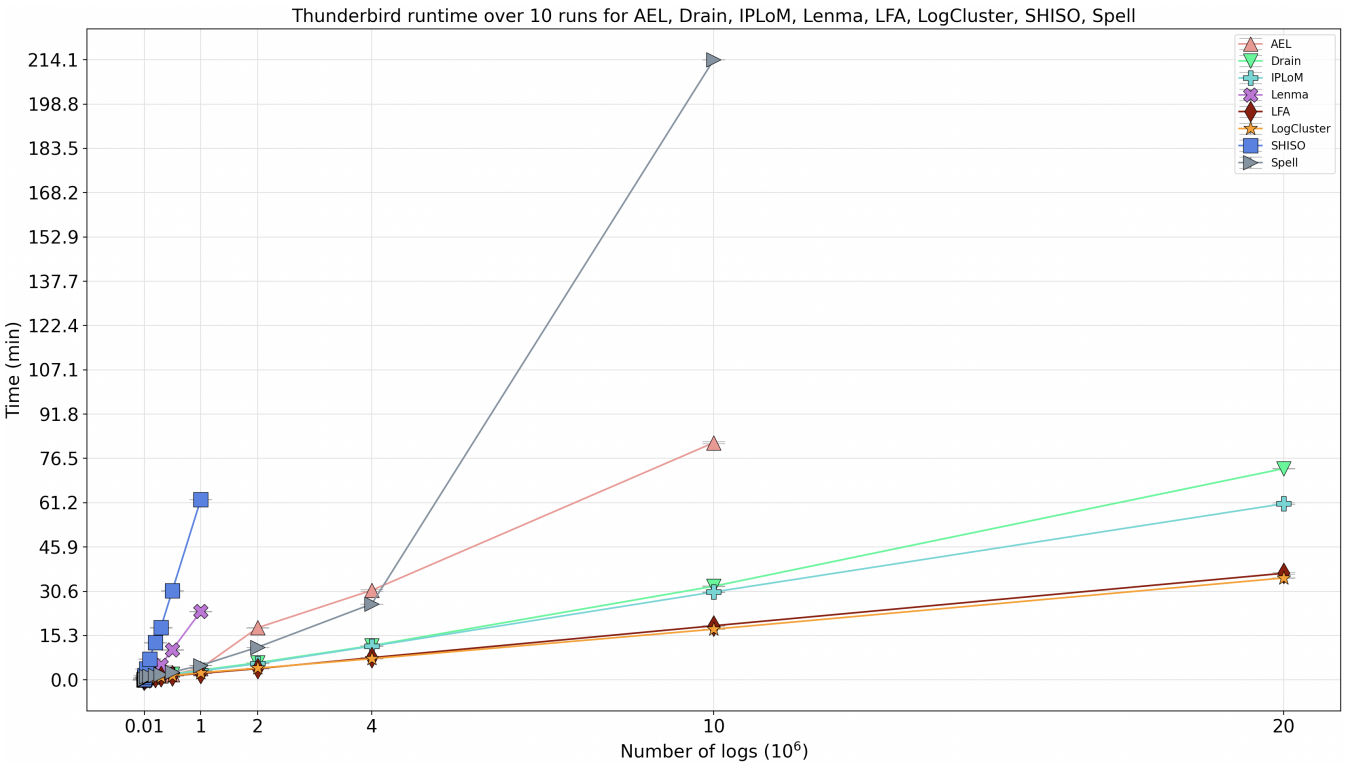

Figure 6. Runtime measurements using the Thunderbird dataset for Drain, Spell, LogMine, and SLCT.

NLM-FSE (Neural language Model-For Signature Extraction) [34] proposes an approach that leverages the use of neural networks. Compared to all the other methods presented in this paper, this approach analyzes log messages at character level, rather than at token level. For training the network, the authors create a synthetic dataset, and annotate each character of each log line, as either being mutable or non-mutable. The former means that the character belongs to a a dynamic part of the logs. The latter means that the character is apart of the log template. Although an interesting approach, its biggest disadvantage is that it is not applicable as it achieves poor results (even on synthetic data).

POP [9] proposes a five-step iterative partitioning method. The first step involves preprocessing log messages by domain knowledge. During this step the main goal is to exclude variable parts that can be easily identified with domain knowledge by means of using regular expressions. During the second step log messages are clustered using a particular metric, namely message length. Specifically, the length of a log message means the number of tokens. As it is possible for logs to have the same number of tokens, even though they belong to different templates, during the third step each group is recursively partitioned into subgroups. The goal here is to obtain subgroups that contain log messages that belong to the same log template. Two messages are considered to belong to the same event type, if the tokens in some positions are the same. During the fourth step the log templates are generated using the tokens' frequency. Specifically, for a given position, if a token appears in all log instances of that group, the token is considered to be apart of the log template. Otherwise, it is replaced with a wildcard. During the fifth and final step, a hierarchical clustering algorithm is used to merge the previously formed subgroups. The reason behind for this is that it might be that some of the groups actually contain over-parsed messages, causing the appearance of a higher number of false negatives. When the final clusters are formed, they are merged into a single log template, facilitated by calculating the Longest Common Subsequence [17]. One of its main advantages is that the method is very fast, allowing for parallel computations. However, it relies on heuristics and it accuracy results are not surpassing other approaches.

LPV [40], uses a 4-step algorithm that leverages vector embeddings to discover log templates. During the first stage, preprocessing is applied. Specifically, duplicates[21] are removed and common variables are substituted (for example, all IP addresses are substituted with a special token, `$$IPADDR$$`).

[21]For example, although `"2005-06-03-15.42.50 instruction cache parity error corrected"` and `"2005-06-03-15.42.53 instruction cache parity error corrected"` have different meta-level information (i.e., different timestamps), they are considered to be duplicates because their content is the same.

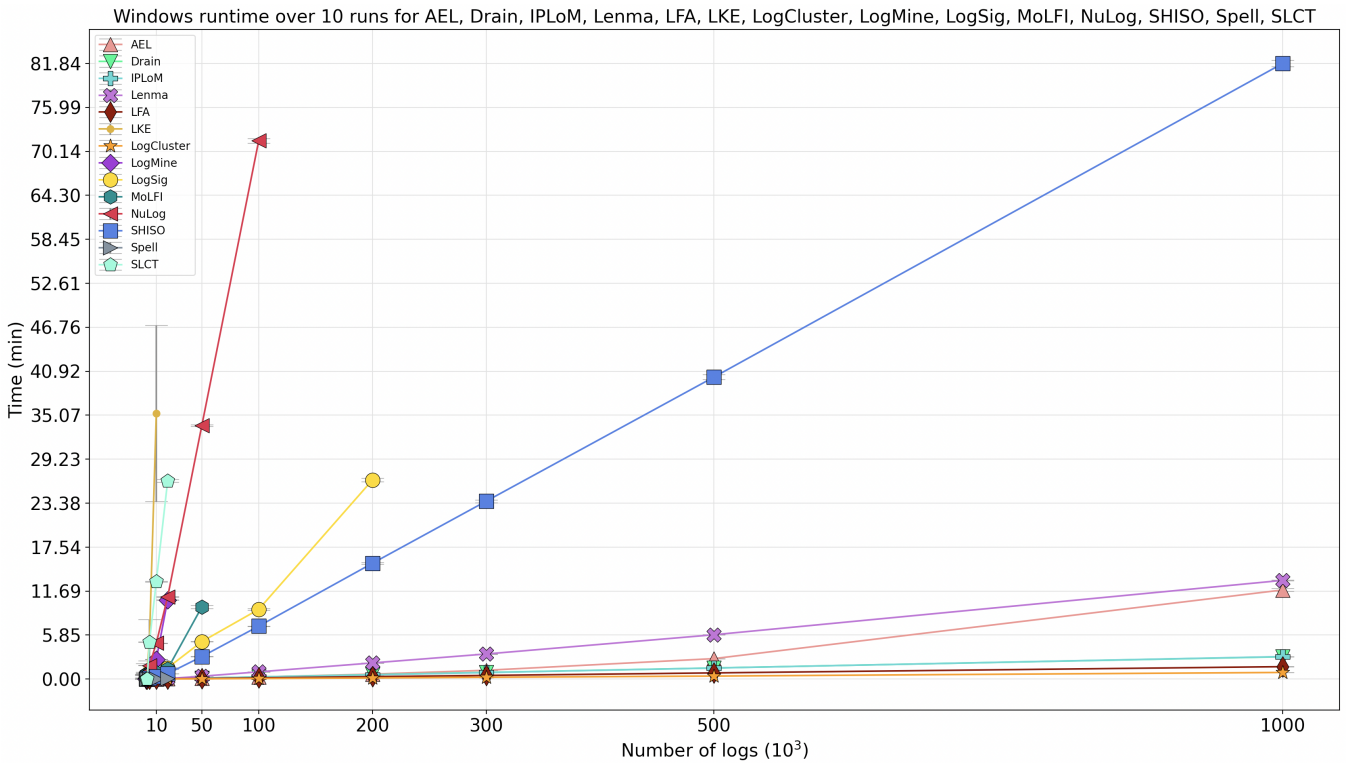

Figure 7. Runtime measurements using the Windows datastet for Drain, IPLoM, Lenma, LogCluster, AEL, SHISO, LFA, Spell, SLCT, LogSig, NuLog, MoLFI, LogMine, LKE. AEL, Lenma, LFA, LogCluster, Drain, IPLoM are able to parse 1M in under 15 minutes. Other methods display different behaviours, for

During the second step, the previously substituted log messages are now embedded into vectors, using *word2vec* [21]. More specifically, each word token from a log message is mapped to a vector. Next, each sequence of tokens (log message) is mapped to a vector – they sum all the tokens' vectors and obtain the representation[22]. During the third stage the log representations are clustered, based on their semantic distance. During the fourth and final step, log templates are extracted.

NuLog (Neural Log) [25] formulates log parsing as a self-supervised[23] learning task. The method has two operation modes, namely the training phase and execution phase. The former is used for training the model for log parsing, whereas the latter is used during the execution (i.e., when log templates are generated by forward passing log messages through the offline trained model). Training the model consists of 2 steps, namely tokenization and masking. For tokenization, each log message is transformed into a sequence of tokens, words being separated by using a specific delimiter (e.g. whitespace). For masking, the authors use a general method from natural language processing called *Masked Language Modelling*. Practically, a random token is replaced by the special <MASK> token for each sequence of tokens (log message). Then, each token sequence is padded with two delimiter tokens, namely <CLS> and <SPEC>. Finally, after training, the model is able to return log templates.

ELA (Event Log Abstraction) [31] uses a 5-step algorithm to parse logs. Compared to other methods such as Drain or AEL which rely heavily on heuristics, it is capable of parsing logs without any user input or hard-coded rules. The first step consists of an automatic preprocessing operation, namely running *nerlogparser* [30] and identifying all the unique log messages. This procedure consists of splitting and labelling each field for each log entry (*nerlogparser*), whereas the second simply means extracting all the different (now-)preprocessed messages[24]. During its second step, the algorithm groups logs based on the word count – they assume that logs that contain the same number of words are likely to belong to the same log template. As the third step, the authors construct a graph model using the count-based

[22]This way, they ensure that substituted logs that have the same template are close to each other in the vector space.

[23]Self-supervised learning can be regarded as the intermediate between supervised and unsupervised machine learning [38]. Specifically, in supervised-learning, machine learning models make use of unlabeled data to yield labels.

[24]For ELA, during its first step, unique messages refer to messages that differ from all other previously parsed log lines.

word groups from the previous step, and, during its fourth step, the authors cluster the log entries using an automatic approach (still, no user parameters are required). During the fifth and final step, log templates are returned. The biggest advantage of ELA is that it does not require any hyperparameters, nor any knowledge about log datasets particularities.

AWSOM-LP [28] proposes an approach that tackles log parsing by using a frequency analysis technique. More specifically, it is composed of 3 steps. During its first stage, domain specific pre-processing is applied by using regular expressions (user input). The second step consists of grouping (clustering) log messages based on a string similarity metric. During the third and final stage, frequency analysis is applied in order to distinguish between constants and variables. In practice, this means counting the number of occurrences of each term for all log messages (that belong to a previously found cluster). Additionally, a post-processing operation is conducted (all numbers that are still present are considered to be variables). Finally, the log templates are returned.

LogStamp [33] introduced a novel approach that tackles log parsing as a sequence labelling task. More specifically, they train a model able to classify the tokens of a log message as either being constant or variables. This is achieved by training a classifier that serves as a tagger. The training data is obtained automatically, by using two different processes (both are aided by BERT [4], which is used for feature representations of log messages). The first process is designed to embed log messages at a coarse level[25], and then, obtaining pseudo-labels for the input data by means of clustering. The second process, embeds log messages at a fine-grained level, which are then passed as input to the classifier. Thus, using the coarse and fine-grained level representations, a classifier is trained to find log templates. Finally, the classifier is used during execution, to predict log templates.

4.1.2 Online log parsing approaches. Figure 5 contains a clustering of the online methods based on their algorithmic approach. In comparison to the offline methods, as the number of online methods is smaller, naturally the number of formed clusters (and respective members) is smaller. Thus, it is premature to draw any conclusions in terms of which algorithmic approach is most preferred within the community. The connection between two methods represents the same concept as in Figure 4.

LenMa [29] proposes a 5-step clustering algorithm with a similarity based on the length of the words in a log message.

[25]Representing logs at a coarse level means extracting features that encompass information about the entire log message, rather than choosing a representation with too many details. Here, the coarse representation means extracting features that allow for distinguishing between logs that are very different from each other. For example, a coarse level representation would mean being able to tell that `"authentication failure; logname= uid=0"` is different from `"check pass; user unknown"`. However, a coarse level representation is not able to distinguish between messages that are similar, but apart of a different underlying log template.

Table 2. Overview of all online log parsing approaches found in the literature.

Year	Paper
2013	SHISO [22]
2016	LenMa [29]
2017	Drain [10]
2019	Spell [5]
2019	Logan [1]
2020	Logram [3]
2020	Paddy [13]

It first creates a word length vector, and a word vector of the message. Secondly, it calculates a similarity metric between the incoming messages and the clusters that have the same number of words. Based on a threshold value T_c, if an incoming log message is not similar enough, a new cluster is created and returned. The most similar cluster is updated with the newly arrived message, and returned.

Drain [10] proposed a 5-step approach that relies heavily on heuristics. It can achieve high parsing accuracies for various datasets and it is recommended for use-cases where log data's diversity is relatively low. As the approach is highly dependent on hard-coded rules, accounting for changes in the structure of log data can be error-prone and hard to maintain. During its first step, logs are preprocessed by using domain knowledge – here, users have to provide regular expressions. During the second, third, and fourth step, the authors construct a 'parse-tree' – a tree structure that allows logs to be parsed using a number of heuristics (for e.g., log messages with the same number of tokens are more likely to have the same log template, the first token of a log message is always constant and apart of the log template, etc.). Finally, logs' tokens are compared using a similarity metric. Finally, the log template is returned and the parse-tree is updated.

Paddy (Parsing Approach with Dynamic Dictionary) [13] is a 4-step log parsing algorithm. During the first stage, log data is preprocessed using domain knowledge by means of regular expressions (provided by users). During the second stage, using an inverted index (implemented using a dictionary structure), log template candidates are retrieved. During the third stage, these are ranked using a similarity metric. More specifically, the similarity metric is a weighted sum (coefficients as hyper-parameters) between *Similarity* (Jaccard similarity [27]) and *LengthFeature* (the length of a log message in terms of number of tokens). During the fourth and final step, the log template is returned and the inverted index is updated.

4.2 Scalability of log parsing approaches

In order to test scalability, we measured the runtime and the memory consumption of each of the methods. Using the

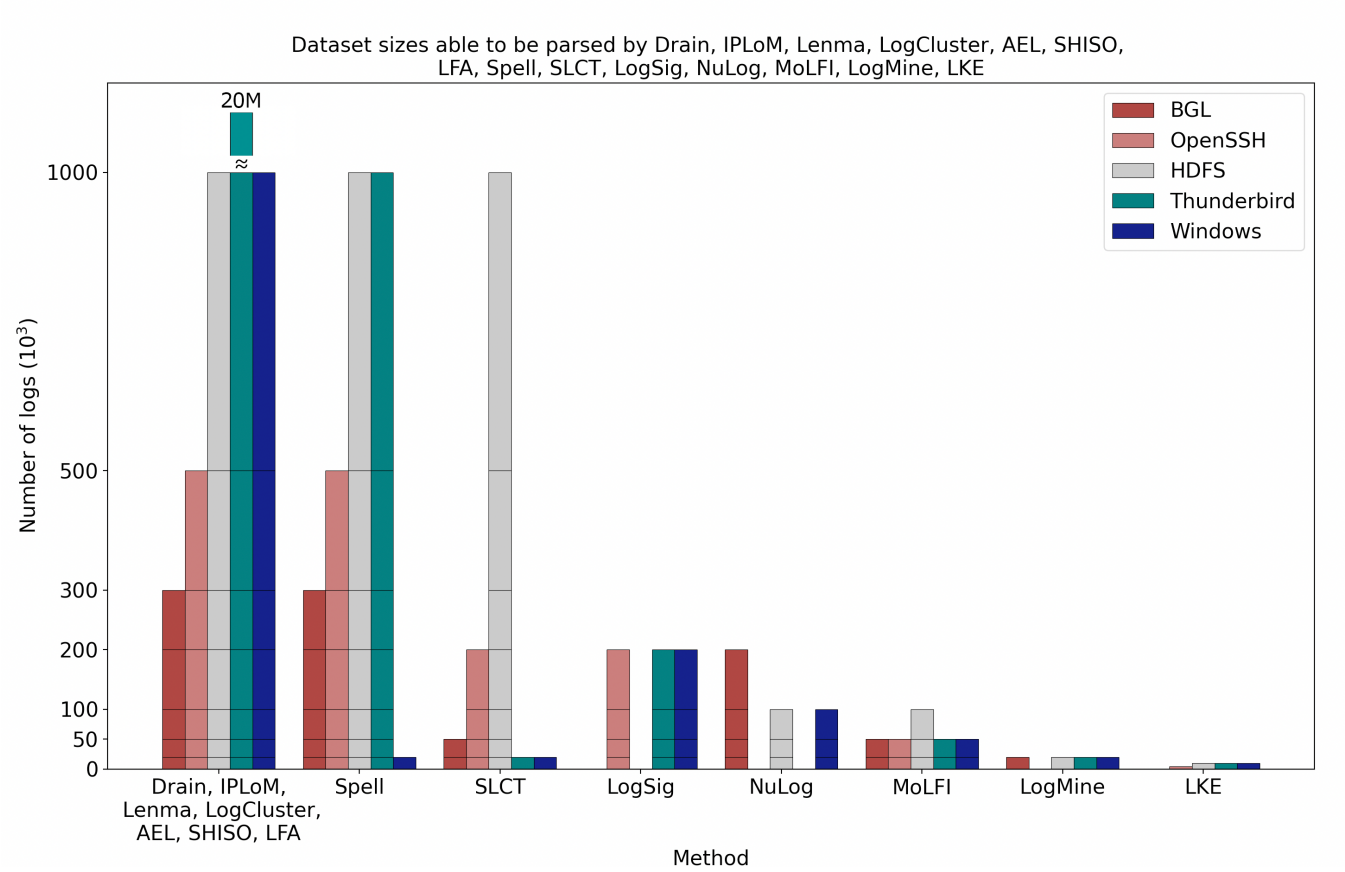

Figure 8. Dataset sizes used for the scalability experiments for Drain, IPLoM, Lenma, LogCluster, AEL, SHISO, LFA, Spell, SLCT, LogSig, NuLog, MoLFI, LogMine, LKE. If a method parsed a dataset of size *N*, all smaller splits were also parsed (for example if Drain parsed the 1M Windows dataset, it also parsed 1k, 2k, 4k, 10k, 20k, 50k, 100k, 200k, 300k, 500k log lines during the experiments)

Table 3. Parsing accuracy results after running each method 10 times (for each dataset). For the experiments, datasets comprised of 2k logs were chosen. The results are averaged over 10 runs.

Dataset	AEL	Drain	IPLoM	Lenma	LFA	LKE	LogCluster	LogMine	LogSig	MoLFI	NuLog	SHISO	SLCT	Spell
BGL	0.957	0.962	0.939	0.689	0.854	0.127	0.835	0.723	0.226	0.948	**0.975**	0.711	0.572	0.786
OpenStack	0.757	0.732	0.341	0.742	0.200	0.787	0.695	0.743	0.866	0.213	**0.886**	0.721	0.867	0.764
HDFS	0.997	0.997	**1.000**	0.997	0.885	**1.000**	0.546	0.850	0.849	0.997	0.953	0.997	0.545	**1.000**
Apache	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	0.708	**1.000**	0.582	**1.000**	0.916	**1.000**	0.730	**1.000**
HPC	0.903	0.887	0.829	0.829	0.816	0.845	0.787	0.784	0.354	0.824	**0.944**	0.324	0.838	0.654
Windows	0.689	0.997	0.566	0.565	0.588	0.989	0.713	0.992	0.689	0.679	**0.998**	0.700	0.696	0.988
HealthApp	0.567	0.780	0.821	0.174	0.548	0.474	0.530	0.684	0.235	0.382	**0.869**	0.397	0.331	0.639
Mac	0.763	0.786	0.673	0.698	0.599	0.221	0.603	0.872	0.477	0.650	**0.826**	0.595	0.557	0.756
Spark	0.905	0.920	0.920	0.883	0.993	0.380	0.798	0.575	0.543	0.650	**0.997**	0.906	0.685	0.905
Avg.	0.838	0.896	0.788	0.731	0.721	0.627	0.691	0.803	0.536	0.707	**0.926**	0.706	0.647	0.833

source code provided by the authors of [12], we modified the benchmarks' code in order to account for both metrics. In order to reduce the measurements noise, we ran each of the experiments 10 times, for each dataset, and for its respective size (Table 4). The experiments were run on a dual socket AMD Epyc2 machine with 64 cores in total (with a dual Nvidia RTX 2080Ti graphics card setup for NuLog). In the following sections, we discuss how we constructed the datasets used for the experiments, and discuss the empirical results obtained.

4.2.1 Constructing the datasets. Using the log data provided by the authors of [12], we created six different datasets. We selected datasets that had at least 300k log lines, namely Android, BGL, OpenSSH, HDFS, Thunderbird, and Windows. The way in which we constructed the datasets was to select the first k lines from the original data (for example, if a dataset had 330k log lines in total, we selected only the first 300k log lines; if a dataset had 2.5 million log lines in total, we selected only the first million log lines). In order to see how the methods scale, the datasets were split in a number of different sizes, namely 1k, 2k, 4k, 10k, 20k, 50k, 100k, 200k, 300k, 500k, 1M log lines. In order to explore the limits of parsing large datasets, we also constructed 20M log lines dataset. For this, we used Thunderbird log data. A visualisation of the constructed datasets sizes can be found in Figure 8.

4.2.2 Scalability experiments. We tested the scalability of the methods from an empirical perspective, namely by running each method 10 times for each of the datasets and their respective sizes. We defined the scalability of a method by taking into account two aspects, namely the ability to parse various datasets and dataset sizes, and the runtime and memory consumption measurements. Unfortunately, not all methods found in the literature could be included, either due to the lack of source code, or insufficient technical implementation. However, the ones that were able to be tested can be found in Table 4.

In [6], the authors claim that SLCT, POP, LogMine, Spell, Drain are scalable. Also, they claim that IPLoM and HLAer are potentially scalable. Thus, we were particularely interested in obtaining the results for these. However, the code for POP and HLAer was not available. Consequently, we were unable to run any experiments for these. We have to mention that most of the algorithms had problems parsing the Android dataset, which in turn made comparisons between methods impossible for this specific dataset. We attribute this to possible differences in ASCII encoding of the Android dataset. On the other hand, there were no issues running the experiments for the rest of the datasets (and comparisons were thus feasible).

Figure 6 shows a visualisation of the runtime measurements of the different methods that were able to be tested (up to 20M log lines). We observe that only four methods are capable of parsing 20M logs, namely LogCluster, LFA, IPLoM and Drain. We also observe a significant difference between these four methods and the others. Up to 4M, it could have been considered that Spell scales similarly to AEL, however the results indicate otherwise (significant difference between runtime from 4M to 10M). Figure 7 shows a visualisation of the runtime measurements of the different methods that were included. Similarly, we observe a significant difference between AEL, LogCluster, LFA, IPloM, Drain, Lenma and the rest of the methods. The figure provides an empirical estimate of runtime, that could aid practitioners in choosing specific methods appropriate for specific use-cases. Additional visualisations that also contain memory consumption can be found in Appendix B.

SLCT. Although the authors of [6] claim the method to be scalable, our empirical results indicate otherwise. Compared to other methods, its runtime is rather high (for example, parsing 20k logs for the Windows dataset requires ~25 minutes). Also, except for the HDFS dataset, it is unable to parse datasets with more than 50k logs.

AEL. AEL obtains good results and is able to parse 10M Thunderbird logs in ~80 minutes. It is also robut to parsing various datasets, thus proving to be scalable.

IPLoM. In terms of scalability, we would recommend using IPLoM as it proved to be scalable – it is able to parse 10M logs in ~30 minutes. However, in terms of memory consumption, compared to other scalable methods (LFA, LenMa, etc.) it requires more resources (Appendix B).

LogCluster. In terms of scalability, we would recommend using LogCluster as it proved to be scalable – it is able to parse 20M logs in ~35 minutes. Additionally, it proved that it can parse various datasets, thus indicating that it is robust.

LogMine. Furthermore, the method showed poor scalability, being unable to parse more than 20k logs (in ~10 minutes), regardless of the dataset used. Although the method looked promising from a design standpoint, our empirical evaluation with real-world log data lead to a different conclusion.

NuLog. In terms of scalability, the method appears to scale poorly (~35 minutes for a 50k log dataset).

LenMa. In terms of scalability, the method can parse 1M logs in roughly ~25 minutes. In terms of accuracy, for some datasets it achieves good results, and for some others the results are underwhelming.

Drain. Results for both scalability and accuracy are promising. For the former, Drain is able to parse 20M logs in ~25 minutes using ~25 GiB of memory (Section 4.2). For the latter, it is able to achieve an average accuracy of 95% over 9 different datasets (Section 4.3). We recommend Drain for use cases where log data is unlikely to change its intrinsic structure, as it is both scalable and accurate. However, if that is not the case, we do not recommend Drain, as constantly updating regular expressions can be a tedious and erroneous process.

4.3 Accuracy of log parsing approaches

In order to test accuracy, we used nine labeled log datasets provided by the authors of [12]. We use the same definition for accuracy as the one proposed in [41]. Thus, we define accuracy as the ratio of the correctly parsed log messages over the total number of log messages. For example, if a

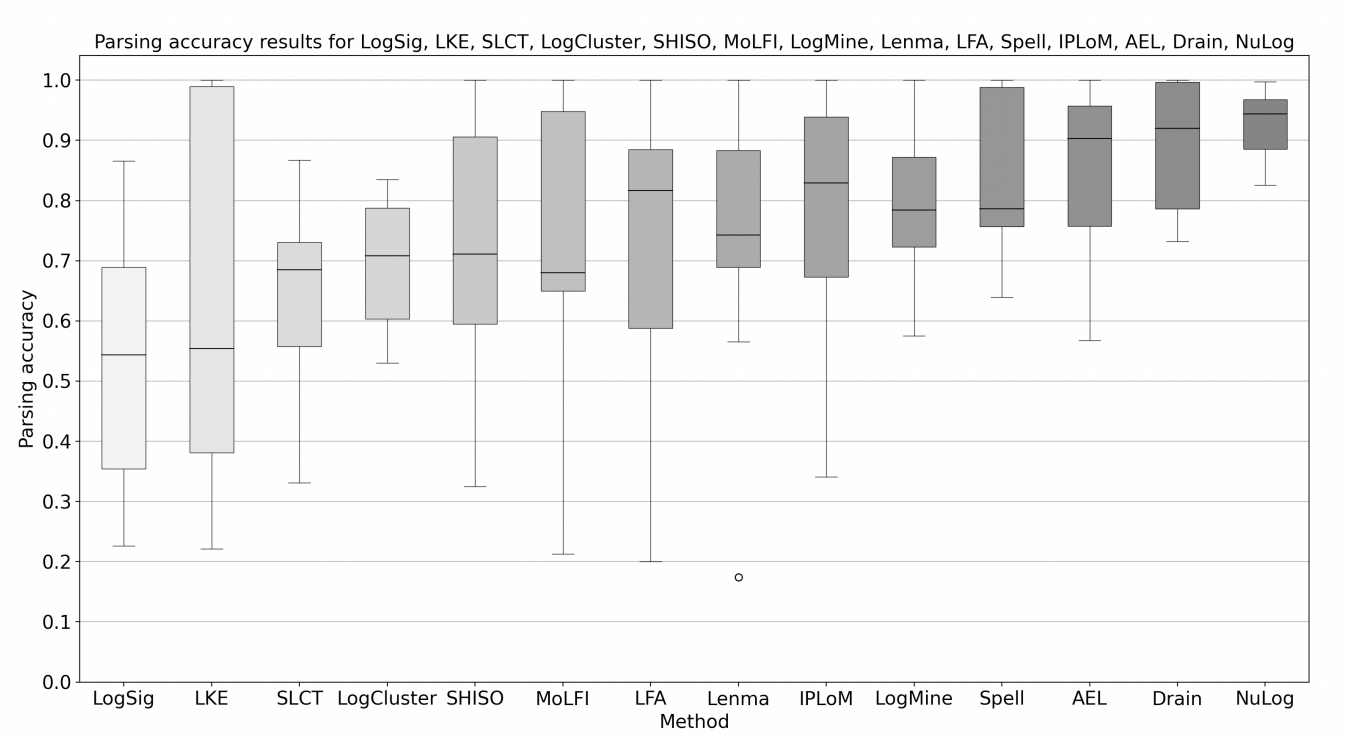

Figure 9. Parsing accuracy results for LogSig, LKE, MoLFI, SLCT, LogCluster, SHISO, Lenma, LogMine, Spell, LFA, IPLoM, AEL, Drain, NuLog.

method parses the first three log messages of a dataset L1, L2, L3, as the log templates T1, T22, T6, the accuracy can be calculated by making use of the ground truth labels for each of the log messages. Specifically, knowing that the first three log messages belong to the log templates T1, T3, T6, the parsing accuracy will be equal to $\frac{2}{3}$. This result is due to the fact that the second log line, L2, was parsed wrongly (being assigned to template T22 instead of T3). If L2 would have been parsed as template T3, the accuracy would have been equal to $\frac{3}{3} = 1$.

Practically, in order to test the methods' accuracy, we ran each method ten times for each of the nine datasets. Then, we averaged the results, obtaining an accuracy score for each method and respective dataset. These can be found in Table 3. We visualise the results in Figure 9. From left to right we arrange log parsers in ascending order based on the average[26] accuracy shown in Table 3. We obtain similar results with [24] and [41]. Specifically, we tried to reproduce Figure 25 [24, p. 70] and Figure 2 [41, p. 127]. Our results indicate that our accuracy measurements are not entirely the same. For example, in [41] IPLoM is second best after Drain, wheras in our case IPLoM is outscored by four other methods. However, we do not consider the differences significant, as they do not pose threats to the validity of the main experiment conclusions. Thus, we observe that NuLog and Drain still obtain the best results, and, in turn, LogSig and LKE still obtain the worst results. One could say that the difference between the accuracy scores of NuLog and Drain might be negligible (3% accuracy), the former achieving an accuracy score of 92%, and the latter 89%. However, we argue that NuLog would be a much better option if one wants to parse logs with high accuracy. The reason for this is that NuLog does not rely on hard-coded rules and heuristics, thus being more robust to parsing various datasets. This is proved by the fact that it does not score lower than 82%, whereas Drain achieves parsing accuracies as low as 73% (even with having hard-coded rules in place).

SLCT. For the former, its overall parsing accuracy is poor having an average of roughly 64%. However, SLCT is not intended for log parsing, but rather for finding frequent patterns in a dataset. Thus, if the use case requires finding log messages that occur repeatedly, SLCT might be a good option.

AEL. AEL's accuracy results looked promising, having an average of around 83%. However, it still heavily depends on heuristics and thus might not be the best solution for log datasets of high diversity.

[26]We chose the average accuracy as the ordering criterion, because this can also provide an estimate of the methods' robustness. Specifically, a parser is more robust if it achieves a higher average accuracy. This way, methods that have a higher average accuracy prove that are able to parse datasets that are potentially more diverse.

IPLoM. In terms of accuracy, the results were not that promising – an average parsing accuracy of roughly 78%. Thus, this makes it questionable if the method can be afor cases where

LogCluster. In terms of accuracy, the results were underwhelming – an average parsing accuracy of roughly 69%. As neither LogCluster nor SLCT were intended for log parsing, but rather for finding frequent patterns in logs, for better accuracy results, we would advise using other methods, like NuLog.

LogMine. In terms of accuracy, LogMine obtained roughly 80%. However, the method was not able to parse the OpenSSH dataset (this dataset was not incuded in the experiments). Thus, we think there are some problems in terms of its robustness to parsing different datasets.

NuLog. However, it has proved to surpass all other methods in terms of parsing accuracy results. Furthermore, it has shown to be robust to all the datasets – lowest accuracy being still above 82%. We highly recommend NuLog if the use case requires high parsing accuracy.

LenMa. In terms of accuracy, LenMa seems to be highly dependent on the particularities of specific datasets, being able to obtain both good and underwhelming results. Specifically, it is able to obtain 100% for the Apache dataset, but obtains 17% accuracy for the HealthApp dataset.

Drain. It is able to achieve an average accuracy of 89% over nine different datasets. We recommend Drain for use cases where log data is unlikely to change its intrinsic structure, as it is both scalable and accurate. However, if that is not the case, we do not recommend Drain, as constantly updating regular expressions can be a tedious and erroneous process.

Since the determination of the accuracy requires the labels to be already present in order to compare against a ground truth, this part of our study is limited to a 2k dataset. While we do not consider this to impact the generality of our finding significantly[27], we hope to produce larger labeled datasets in future work to expand this part of our study.

5 Discussion & Conclusion

In this paper we analyzed the SOTA log parsing methods. The aim of this survey was directed towards three research questions, namely mapping out SOTA log parsing approaches, evaluating the scalability and the accuracy of these methods. Regarding the first question, log parsing approaches are divided in two main categories, offline and online approaches. For offline approaches, 23 were successfully identified, whereas, for online approaches, 9 were successfully identified. We visualised the clustering of the methods based on their algorithmic approach and their connections with other related works. This way, we identify future oportunities. The answer to the second research question is that it seems like there are only a few scalable methods available namely, Drain, IPLoM, LFA, LogCluster (these can run their methods in a matter of minutes, and seem robust to parsing various log formats). However, out of the four, IPLoM consumes significantly more memory than the rest. In regards to the third research question, the methods have various accuracies, and for most, their performance is highly dependent on the specifics of a dataset. However, NuLog proved to be robust against datasets' particularities, and obtained the best parsing accuracy results (over 92% over 9 datasets).

[27] For example, for datasets sizes 25 times greater than 2k log datasets, the runtime is ~5 minutes, which is still a reasonable amount of time in order to obtain other parsing accuracy estimates.

References

[1] Amey Agrawal, Rohit Karlupia, and Rajat Gupta. 2019. Logan: A Distributed Online Log Parser. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1946–1951. https://doi.org/10.1109/ICDE.2019.00211

[2] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering Points to Identify the Clustering Structure. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data* (Philadelphia, Pennsylvania, USA) *(SIGMOD '99)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/304182.304187

[3] Hetong Dai, Heng Li, Weiyi Shang, Tse-Hsun Chen, and Che-Shao Chen. 2020. Logram: Efficient Log Parsing Using n-Gram Dictionaries. arXiv:2001.03038 [cs.SE]

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[5] Min Du and Feifei Li. 2019. Spell: Online Streaming Parsing of Large Unstructured System Logs. *IEEE Transactions on Knowledge and Data Engineering* 31, 11 (2019), 2213–2227. https://doi.org/10.1109/TKDE.2018.2875442

[6] Diana El-Masri, Fabio Petrillo, Yann-Gaël Guéhéneuc, Abdelwahab Hamou-Lhadj, and Anas Bouziane. 2020. A systematic literature review on automated log abstraction techniques. *Information and Software Technology* 122 (2020), 106276. https://doi.org/10.1016/j.infsof.2020.106276

[7] 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 Conference on Data Mining*. 149–158. https://doi.org/10.1109/ICDM.2009.60

[8] 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* (Indianapolis, Indiana, USA) *(CIKM '16)*. Association for Computing Machinery, New York, NY, USA, 1573–1582. https://doi.org/10.1145/2983323.2983358

[9] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R. Lyu. 2018. Towards Automated Log Parsing for Large-Scale Log Data Analysis. *IEEE Transactions on Dependable and Secure Computing* 15, 6 (2018), 931–944. https://doi.org/10.1109/TDSC.2017.2762673

[10] 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)*. 33–40. https://doi.org/10.1109/ICWS.2017.13

[11] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R. Lyu. 2021. A Survey on Automated Log Analysis for Reliability Engineering. *ACM Comput. Surv.* 54, 6, Article 130 (July 2021),

37 pages. https://doi.org/10.1145/3460345

[12] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. 2020. Loghub: A Large Collection of System Log Datasets towards Automated Log Analytics. arXiv:2008.06448 [cs.SE]

[13] Shaohan Huang, Yi Liu, Carol Fung, Rong He, Yining Zhao, Hailong Yang, and Zhongzhi Luan. 2020. Paddy: An Event Log Parsing Approach using Dynamic Dictionary. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. 1–8. https://doi.org/10.1109/NOMS47738.2020.9110435

[14] Zhen Ming Jiang, Ahmed E. Hassan, Parminder Flora, and Gilbert Hamann. 2008. Abstracting Execution Logs to Execution Events for Enterprise Applications (Short Paper). In *2008 The Eighth International Conference on Quality Software*. 181–186. https://doi.org/10.1109/QSIC.2008.50

[15] Staffs Keele et al. 2007. *Guidelines for performing systematic literature reviews in software engineering*. Technical Report. Citeseer.

[16] Satoru Kobayashi, Kensuke Fukuda, and Hiroshi Esaki. 2014. Towards an NLP-Based Log Template Generation Algorithm for System Log Analysis. In *Proceedings of The Ninth International Conference on Future Internet Technologies* (Tokyo, Japan) *(CFI '14)*. Association for Computing Machinery, New York, NY, USA, Article 11, 4 pages. https://doi.org/10.1145/2619287.2619290

[17] David Maier. 1978. The Complexity of Some Problems on Subsequences and Supersequences. *J. ACM* 25, 2 (apr 1978), 322–336. https://doi.org/10.1145/322063.322075

[18] Adetokunbo Makanju, A. Nur Zincir-Heywood, and Evangelos E. Milios. 2012. A Lightweight Algorithm for Message Type Extraction in System Application Logs. *IEEE Transactions on Knowledge and Data Engineering* 24, 11 (2012), 1921–1936. https://doi.org/10.1109/TKDE.2011.138

[19] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. 2018. A Search-Based Approach for Accurate Identification of Log Message Formats. In *Proceedings of the 26th Conference on Program Comprehension* (Gothenburg, Sweden) *(ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 167–177. https://doi.org/10.1145/3196321.3196340

[20] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael Rung-Tsong Lyu, and Hua Cai. 2013. Toward Fine-Grained, Unsupervised, Scalable Performance Diagnosis for Production Cloud Computing Systems. *IEEE Transactions on Parallel and Distributed Systems* 24, 6 (2013), 1245–1255. https://doi.org/10.1109/TPDS.2013.21

[21] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1301.3781

[22] Masayoshi Mizutani. 2013. Incremental Mining of System Log Format. In *2013 IEEE International Conference on Services Computing*. 595–602. https://doi.org/10.1109/SCC.2013.73

[23] 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 2010)*. 114–117. https://doi.org/10.1109/MSR.2010.5463281

[24] Sasho Nedelkoski. 2021. Deep anomaly detection in distributed software systems. (2021).

[25] Sasho Nedelkoski, Jasmin Bogatinovski, Alexander Acker, Jorge Cardoso, and Odej Kao. 2020. Self-Supervised Log Parsing. *CoRR* abs/2003.07905 (2020). arXiv:2003.07905 https://arxiv.org/abs/2003.07905

[26] Xia Ning, Geoff Jiang, Haifeng Chen, and Kenji Yoshihira. 2013. HLAer : a System for Heterogeneous Log Analysis.

[27] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. [n. d.]. Using of Jaccard Coefficient for Keywords Similarity.

[28] Issam Sedki, Abdelwahab Hamou-Lhadj, and Otmane Aït Mohamed. 2021. AWSOM-LP: An Effective Log Parsing Technique Using Pattern Recognition and Frequency Analysis. *CoRR* abs/2110.15473 (2021). arXiv:2110.15473 https://arxiv.org/abs/2110.15473

[29] Keiichi Shima. 2016. Length Matters: Clustering System Log Messages using Length of Words. arXiv:1611.03213 [cs.OH]

[30] H. Studiawan, F. Sohel, and C. Payne. 2018. Automatic log parser to support forensic analysis. In *16th Australian Digital Forensics Conference*. https://researchrepository.murdoch.edu.au/id/eprint/42841/

[31] Hudan Studiawan, Ferdous Sohel, and Christian Payne. 2020. Automatic Event Log Abstraction to Support Forensic Investigation. In *Proceedings of the Australasian Computer Science Week Multiconference* (Melbourne, VIC, Australia) *(ACSW '20)*. Association for Computing Machinery, New York, NY, USA, Article 1, 9 pages. https://doi.org/10.1145/3373017.3373018

[32] 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* (Glasgow, Scotland, UK) *(CIKM '11)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2063576.2063690

[33] Shimin Tao, Weibin Meng, Yimeng Chen, Yichen Zhu, Ying Liu, Chunning Du, Tao Han, Yongpeng Zhao, Xiangguang Wang, and Hao Yang. 2021. LogStamp: Automatic Online Log Parsing Based on Sequence Labelling. *Interface* 19, 03 (2021), 03.

[34] Stefan Thaler, Vlado Menkonvski, and Milan Petkovic. 2017. Towards a neural language model for signature extraction from forensic logs. In *2017 5th International Symposium on Digital Forensic and Security (ISDFS)*. 1–6. https://doi.org/10.1109/ISDFS.2017.7916497

[35] R. Vaarandi. 2003. 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)*. 119–126. https://doi.org/10.1109/IPOM.2003.1251233

[36] 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)*. 1–7. https://doi.org/10.1109/CNSM.2015.7367331

[37] Wikipedia contributors. 2021. Logging (software) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Logging_(software)&oldid=1042915159 [Online; accessed 22-October-2021].

[38] Wikipedia contributors. 2021. Self-supervised learning — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Self-supervised_learning&oldid=1054652741 [Online; accessed 19-January-2022].

[39] Wikipedia contributors. 2022. Conditional random field — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Conditional_random_field&oldid=1067144144. [Online; accessed 1-February-2022].

[40] Tong Xiao, Zhe Quan, Zhi-Jie Wang, Kaiqi Zhao, and Xiangke Liao. 2020. LPV: A Log Parser Based on Vectorization for Offline and Online Log Parsing. In *2020 IEEE International Conference on Data Mining (ICDM)*. 1346–1351. https://doi.org/10.1109/ICDM50108.2020.00175

[41] 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)*. 121–130. https://doi.org/10.1109/ICSE-SEIP.2019.00021

A Queries

In this section you can find the papers that were selected for the survey. Each subsection contains the number of papers selected for the respective query.

A.1 Google Scholar

These were the queries used in Google Scholar.

Query 1: "log parsing"

From the first 100 results, 27 were selected. 29 additional papers were selected during the process of backward snowballing.

Query 2: "log parsing survey"

From the first 100 results, 2 were selected. 2 additional papers were selected during the process of backward snowballing.

Query 3: "log abstraction"

From the first 100 results, 7 were selected. 2 additional papers were selected during the process of backward snowballing.

Query 4: "log abstraction survey"

From the first 100 results, 0 were selected.

Query 5: "event log parsing"

From the first 100 results, 6 were selected. 1 additional paper was selected during the process of backward snowballing.

Query 6: "log signature extraction"

From the first 100 results, 6 were selected. 0 additional papers were selected during the process of backward snowballing.

Query 7: "event log signature extraction"

From the first 100 results, 0 were selected.

A.2 Scopus

This query was used in Scopus.

Query 1: "TITLE-ABS-KEY(log AND parsing) OR ((logs OR log OR logging OR events OR "event log" OR "event logs" OR "event logs templates" OR "event log signatures") AND (abstractionOR parsing))"

From all 392 results, 13 were selected. 0 additional papers were selected during the process of backward snowballing.

B Experiments visualisations

This section contains visualisations of runtime and memory consumption measurements for IPLoM, AEL, Lenma, MoLFI, SHISO, LogCluster, LogSig, and LKE.

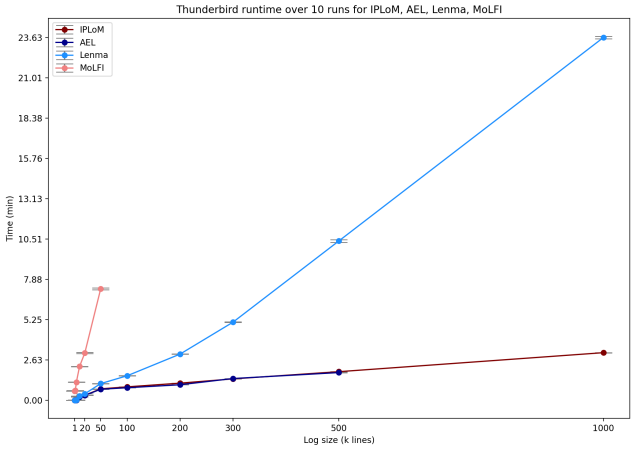

Figure 10. Runtime measurements using the Windows dataset for IPLoM, AEL, Lenma, and MoLFI.

In Figure 10 we can observe that there is a significant difference between IPLoM & AEL and Lenma. Compared to the first two that parse the dataset in about 1.5 minutes (for 500k log lines), Lenma parses 500k in 10 minutes. Also, in Figure 11 we notice a significant difference between the aforementioned methods and MoLFI. The first three methods parse 50k logs in roughly 1 minute, compared to MoLFI which takes 8.

We plot the same measurements for the Windows dataset.

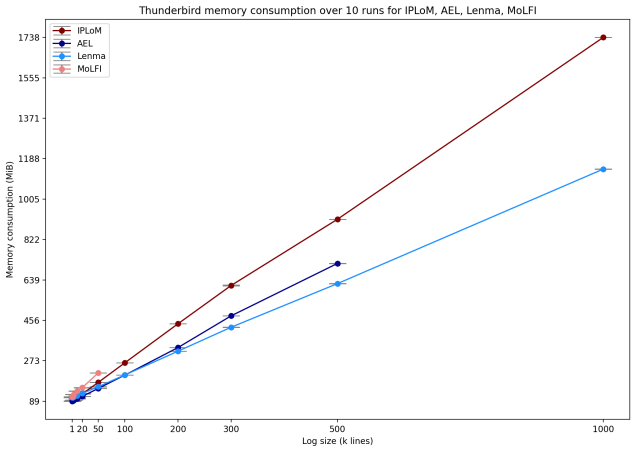

Figure 11. Memory consumption measurements using the Thunderbird dataset for IPLoM, AEL, Lenma, and MoLFI.

These can be found in Figure 12 and Figure 13 respectively.

It can be see that the methods perform similarly to the Thunderbird dataset. However, we notice some differences. For the Windows dataset, the runtime of Lenma reduces significantly compared to the previous scenario. Also. MoLFI runtime seems to increase, compared to the experiments performed on Thunderbird.

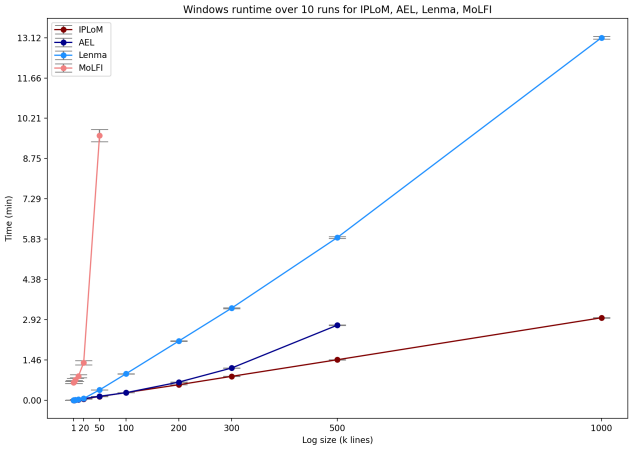

Figure 12. Runtime measurements using the Windows dataset for IPLoM, AEL, Lenma, and MoLFI.

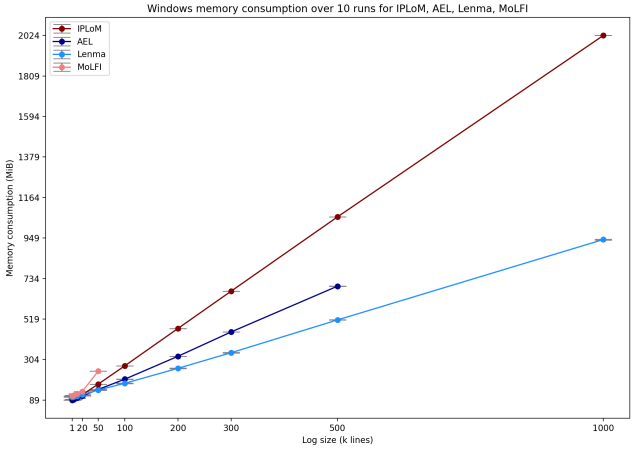

Figure 13. Memory consumption measurements using the Windows dataset for IPLoM, AEL, Lenma, and MoLFI.

Table 4. Scalability experiments successfully run for each dataset and method.

Dataset	AEL	Drain	IPLoM	Lenma	LFA	LKE	LogCluster	LogMine	LogSig	MoLFI	NuLog	SHISO	SLCT	Spell
BGL 1k	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓
BGL 2k	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓
BGL 4k	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓
BGL 10k	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓
BGL 20k	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓
BGL 50k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓	✓
BGL 100k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✓	✓	✗	✓
BGL 200k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✓	✓	✗	✓
BGL 300k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✓
HDFS 1k	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
HDFS 2k	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
HDFS 4k	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
HDFS 10k	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
HDFS 20k	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓
HDFS 50k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓	✓
HDFS 100k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓	✓
HDFS 200k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✓	✓
HDFS 300k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✓	✓
HDFS 500k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✓	✓
HDFS 1M	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✓	✓
OpenSSH 1k	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓
OpenSSH 2k	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓
OpenSSH 4k	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓
OpenSSH 10k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✗	✓	✓	✓
OpenSSH 20k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✗	✓	✓	✓
OpenSSH 50k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✗	✓	✓	✓
OpenSSH 100k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗	✓	✓	✓
OpenSSH 200k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗	✓	✓	✓
OpenSSH 300k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✓
OpenSSH 500k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✓
Thunderbird 1k	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓
Thunderbird 2k	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓
Thunderbird 4k	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓
Thunderbird 10k	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓
Thunderbird 20k	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	✓
Thunderbird 50k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓
Thunderbird 100k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗	✓	✗	✓
Thunderbird 200k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗	✓	✗	✓
Thunderbird 300k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✓
Thunderbird 500k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✓
Thunderbird 1M	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✓
Windows 1k	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Windows 2k	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Windows 4k	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Windows 10k	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Windows 20k	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓
Windows 50k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✗	✗
Windows 100k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✓	✓	✗	✗
Windows 200k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗	✓	✗	✗
Windows 300k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✗
Windows 500k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✗
Windows 1M	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✗

Table 4. Scalability experiments successfully run for each dataset and method. (Continued)

Dataset	AEL	Drain	IPLoM	Lenma	LFA	LKE	LogCluster	LogMine	LogSig	MoLFI	NuLog	SHISO	SLCT	Spell
Thunderbird 2M	✓	✓	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓
Thunderbird 4M	✓	✓	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓
Thunderbird 20M	✗	✓	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
Thunderbird 50M	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Thunderbird 100M	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

Table 5. Parsing accuracy experiments able to be run on various datasets by the selected methods.

Dataset	AEL	Drain	IPLoM	Lenma	LFA	LKE	LogCluster	LogMine	LogSig	MoLFI	NuLog	SHISO	SLCT	Spell
BGL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
OpenStack	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
HDFS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Apache	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
HPC	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Windows	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
HealthApp	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Mac	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Spark	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓