Rethinking log parsing in the context of modern software ecosystems

Master’s Thesis

Ștefan Petrescu

TU Delft

Rethinking log parsing in the context of modern software ecosystems

by

Ștefan Petrescu

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

AT THE
DELFT UNIVERSITY OF TECHNOLOGY,
TO BE DEFENDED PUBLICLY ON WEDNESDAY JULY 20, 2022 AT 10:00 AM.

Student number:	5352150	
Thesis committee:	Prof. dr. J.S. Rellermeyer,	LUH and TU Delft, supervisor
	Associate Prof. dr. L. Chen,	TU Delft
	Assistant Prof. dr. L.M. da Cruz,	TU Delft

July, 2022

Abstract

Modern systems generate a tremendous amount of data, making manual investigations infeasible, hence requiring automating the process of analysis. However, running automated log analysis pipelines is far from straightforward, due to the changing nature of software ecosystems caused by the constant need to adapt to user requirements. In practice, these are comprised of a series of steps that collectively aim at turning raw logs into actionable insights. The first step is log parsing which aims to abstract away from raw logs toward structured information. Log parsing is paramount, as it influences the performance of all subsequent downstream tasks that rely on its output. Although previous works have investigated the performance of log parsing, given the increase in data heterogeneity witnessed over the past decades, the validity of current estimates is questionable, as there is a lack of understanding of how log parsing methods perform in modern contexts. Consequently, we investigate the field and, in the process, we discover that misleading metrics are adopted, which produce incomplete performance estimates. Furthermore, motivated by an industry use case within the infrastructure of a large international financial institution, we discover that the current log parsing paradigm is not aligned with what is required in practice. Consequently, to address these current limitations, in this work we contribute with the following. We (1) evaluate the field of log parsing, (2) propose a new log parsing paradigm and create a benchmark dataset to facilitate future research, and (3) propose and evaluate a machine learning model that solves log parsing within the new paradigm.

Acknowledgments

This thesis marks the end of two years of studying at TU Delft, which has been an invaluable experience, both from a personal and academic perspective. I have met smart, talented, and hard-working people who pushed me to better myself and to see things from a different perspective. You have made this experience truly unique, and I am deeply indebted to you.

First and foremost, I would like to thank my supervisor, Jan Rellermeyer. You have inspired me to always aim higher, and I could not have undertaken this project without your guidance. I would also like to thank you for your constant support and fruitful discussions throughout this year. Secondly, I would like to express my deepest appreciation to Floris den Hengst, my daily supervisor. You have helped me tremendously throughout this project, and I can say that your advice and ideas truly reshaped the way I think. Thirdly, I would like to extend my sincere thanks to the AIOps I3 team at ING, and to the people at the AI for Fintech Research lab for helping me shape my research ideas and for providing an amazing working environment. Specifically, I would like to thank Evert-Jan van Doorn, Pinar Kahraman, and Eileen Kapel for their help.

Lastly, I would like to thank my family and friends. I would like to thank my sister, for being my best friend and for always believing in me. Finally, I want to dedicate this to my mother and father especially. Words cannot express my gratitude for the unconditional love and support you have shown me through my journey.

Ștefan Petrescu
Delft, July 2022

Contents

1 Introduction

Modern software systems have become vital to society, contributing toward improving the quality of our lives, and enabling services that provide unprecedented benefits. These range from systems that keep our infrastructure intact (e.g. railway systems software), to systems intended for entertainment purposes (e.g. social media platforms). For example, payment systems facilitate financial transactions and bring invaluable economic benefits, sustaining a historically unprecedented paradigm for transferring monetary value. However, payment systems involve financial institutions, instruments, technologies, etc. [47] which subsume a huge amount of software components, creating a highly complex ecosystem. Thus, to benefit from the wonders of modern software systems, it is paramount to tackle the challenges associated with their ever-increasing complexity and data volume.

As the world becomes progressively more digital, the need to enable more and more services brings forth new challenges, requiring designing solutions that match up market's needs. Consequently, modern systems are designed to move away from large-scale applications standardized on single technologies [5] (e.g., monolithic application) to flexible and easy to evolve architectures [20] (e.g., microservices). However, this implies adopting distributed architectures which are incomparably harder to maintain, making software systems' complexity increase drastically. Furthermore, as a consequence of the vast number of software components subsumed by these systems, the volume of data that they generate is unparalleled, reaching terabytes of log data daily [21]. Thus, automated data analysis processes are required, as it is infeasible to analyze this volume of information manually.

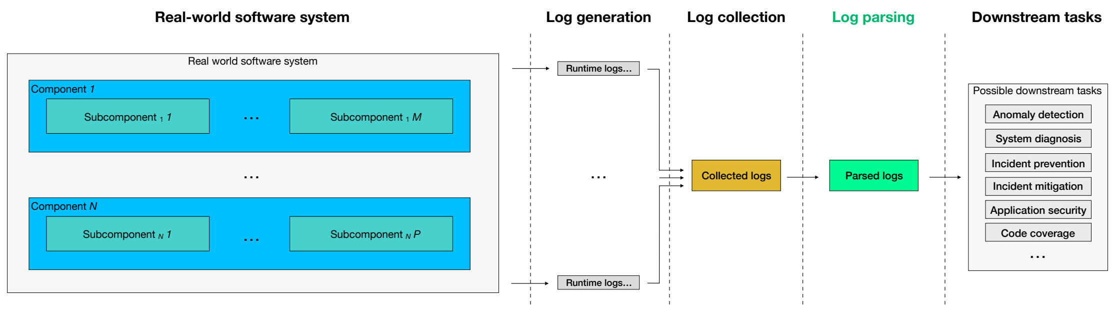


Figure 1.1: Example of a log cycle in practice. As we illustrate, the parsing step (in green) plays a crucial role in the pipeline, consuming aggregated information from the entire software stack (in blue) and fueling a variety of downstream tasks that have the goal of discovering critical insights about the behavior of the running systems.

Implementing processes that enable the ever-increasing market's needs implies automating the analysis of the data generated by systems, and for that logs are by far the most valuable resource. Specifically, these provide the highest level of monitoring granularity, recording the entire execution of the system. Thus, valuable information can be extracted from logs, and used for various purposes, such as risk management, error analysis, incident prediction, incident mitigation, anomaly detection, etc. [52]. For example, logs can be leveraged to predict incidents within the infrastructure of large financial institutions, as a result of being distilled by automated processes that generate early warnings for engineers [51]. Thus, for maintaining services' quality while keeping competitive edges, it is essential to leverage the logs that they generate, by automating the processes of log analysis.

Automated log analysis processes are usually split into sequences of steps which collectively aim at distilling actionable insights from raw logs. These sequences receive as input data at the lowest level of abstraction, namely raw logs, and can provide various types of output depending on the particular problem, such as regression, classification, etc. For example, an automated log analysis process may have as input raw logs that contain details about users' authentication, and output alerts if anomalies are detected. However, implementing such sequences is challenging, as these have to handle the tremendous complexity of modern systems, providing solutions that are both accurate and scalable. Furthermore, as a consequence of the need to adapt to user requirements, systems are constantly changing, making the problem of automating log analysis processes hard to constrain and solve.

In the literature, the first step in automated log analysis pipelines is known as *log parsing*, which aims to transform raw logs into structured information. Specifically, it abstracts away from raw logs toward a representation that can be used by other downstream processes. We visualise its role in Figure 1.1. As it influences the performance of all subsequent tasks, its importance is paramount and it has been categorised as one of the main five trends in the *logging* field [10]. Log parsing separates the constant part from the variable part of log messages. The former refers to the static parts in log messages, which correspond to the textual logging statements written by developers in the software code, whereas the latter refers to the dynamic information generated at runtime. Even though the problem may sound straightforward, the literature has yet to produce clear winners, as log parsing has actually proved to be a hard problem to solve in practice.

As a consequence of the increase in complexity witnessed over the past decades, log data heterogeneity makes automating the process of log analysis increasingly harder. Specifically, more and more diverse data are centralized, in turn making log parsing increasingly more difficult to solve. These data are diverse in the sense that they are generated by a vast number of software components subsumed by systems. It is desirable to centralize log collection, to ensure that the causality between the various software components of systems can still be identified during and after execution. However, as a consequence of the resulting heterogeneous log data, robust and scalable solutions are required for analysis, which are difficult to implement.

Currently, the field lacks methods that are able to provide robust and scalable solutions for log parsing, and it may even be that the existing ones are not aligned with what is actually required in practice. Motivated by an industry use-case within a large financial institution, we discover that applying log parsing methods in a highly diverse software ecosystem proves to be harder than expected, and we find that there is a lack of understanding on how the log parsing methods proposed in the literature perform in real-world scenarios. Moreover, we discover that the field lacks alignment with industry goals, as in practice it is desirable to have particular representations for the structured output of the log parsing process, which is not the case within the current log parsing paradigm. Thus, to address these current limitations, in the following section we present the research questions that we aim to answer in this work.

1.1 Research questions

Currently, there is no clear understanding of how log parsing methods perform in industry contexts; and how methods can be evaluated, to best align with industry goals. Furthermore, the current log parsing paradigm might have some limitations, which might make applying it impractical. Thus, to address the current research gap, in this work, we aim to answer the following research questions:

RQ1: How has log parsing been approached in literature?

RQ2: What is the goal of log parsing and how is that evaluated in literature?

RQ3: How do log parsing methods perform in the context of modern software ecosystems?

RQ4: How is the log parsing paradigm aligned with industry goals, and how can that be improved?

To address these research questions, we test log parsing methods on (1) publicly available data, (2) on data that emulates modern-systems log data, and (3) on logs generated within the infrastructure of a large international financial institution. Our findings indicate that applying log parsing in practice is impractical, due to the subpar performance. Additionally, the results indicate that the current log parsing paradigm is incomplete, as it is not fully aligned with what is actually required in practice. Thus, we study the limitations of the current log parsing paradigm and propose an alternative option, that aims to address the incompleteness of the former. Furthermore, we propose a method able to operate in this paradigm, and for this, we investigate three machine learning models and choose the one that obtains the best performance, in terms of the accuracy-scalability trade-off. Lastly, we propose a benchmark dataset for evaluating log parsing methods within the new paradigm, comprised of `('runtime log', 'ground truth logging statement')` entries.

1.2 Organization

Towards answering **RQ1**, in Chapter 2 we study related work and different log parsing methods proposed in the literature.

In Chapter 3 we answer research questions **RQ2** and **RQ3** by analyzing how log parsing methods are evaluated in the literature and by subsequently evaluating their performance in the context of modern systems.

Finally, In Chapter 4 we answer **RQ4**, by discussing the current limitations of the current log parsing paradigm and presenting ways in which these can be addressed. Subsequently, we propose and evaluate a method that aims to solve the paradigm resulted from the answer of **RQ4**.

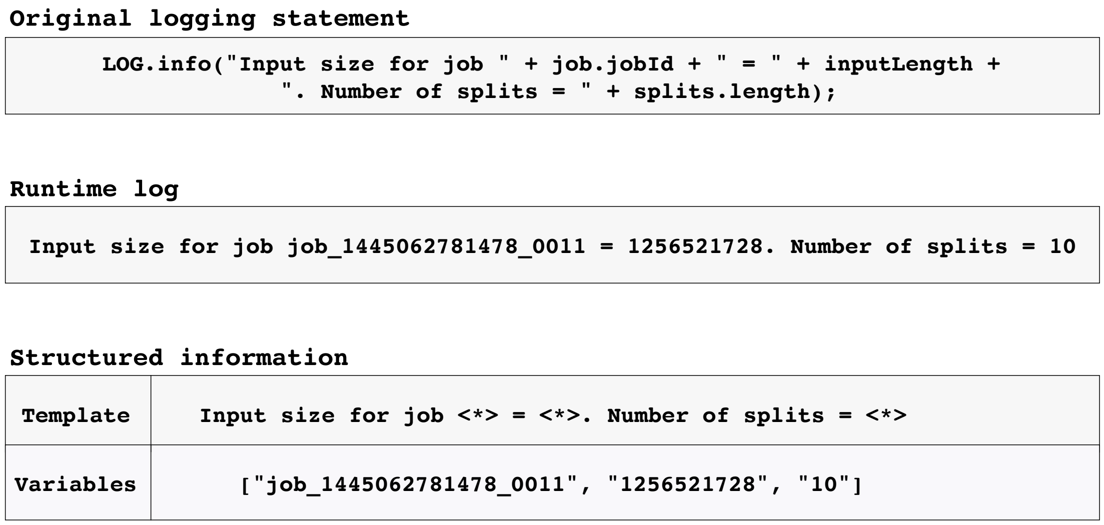
2 Background & Related work

Logs record runtime information of software systems for monitoring, auditing, and diagnosing problems [45]. For instance, system administrators primarily rely on logs to conduct investigations in an effort to understand and mitigate incidents. Additionally, in case of errors or anomalies, log messages can facilitate the process of generating alerts [52] or conducting root cause analyses [50].

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 unused even though critical information could be distilled from the log content. For example, some systems can produce 30-50 Gigabytes of logs per hour [26], making it infeasible for humans to manually investigate all messages in reasonable time. Consequently, automated log analysis techniques are necessary, as they extract information on software systems from log data automatically [14].

To draw useful conclusions from large amounts of log data using automated log analysis, a sequence of steps is required which collectively aim at translating the raw information in actionable insights. The first step is to abstract away from raw log data to obtain meaningful representations, as subsequent steps expect data to be in a particular structure. To do that, most techniques require a basic syntax-derived exploration and interpretation of the logs [32], known in the literature as *log parsing*[*].

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



Original logging statement

```
LOG.info("Input size for job " + job.jobId + " = " + inputLength +
        ". Number of splits = " + splits.length);
```

Runtime log

```
Input size for job job_1445062781478_0011 = 1256521728. Number of splits = 10
```

Structured information

Template	Input size for job <*> = <*>. Number of splits = <*>
Variables	["job_1445062781478_0011", "1256521728", "10"]

Figure 2.1: Example of how log parsing processes runtime logs, resulting in structured information that contains the extracted log template and the list of variables discovered.

Log parsing is of utmost importance because it transforms the incoming logs into structured information, which in turn can directly influence the performance of automated log analysis techniques [12]. Given its significance, it has been classified as one of the main five trends in the *logging* field [10], providing input for a wide range of automated tasks, such as anomaly detection [15], application security [32], root-cause analysis for failure diagnosis [52], etc.

Log parsing structures raw log data into (1) the underlying log templates corresponding to the static part of the logging statements in the software, (2) their respective parameters corresponding to the dynamic part of the logging statements, and (3) log meta information. Log meta information is usually added by a logging framework [14] and thus relatively easy to obtain. Consequently, the main challenge of log parsing is to discover the log templates and parameters, to obtain a better representation of the input data. An example of log parsing is shown in Figure 2.1. Thus, after the constant and variable parts of a raw log message are discovered, these combined represent the log template, where variables are replaced by generic tokens, namely <*>. This type of transformation is challenging, especially if the number of log messages is large and accuracy needs to be high, and tackling this optimally remains an open issue.

Whilst there is a limited review of log parsing in general, recent work has been published [52] [14] [10] [8], that created in-depth overviews of current approaches. These outlined methods' advantages and limitations, with respect to aspects, such as robustness, efficiency, and scalability. Specifically, Zhu et al. [52] evaluated log parsing methods to provide a basis for future research of the field, He et al. [14] included log parsing as a chapter in their survey on automated log analysis in reliability engineering, and Gholamian et al. [10] added a log parsing chapter in their survey on logging in software. To the best of our knowledge, the work by El-Masri et al. [8] is the only one that focused specifically on log parsing methods.

To build the foundation for understanding how log parsing performs in industry contexts, in this chapter we answer the following research question:

RQ1: How has log parsing been approached in literature?

In the following section we answer the research question by (1) mapping out log parsing methods found in the literature, and (2) visualizing how log parsing methods cluster based on their algorithmic approach.

2.1 How has log parsing been tackled?

Inspired by similar works [8] [14], 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 discover templates given a static set of log messages. They require a training phase, during which the templates are discovered. After this, they parse incoming logs by matching with the templates found during training in either batch or stream [8]. As changes/updates in software can introduce new log templates, one drawback of offline approaches is that it requires the training phase to be re-run periodically.

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, these (approaches) discover event templates without an offline training phase. Furthermore, as event templates are being updated dynamically, such methods can be integrated seamlessly for downstream tasks [14]. Online parsers are recommended when the decision time is relatively 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 2.1 contains a list of the selected offline log parsing approaches, whereas Table 2.2 contains a list of the selected online approaches. We excluded approaches for which the authors explicitly mentioned that their method was preliminary (we only considered work that was feature complete), and did not consider approaches that had their technical description insufficient to reproduce. In the following paragraphs, we continue by describing the approaches found in literature, and by visualizing how these methods cluster together.

Table 2.1: Overview of offline log parsing approaches found in the literature.

Year	Method used
2003	SLCT [40]
2008	AEL [18]
2009	LKE [9]
2010	LFA [29]
2011	LogSig [37]
2012	IPLoM [24]
2013	HLAer [31]
2014	NLP-LTG [19]
2015	LogCluster [41]
2016	LogMine [11]
2017	NLM-FSE [39]
2017	POP [12]
2018	MoLFI [25]
2020	LPV [49]
2020	NuLog [30]
2020	ELA [36]
2021	AWSOM-LP [33]
2021	LogStamp [38]

Table 2.2: Overview of online log parsing approaches found in the literature.

Year	Paper
2013	SHISO [28]
2016	LenMa [34]
2017	Drain [13]
2019	Spell [7]
2019	Logan [1]
2020	Logram [3]
2020	Paddy [16]

SLCT (Simple Logfile Clustering Tool) [40] uses a clustering algorithm to identify log templates. A frequent pattern mining algorithm is applied, consisting of three steps. During the first step, a table of frequent words is constructed. Whether a word occurs frequently or not is determined by a user-specified threshold. In the second step, cluster candidates are formed. More specifically, log lines with at least one frequent word are added to a candidate table. In the final step, the clusters with a support higher than a user-defined threshold are returned as log templates.

AEL (Abstracting Execution Logs) [18] proposes a rule-based approach that consists of three steps. The first step uses data-specific heuristics in order to replace the dynamic parts of a log message with generic tokens[†]. All "word=value" pairs present in log messages are regarded 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 generic tokens in a log line. The third step iterates through all of the previously created groups and returns the log templates. More specifically, by using an unspecified 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 be apart of the same template. One of *AEL*'s biggest disadvantages is that, if the first step cannot be followed, the method cannot be used, as the similarity comparisons cannot be made anymore.

LKE (Log Key Extraction) [9] proposes a three-step clustering approach. In the first step, parameters are removed by leveraging domain specific knowledge (for example, IP addresses, etc.). During the second step, log messages are clustered based on a weighted edit string distance. During the third step, the clusters are refined by means of additional heuristics. For example, authors consider two logs to have a different underlying template if the frequency of a word at a specific position is lower than a certain threshold q. The biggest disadvantage of this method is that it involves hand-crafted rules, such as regular expressions and user-defined thresholds.

LFA (Log File Abstraction) [29] proposes a two-step algorithm. During the first step, a word frequency table is built which registers words' frequency. Compared to the first step of *SLCT*, *LFA* also considers words' positions within a log line. During the second step, each log line is analyzed. Specifically, for each log message, each word is compared to its frequency at its respective position. Lastly, log templates are returned.

LogSig [37] proposes a clustering algorithm, consisting of three main steps. It first splits log lines into pairs of tokens[‡] by extracting all pairs of terms in the log messages. During its second stage, the pairs of tokens are partitioned into k groups, k being a support value parameter. The third step involves creating the log templates from the resulting groups, and returning them.

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

[‡]Any sequence of characters separated by whitespace 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.

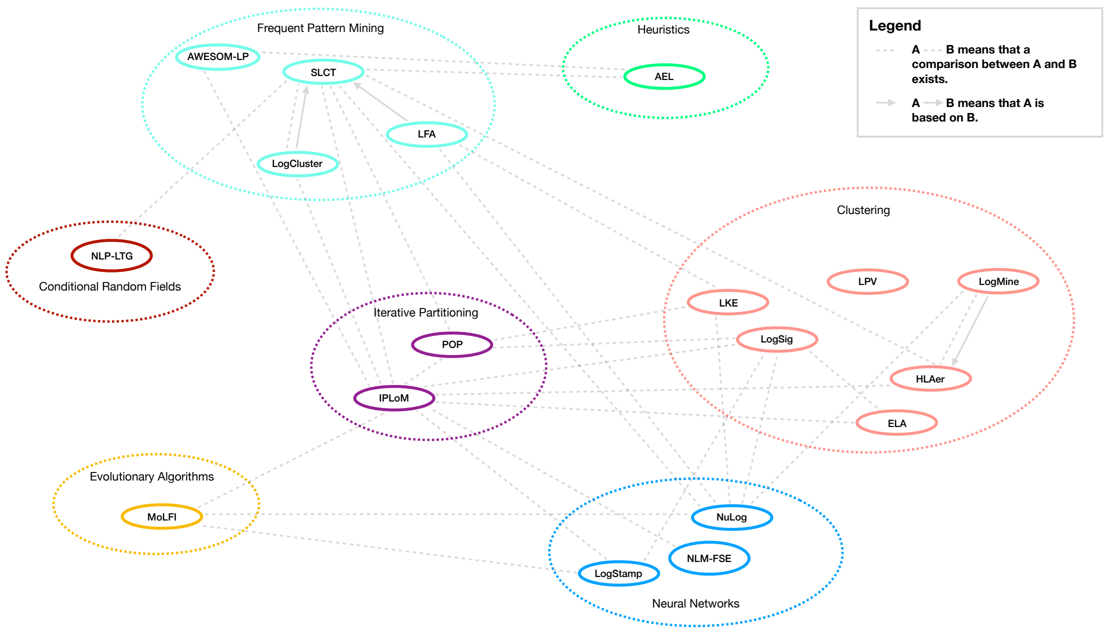


Figure 2.2: Graphical representation of how the offline methods are clustered together based on their algorithmic approach. The dotted lines represent connections between various methods. Specifically, a dotted line means that a comparison exist between two connected methods.

IPLoM (Iterative Partitioning Log Mining) [24] tackles log parsing with a four-step hierarchical partitioning algorithm. During the first step, log lines are partitioned by their token 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 message that has a length of n tokens, the column with the least variability is most likely to contain the template. The third step partitions the messages 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) [31] proposes a three-step log parsing approach. During its first step, 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*[§] [2] algorithm. During the third step, for each of the previously found

[§]Although the *OPTICS* algorithm has an $O(n^2)$ memory complexity [8], the authors *HLAer* still claim the method to be scalable as it can be parallelized.

clusters, a pattern recognition algorithm is applied such that event templates are found and returned.

NLP-LTG (Natural Language Processing–Log Template Generation) [19] proposes an approach based on a statistical modelling technique, namely Conditional random field [42] (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 [41] is designed to overcome the shortcomings of *SLCT* and *IPLoM*, and introduces a three-step frequent pattern mining algorithm. In contrast to *SLCT* and *IPLoM*, *LogCluster* does not take words' positions into account. During the first step, it discovers frequent words in the log dataset. During the second step, it generates a set of candidate clusters. During the 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 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 Link End 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 "`Link End`" 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 [11] introduces a log parsing approach based on clustering, consisting of four steps. The first step involves tokenization and type detection (for example, a type can be a date, timestamp, IP address, number etc.). *LogMine* replaces detected variables, such as numbers or dates with their type. For example, after the first step, the log message "`session opened for user anna uid = 10`" becomes "`session opened for user name uid = number`". The second step involves a clustering of the logs, 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 the fourth step, the authors use an algorithm that generates a hierarchy of patterns, and returns the log templates.

NLM-FSE (Neural Language Model-For Signature Extraction) [39] proposes an approach that leverages the use of neural networks. Compared to all the other methods presented in this work, 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 [12] 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 might 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. For example, two messages are considered to belong to the same log template, 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, and otherwise 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 this is that it might be that some of the groups to 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* [23]. *POP*'s main advantage is that it is very fast, allowing for parallel computations. However, it relies on heuristics and its accuracy results are not surpassing other approaches.

MoLFI [25] proposes a seven-step evolutionary algorithm for log parsing. During the first step, preprocessing operations are followed, namely regular expressions are used to remove variables based on domain knowledge. Specifically, the respective variables are replaced with `#spec#`, duplicates are removed, log messages are transformed into sequences of tokens (using various separators), and messages are grouped into groups based on the number of tokens (messages with the same number of tokens are grouped into the same clusters). During the second step, an encoding schema is proposed for transforming the preprocessed logs. The third step generates the *initial population* using a specific algorithm. During the fourth step, *crossover* is implemented. During the fifth step, the authors implement the *mutation* operation. The sixth and seventh step are responsible for post-processing and choosing a Pareto optimal solution. Lastly, log templates are returned.

LPV [49], uses a four-step algorithm that leverages vector embeddings to discover log templates. During the first step, preprocessing operations are followed. Specifically, duplicates[¶] are removed and common variables are substituted (for example, all IP addresses

[¶] For example, "`2005-06-03-15.42.50 instruction cache parity error corrected`" and "`2005-06-03-15.42.53 instruction cache parity error corrected`" are considered to be duplicates, even though they have different meta-level information (i.e., different timestamps).

are substituted with a special token, `$$IPADDR$$`). During the second step, the previously substituted log messages are embedded into vectors, using *word2vec* [27]. 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 a representation[‖]. During the third stage the log representations are clustered, based on their semantic distance. During the fourth and final step, log templates are returned.

NuLog (Neural Log) [30] introduces an approach that tackles log parsing as a self-supervised learning [48] 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 for parsing logs. Training the model consists of two steps, namely tokenization and masking. For tokenization, each log message is transformed into a sequence of tokens. For masking, the authors use a method from natural language processing called *Masked Language Modelling*. Specifically, a random token is replaced by the special `<MASK>` token for each sequence of tokens (log message). Then, each sequence is padded with two delimiter tokens. Finally, after training, the model returns log templates.

ELA (Event Log Abstraction) [36] uses a five-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* [35] 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 preprocessed messages[**]. During its second step, the algorithm groups logs based on the word count – the authors assume that logs that contain the same number of words are likely to belong to the same underlying log template. As the third step, the authors construct a graph model using the count-based 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 [33] proposes an approach that tackles log parsing by using a frequency analysis technique. Specifically, it is composed of three steps. During its first step, 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 step, frequency analysis is applied in order to distinguish between con-

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

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

stants and variables, which 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). Lastly, the log templates are returned.

LogStamp [38] 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[††], 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 for extracting log templates.

SHISO [28] uses a two-step clustering approach to discover log templates. During the first step, namely the *Search Phase*, log messages are split into a sequence of tokens based on heuristics, with the intention of discovering log formats. During the second step, namely the *Adjustment Phase*, existing log templates are refined, by means of applying a specific algorithm. Specifically, *SHISO* uses a tree-form structure to guide the parsing process, where each node is correlated with a log group and an log template. The number of children nodes in all the layers is the same and is manually configured beforehand. *SHISO*'s biggest disadvantages are its poor accuracy and poor efficiency, resulting from sensitivity to path explosion [14].

LenMa [34] proposes a five-step log parsing method based on a clustering algorithm. Specifically, the clustering algorithm uses a similarity metric based on the length of the words in a log message. For each log message, it creates 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 clusters are updated as newly arrived messages come in, and returned.

Drain [13] proposed a five-step approach that relies heavily on heuristics. During its first step, logs are preprocessed by leveraging domain knowledge – here, users have to pro-

[††]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.

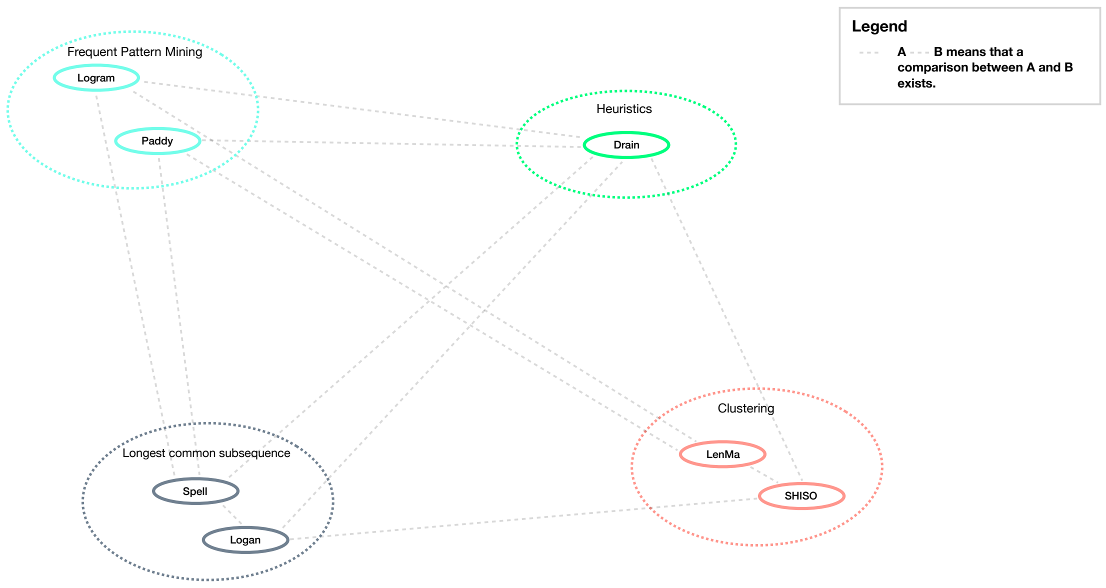


Figure 2.3: Graphical representation of how the online methods are clustered together based on their algorithmic approach. The arrows represent connections between various methods. More specifically, an arrow means that the method from which it originates, evaluated against the method towards which it points.

vide 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 example, log messages with the same number of tokens are more likely to have the same log template, or the first token of a log message is always constant and apart of the log template, etc. During its fifth step, logs' tokens are compared using a similarity metric, and lastly, the log templates are returned, and the parse-tree is updated. *Drain* can achieve high accuracy for various datasets, and it is recommended for use-cases where log 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.

Spell [7] introduces an approach based on a longest common subsequence (LCS) algorithm. During the first step, logs are parsed into a sequence of tokens using a pre-defined set of delimiters. During the second step, the LCS of the incoming log is discovered. During the third step, a new LCS map is created. This method is similar to *SHISO* and *LenMa*, as it maintains a list of log groups. To accelerate the parsing process, *Spell* uses specialized data structures, namely prefix tree and inverted index. In addition, *Spell* provided a parallel implementation [14].

Paddy (Parsing Approach with Dynamic Dictionary) [16] is a four-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. Specifically, the similarity metric is a weighted sum (coefficients as hyper-parameters) of *Similarity* (Jaccard similarity [43]) and *LengthFeature* (the length of a log message in terms of number of tokens). During the fourth and final step, the log templates are returned and the inverted index is updated.

RQ1. We discover that in the literature there are currently 25 methods that propose solving log parsing. We observe that these can be categorised in two main branches, online and offline methods. Lastly, we study trends in the log parsing literature, by outlining the most preferred underlying algorithms for the methods, namely clustering and frequent pattern mining.

2.2 Discussion

Automated log analysis subsumes a sequence of steps which collectively aim at drawing actionable insights from large volumes of log data. One of the most important steps in this sequence is *log parsing*. We investigate this field, and discover 25 log parsing methods, which are categorised in two main branches, *online* and *offline* methods. Subsequently, we observe trends in literature in terms of how log parsing has been approached, and we discover that most used underlying algorithms for the methods are *clustering* and *frequent pattern mining*. To summarize, in this chapter we have contributed by studying how log parsing has been approached so far in literature, by mapping out proposed methods, and by creating visualisations of methods' clustering based on their algorithmic approach.

3
Log parsing in the context of modern software ecosystems

The trend of moving away from large-scale applications standardized on single technologies [5] (e.g., monolithic application) to flexible and easy to evolve architectures [20] (e.g., microservices) provided countless benefits [6], but also increased software complexity, as it introduced challenges associated with distributed architectures which subsume incomparably more components. As a consequence of (1) the increase in size and complexity of systems, and (2) because of the diversity of all subsumed infrastructure resources, the logs generated by these systems have become increasingly more *heterogeneous*, which in turn poses different challenges for applying log parsing and, subsequently, for automating the log analysis process. These systems are expected to become more complex in the future [17], thus making log parsing progressively harder.

We argue that log parsing methods have failed to adapt to this increased complexity over the past decade. For instance, most methods evaluate on the same datasets, some of which are even comprised of 15 year-old data which no longer reflect recent real-world systems. Thus, it is paramount to test how log parsing methods perform in the context of modern software ecosystems, as their current performance estimates might not hold in practice.

In this chapter, we close this gap by evaluating the performance of log parsers in the context of modern software ecosystems as in the log parsing field, the lack of evaluation on recent real-world systems, and the use of relatively old datasets for benchmarking make performance estimates for current deployment questionable. To address this, we test the 14 most-recognized log parsing methods (1) on publicly available data, (2) on data that emulates modern-systems log data, and (3) on industry data generated within the infrastructure of a large international financial institution, and we investigate the performance in each scenario. Specifically, we are tackling the following research questions:

RQ2 What is the goal of log parsing and how is that evaluated in literature?

RQ2.1 How can log parsers be evaluated to best align with the goal of log parsing?

RQ2.2 What is the performance of log parsers on publicly available data?

RQ3: How do log parsing methods perform in the context of modern software ecosystems?

To facilitate future works and reproducibility, we contribute code and a novel public benchmark dataset[*]. In the following sections, we answer the research questions.

3.1 Log parsing evaluations in literature

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

Zhu et al. [52] evaluate 13 representative log parsers. They analyze the methods with regards to three aspects, *Accuracy*, *Robustness*, and *Efficiency*. To do so, the authors run experiments on 16 different homogeneous datasets between 17 and five years old. They make the code and labeled datasets publicly available, with the aim of providing a basis for further developments. While this marked an important step towards more robust and reproducible insights about log parser performance, unfortunately the authors focused their research entirely on isolated logs from single applications.

He et al. [14] survey automated log analysis methods in the area of reliability engineering. Their contribution is to classify the methods based on four aspects, namely *mode*, *coverage*, *preprocessing*, *technique*. Gholamian et al. [10] survey the logging field, namely by outlining main logging practices and trends. They include log parsing in one of their chapters, and describe methods found in literature with respect to advantages and disadvantages.

To the best of our knowledge, the work by El-Masri et al. [8] is the only survey solely intended to analyze log parsing approaches. This work aims to bridge the gap between industry and academia, and to do so, the authors use seven criteria that describe log parsing

[*]Code and dataset are available at https://github.com/spetrescu/are-log-parsers-ready-for-prime-time.

methods, namely *mode*, *coverage*, *efficiency*, *scalability*, *system knowledge independence*, *delimiter independence*, *parameters tuning effort*. Again, however, little attention is paid to the problem of deriving insights about entire software stacks.

In contrast to all of these works, we discuss an important aspect that has not been sufficiently researched, namely the performance of log parsing methods on heterogeneous data as typically found in the logs of complex applications that encompass a full stack of software. We consider previous evaluations found in existing studies to be incomplete, as they reuse outdated homogeneous datasets in their evaluations, and do not consider current data characteristics.

3.2 Log parsing's goal and its evaluation in literature

In the literature, log parsing is formulated clearly as the task of mining the underlying software logging statements that generate runtime logs, also known as log templates. However, close inspection of the de-facto standard for evaluation reveals that the metrics used do not reflect this goal. As a consequence, confusion arises around the role of log parsing, both from research and industry perspectives.

Implementing data-driven processes such as automated log analysis requires leveraging data to the fullest, and for such pipelines, each step has to provide input that maximizes the chances of drawing actionable insights for subsequent tasks. Consequently, as log parsing is the first step, it is paramount for it to prevent any feature-rich information loss in its transformation, as that would result in hindering the performance of all subsequent tasks. Specifically, it is of utmost importance to preserve possible latent features present in the raw data, for log parsing to provide a maximally feature-rich representation for downstream tasks. Thus, the desired representation for a raw log is one in which the discovered variables are replaced with generic tokens, and the rest of the initial log structure remains the same. For example, considering the runtime log: '`Received block blk_8829 of size 671 from 10.251.38.214`' with its respective ground truth template: '`Received block <*> of size <*> from <*>`', it is desirable to parse the log message such that the initial data feature-richness is preserved after applying the transformation. In this case, the runtime log would be parsed to: '`Received block <*> of size <*> from <*>`', with the accompanying list of discovered variables '`['blk_8829', '671', '10.251.38.214']`', which fully matches the ground truth template from a textual perspective. However, the standard in the field is to evaluate methods for solving a different problem, which in turn causes ambiguity for applying these in practice.

The standard evaluation metric used in the field actually assess parsers' ability to classify logs, rather than assessing the quality of the log templates that they generate. Specifically, *parsing accuracy*, introduced by He et al. [14], measures parsers' ability to identify types of logs, which reflects the ability to group similar log messages rather than log parsing's primary goal of separating static from variable content. Proof of its misalignment is that a perfect parsing accuracy can be obtained even if templates generated are entirely different from the ground truth templates. Considering the previous example, the log message: '`Received block blk_8829 of size 671 from 10.251.38.214`' can be parsed to something entirely different from the ground truth template, such as '`Example template`' or '`E v e n t t e m p l a t e`' and a perfect parsing accuracy can still be obtained. Thus, we consider that, as a consequence of adopting this metric, the field has drifted away from the goal of log parsing and from obtaining realistic estimates of the quality of templates produced by the parsers.

Another issue that is currently not addressed in the field has to do with the way in which the evaluation design is setup. Not only are the standard evaluation metrics used incomplete, the methods are, more importantly, trained and tested on the same data, which makes their claims questionable from the perspective of generalization. This is a very important observation, as applying these methods in a production context requires at least some estimate of how they perform on unseen data. Here, having to deal with unseen data is the default scenario, regardless of the production environment considered. Specifically, the goal is to deploy software at a fast pace, and that means realising new versions of existing systems, which in turn can generate new logging statements and/or put the system in new states to trigger previously unseen logging statements. Thus we argue that, as yardstick of practical relevance, log parsers should also be evaluated on unseen data, as that is paramount for providing an estimate of their robustness and general applicability.

RQ2. To remove the ambiguity around the goal of log parsing once and for all and to answer the first research question, we define log parsing as the task of identifying log templates in a runtime log message, to extract the static parts that were present in the original software logging statements. In the next section we address the evaluation issue by proposing metrics that are aligned with the goal of log parsing.

3.3 Log parsing's evaluation metrics

RQ2.1. To ensure that evaluation reflects the goal of log parsing discussed in the previous section, and to answer the second research question, we consider two evaluation metrics that align best with the goal of log parsing, namely *log template accuracy* and *edit-distance*. These are inspired by similar metrics that have been proposed in recent work by Nedolski et al. [30] and Liu et al. [22], which also aim to address the problem of the incomplete evaluation standard in the field.

Log template accuracy and *edit-distance* aim to asses the quality of templates generated, as they are intended to compare parsed logs against their respective ground truth templates. Consequently, as this aligns perfectly with the goal of log parsing, we adopt these metrics. It is necessary for the field to adopt metrics based on textual similarity, as they quantify the ability of parsers to produce templates, that are evaluated against the ground truth labels, thus providing estimates for the quality of the output generated. Furthermore, we argue that the field should adopt these metrics since they objectively quantify how similar templates generated from log data are to the templates that generated the log data.

Log template accuracy is defined as the ratio of the number of correctly parsed logs, over the total number of logs. Specifically, a log message is parsed correctly if its textual content matches the ground truth template (generated by human experts or mined from software code). Log template accuracy can range from 0 to 1, zero meaning that a parser is unable to match any of the corresponding ground truth templates, whereas one means that the parser is able to match all of the corresponding ground truth templates. For example, considering the following log messages: '`Example log 1`' and '`Log 2`' with their respective ground truth templates: '`Example log <*>`' and '`Log <*>`', an accuracy of 1 can be obtained only if both parsed logs match the ground truth templates. For a single log message, *log template accuracy* can be either 1 or 0, for a successful or unsuccessful match respectively. This metric is, compared to the state of the art in evaluation, strict, as parsed log have to perfectly match their respective ground truth templates. Intuitively, a higher *log template accuracy* means better performance, and a lower accuracy means worse performance.

Edit-distance is defined as the Levenshtein distance [44]. As it is desirable to have an estimate of how close the parsed template is from its respective ground truth label, this metric is as a more fine-grained alternative to *log template accuracy*. Specifically, it represents the minimum number of operations required to transform a parsed log into its ground truth correspondent. Compared to the *log template accuracy* metric, *edit-distance* is less strict, and is able to provide a finer-grained estimate of how close the parsed template is from the ground truth. Intuitively, a high *edit-distance* means poor performance, whereas, a low edit-distance means good performance.

3.4 Log parsing's performance on publicly available data

Based on the goals of log parsing and the performance metrics considered in the previous section, parsers were tested on log data in the context of modern software ecosystems.

To test log parsers, experiments are run on nine publicly available datasets. These have been used extensively in the field for evaluations and as a consequence, the log parsing methods considered are expected to obtain the best *log template accuracy* and *edit-distance scores*, compared to parsing combined data or industry data. Additionally, parsers are expected to obtain best scores on these datasets, as they contain logs generated within less complex software environments, compared to modern large scale systems.

The results of our experiments are summarized in Table 3.1 and 3.2. 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). Note that, across the different datasets and methods, an average *log template accuracy* of 0.2 is obtained. In contrast, previous reports obtained *parsing accuracy* average results of 0.67. However, *parsing accuracy* is not reflective of the log parsing goal, and previous results therefore misrepresent the quality of existing solutions. Specifically, one might think that an average 0.67 *parsing accuracy* represents matching 1200/2000 templates perfectly, whereas, the actual number of templates matched, based on *log template accuracy* is 400/2000. Thus, our results differ from previous estimates by a large margin, and highlight the incompleteness and ambiguity of the de-facto standard for evaluation in the field.

Furthermore, we observe that accuracies seem to correlate for particular datasets. For example, most methods obtain accuracies of approximately 0.6 for the *Apache* dataset, 0.3 for *BGL*, 0.6 for *HPC*, 0.1 for *Spark*, 0 for *HDFS*. This indicates that methods rely on dataset-specific hard-coded rules and heuristics. *NuLog*, on the other hand, is (notably more) robust with an average accuracy of 0.47. Based on this observation, we argue that it is worthwhile to consider methods that are not heavily reliant on hard-coded rules and heuristics but instead employ methods based on machine learning that, when trained well, have the potential to generalize better in practical applications.

RQ2.2. Our findings indicate that parsers are not able to generate templates that match their respective ground truth templates, and compared to the previous estimates in the field, we discover that the actual performance differs by a large margin. We observe *NuLog* as being the only exception, a method that proved to be robust to the various datasets, compared to the rest of the methods. We believe that this is a consequence of being intrinsically designed to move away from hand-crafted rules.

3.5 Log parsing's performance in modern software ecosystems

To answer the research question, we test methods on a dataset that is intended to emulate industry data, and on industry data generated within the software ecosystem of a large international financial institution. The results can be found in the following sections.

3.5.1 Combined dataset

To test parsers on highly diverse log data, we create a dataset that represents a contemporary software ecosystem. Specifically, we combine the nine publicly available datasets intro-

Table 3.1: Log template accuracy results after running each method 10 times, for each dataset. The measurements are averaged over 10 runs.

Dataset	Drain	IPLoM	LenMa	LFA	LKE	LogCluster	LogMine	LogSig	MoLFI	NuLog	SHISO	SLCT	Spell
Apache	**0.694**	**0.694**	0	0.688	0	0	**0.694**	0	0.270	0.560	0	0.424	**0.694**
BGL	0.341	0.292	0.082	0.230	0.057	0.067	0.220	0.081	0.324	**0.853**	0.064	0.207	0.196
HDFS	0	0	0	0	0	0	0	0	**0.435**	0	0	0	
HealthApp	0.238	0.158	0.136	0.149	0.133	0.138	0.220	0.126	0.166	**0.341**	0.041	0.322	0.152
HPC	0.620	0.638	0.632	0.609	0.360	0.632	0.632	0.509	0.632	**0.827**	0.226	0.661	0.530
Mac	0.224	0.041	0.132	0.101	0.172	0.162	0.228	0.118	0.042	**0.274**	0.163	0.148	0.032
OpenStack	0.018	0	0.018	0.008	0.010	0.010	0.010	0.010	0	**0.359**	0.018	0.119	0
Spark	0.194	0.192	0.004	0.190	0.001	0.006	0.038	0	0.208	0.204	0.004	**0.543**	0.192
Windows	0.159	0.001	0.154	0.142	0.148	0.153	0.156	0.150	0.006	**0.387**	0.151	0.140	0.004
Avg.	0.276	0.276	0.128	0.235	0.128	0.129	0.244	0.110	0.183	**0.471**	0.074	0.284	0.200

Table 3.2: Edit-distance results after running each method 10 times, for each dataset. The measurements are averaged over 10 runs.

Dataset	Drain	IPLoM	LenMa	LFA	LKE	LogCluster	LogMine	LogSig	MoLFI	NuLog	SHISO	SLCT	Spell
Apache	10.426	10.442	13.760	10.576	14.872	16.274	10.426	14.456	10.179	**4.679**	12.648	11.234	10.442
BGL	4.930	6.882	8.373	12.524	12.582	12.955	18.598	11.921	10.969	**2.981**	8.630	9.841	7.900
HDFS	8.820	16.208	10.762	30.819	17.940	28.340	16.524	18.989	19.843	**2.867**	10.114	13.641	9.274
HealthApp	18.502	11.882	16.540	20.277	28.422	16.844	19.598	17.088	21.859	**11.595**	24.430	13.840	8.540
HPC	2.015	2.323	2.906	3.182	7.649	3.580	3.218	4.419	4.845	**1.275**	7.854	2.625	5.129
Mac	19.882	20.928	19.984	41.804	26.260	21.328	**17.048**	28.043	28.273	21.417	19.810	34.560	22.593
OpenStack	28.386	23.330	18.535	28.138	29.173	31.486	23.980	21.881	67.894	**5.605**	18.582	20.986	27.984
Spark	3.532	5.246	10.945	9.178	18.116	17.082	16.004	12.968	14.146	**2.921**	7.910	6.028	6.129
Windows	6.172	15.758	20.662	10.238	11.834	6.967	6.919	7.667	11.943	6.067	5.624	7.006	**4.406**
Avg.	11.407	12.555	13.607	18.526	18.539	17.206	14.702	15.281	21.106	**6.600**	12.8445	13.307	11.377

Table 3.3: Log template accuracy results after running each method 10 times, for each dataset. The measurements are averaged over 10 runs.

Dataset	Drain	IPLoM	LenMa	LFA	LKE	LogCluster	LogMine	LogSig	MoLFI	NuLog	SHISO	SLCT	Spell
Combined	0.258	0.214	0.140	0.180	0.140	0.128	0.258	0.092	0.180	**0.323**	0.067	0.280	0.186
Industry	**0.056**	0.041	0.001	0.022	0.001	0.002	0.054	0	0.048	0.050	0.002	0.034	0.041

Table 3.4: Edit-distance results after running each method 10 times, for each dataset. The measurements are averaged over 10 runs.

Dataset	Drain	IPLoM	LenMa	LFA	LKE	LogCluster	LogMine	LogSig	MoLFI	NuLog	SHISO	SLCT	Spell
Combined	17.302	14.094	18.559	24.144	27.633	21.306	15.858	24.756	19.021	**8.721**	21.791	22.274	17.454
Industry	27.201	24.122	32.280	41.960	68.551	47.006	23.506	37.911	49.145	23.239	31.908	46.690	24.664

duced in the previous section. This dataset is more reflective of industry because it is more heterogeneous, which is something we expect in industry, due to how modern software subsumes various components that can generate different types of log data, for example, logs that are produced within different programming languages and frameworks. Thus, this dataset is considered as the baseline for estimating how log parsers perform in practical settings, and a combined dataset is created by drawing a uniform sample across the nine datasets of *2k* log messages[†].

We repeated our previous experiment on this combined dataset, and the results can be found in Table 3.3 and Table 3.4. A comparison between the homogeneous and the heterogeneous case indicates that methods show a marked performance drop. We attribute this result to the limited variability of logs that originate from a specific system. Specifically, it is expected for log parsers to separate between the constant and variable part of a message by leveraging access to similar messages that are different only in terms of variables. For example, considering the following log messages: '`Template log 1`' and '`Template log 2`', it is expected for methods to leverage the similarity between these two, and to eventually discover the underlying template: '`Template log <*>`'. However, if log data has more diversity and less logs that originate from the same system, the performance takes a substantial hit in terms of *log template accuracy* and *edit-distance*, as methods are not able to discover patterns in the data as easily. Consequently, as methods have to parse the same amount of log data (*2k*) as in the previous experiment, but with less logs from the same distribution (system), it becomes harder for parsers to recognize variables and to generate quality templates. We know that methods rely on having access to these types of features (similar logs with different variables) to be able to discover patterns in data. However, this is not the case with highly diverse logs. Thus, we argue that it is worthwhile creating methods that are able distinguish between constants and variables in a general sense, without relying on heuristics.

3.5.2 Industry dataset

A labeled dataset of *2k* samples was gathered from the software infrastructure of a large international bank. The performance obtained on industry data is subpar, and our findings indicate that log parsing is in fact harder to apply in industry settings. The results for running log parsers on industry data can be found in Table 3.3 and Table 3.4. The highest *log template accuracy* is roughly 0.05, which means that the best performing methods parse only 100/2000 log messages correctly. Thus, applying such parsers in production settings is problematic.

RQ3. Based on the experimental results and an error analysis, in the following paragraphs, we answer the third research question by analyzing the reasons behind obtaining

[†]To obtain *2k* log lines, we sampled two extra log lines from the *Mac* dataset. We choose this dataset for adding two extra samples because it contained the biggest log diversity, having the highest number of underlying templates.

such poor performance scores on industry data.

Data heterogeneity/diversity. One of the factors that resulted in the poor results is data heterogeneity. Specifically, data diversity in modern systems is a major problem for parsers, making them unable to identify the underlying templates correctly. In our experiments we discover that the properties of the data found in the industry dataset are very similar with the properties of the combined dataset, as logs originate similarly from different data distributions (systems), as a consequence of being centralized. In comparison to the combined dataset, the log diversity found in the industry dataset is higher, but the properties of the dataset are intrinsically the same (clusters of log data generated by different systems). Compared to the individual datasets, the log diversity found in the industry dataset is incomparably higher, as it is generated by a incomparably larger number of software components. Consequently, this makes it extremely difficult for parsers to discover the underlying templates on industry data, which is reflected in the *log template accuracy* results, and thus the problem is arguably harder than expected from the results obtained on the combined dataset. In terms of *edit-distance* we observe a drop in performance, which can also be attributed to the aspects discussed in the next paragraphs.

Jargon and high information denseness. Another factor that hinders parsers performance is the jargon present in data, and the high information denseness. Compared to publicly available data, for a production log, templates and parameters are hard to separate and identify. For example, parameters can contain various alphanumeric characters, but also symbols that make it hard for parsers to generate templates that match the corresponding ground truth labels. Specifically, parsers rely on heuristics that prove not working on industry data. Additionally, another troublesome aspect is the way in which log messages are cascaded. For example, it might be that an error occurs on a specific infrastructure resource, which then sends that information to other resources which concatenate and display similar information. In these cases, parsers have to be robust against nested templates.

Labeling industry log data. Creating labeled industry data is a hard task, which in turn affects negatively estimating the performance, as subjectivity is added during the labeling process. Due to data heterogeneity and jargon, labeling industry log data is a harder task than labeling publicly available data, even for experts, as getting access to the ground truth code logging statements is something very difficult and, in some cases, even impossible. Specifically, we found that the biggest issue with labeling industry log data is that its nature makes deciding between what is a constant and variable in a log message very hard. Most messages are formatted in ways that generate ambiguity when identifying the templates. For example, messages can contain many concatenated symbols, or have duplicated text concatenated, which makes it difficult to decide on a ground truth.

3.5.3 Generality of our findings

We analyze[‡] the similarities and differences between the publicly available log datasets, the combined dataset, and the industry dataset, and show how combined data is of the same nature as industry data, in contrast to most publicly available data.

In Table 3.5 we present the different measurements used to quantify the heterogeneity of the log datasets used in our experiments. We use three metrics to serve as a proxy for the heterogeneity of log data, and we draw comparisons. Specifically, we choose three metrics for the analysis of each dataset, namely: *unique number of words*, *unique number of characters*, *unique number of log lines' character length*.

In Figure 3.1, Figure 3.2, and Figure 3.3 we see how the *Combined* and *Industry* data are always toward the top right of each visualisation, proving that these data are inherently more heterogeneous, as they combine more characters, words, etc. However, we also observe that a publicly available dataset also scores high in terms of the metrics considered. The reason for this is that the nature of this particular dataset is actually similar to industry data, being the only exception from the publicly available data.

Table 3.5: Statistics for the log datasets analyzed.

Dataset	Apache	BGL	HDFS	HealthApp	HPC	Mac	OpenStack	Spark	Windows	Combined	Industry
Word level	874	2068	3599	1512	510	2981	1445	1970	1206	3123	4421
Character level	46	75	56	71	65	90	72	70	82	91	92
Log length level	9	114	59	55	50	186	50	63	66	157	181

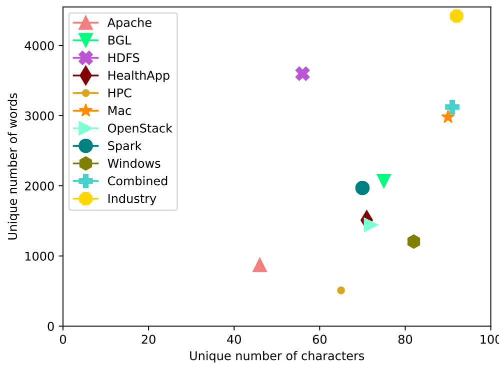


Figure 3.1: Unique number of characters versus unique of words.

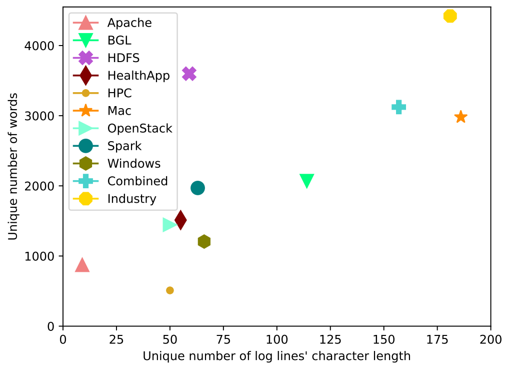


Figure 3.2: Unique number of log lines' character length versus unique number of words.

[‡]Analysis' code is available at https://github.com/spetrescu/analysis-data-heterogeneity.

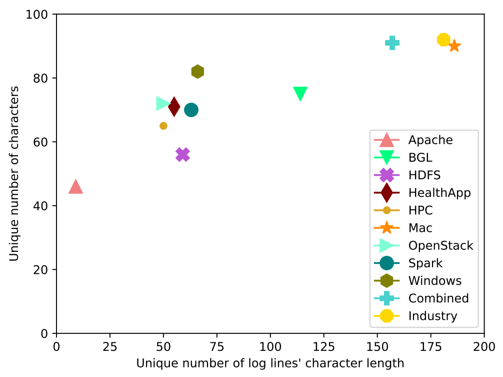


Figure 3.3: Unique number of log lines' character length versus unique number of characters.

3.6 Discussion

The increase in data heterogeneity witnessed over the past decades makes applying current log parsing methods in industry questionable. As in practice automated log analysis processes rely on log parsing methods, we investigate their performance on heterogeneous data, to decide if they are applicable. Thus, we investigate the log parsing field, and discover that the field adopted incomplete evaluation strategies. To address this, we consider metrics that are aligned with the goal of log parsing, and evaluate methods accordingly. We discover that the methods perform poorly, which makes their applicability questionable for facilitating quality input for downstream tasks. Furthermore, we combine publicly available logs into a dataset that emulates industry data, and compare it to both publicly available data and industry data. We evaluate methods on these datasets and discover that (1) the performance is poor as methods are not designed to account for jargon-and-information-heavy data , (2) industry data may be harder to label than expected, and (3) the paradigm needs to be shifted to align with industry requirements.

In this chapter, we contribute by answering research questions regarding (1) what is the goal of log parsing, (2) how is log parsing currently evaluated, (3) what metrics can be considered for evaluating log parsing methods in a way that is aligned with its original goals, (4) how do log parsing methods perform using the considered metrics, (5) what is the performance of log parsers in the context of modern systems, and (6) what are the main challenges for applying log parsing in industry.

4
Entity parsing: a new paradigm for parsing log messages

The log parsing literature claims to transform raw logs into "structured information", but at a closer look, this claim becomes questionable, as the structure generated by log parsing is not necessarily the same as what might be reasonable to expect. Specifically, after an investigation of the field, we discover that the claim holds only for a very restricted interpretation of "structured". In the current log parsing paradigm, the structure created from raw logs only allows for a binary type of classification of logs' textual content, namely only differentiating between constants and variables. This creates unclarity, as it would be reasonable for one to assume that "structured information" represents a data structure that encompasses a variety of entities, rather than only two entities – the static and variable parts of logging statements. Consequently, in light of this observation, the nature of the output provided by log parsing makes its claimed benefits questionable.

For creating structured information, it is desirable to map various entities such as software components' identifiers, IP addresses, etc., and to add these as values to a data structure that contains key-value pairs for the various fields of interest. However, this type of output is currently not supported by the log parsing paradigm, thus preventing the creation of a maximally feature-rich representation for downstream tasks. To better understand the current limitations of the field, in Figure 4.1 we visualize the difference between the claimed benefits of log parsing versus its actual capabilities. We consider this comparison

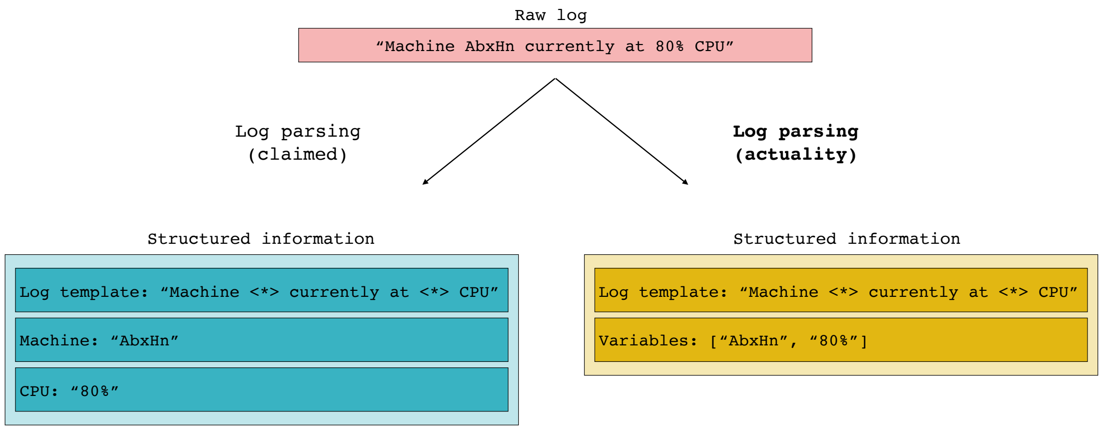

Raw log
"Machine AbxHn currently at 80% CPU"
Log parsing (claimed)
Log parsing (actuality)
Structured information
Log template: "Machine <*> currently at <*> CPU"
Machine: "AbxHn"
CPU: "80%"
Structured information
Log template: "Machine <*> currently at <*> CPU"
Variables: ["AbxHn", "80%"]

Figure 4.1: Comparison between the output claimed to be generated by log parsing versus the actual output. In reality, log parsing separates between constants and variables, and generates a very limited type of structured information. However, it is desirable to obtain actual structure from raw log data, such as the data structure on the left side of the figure.

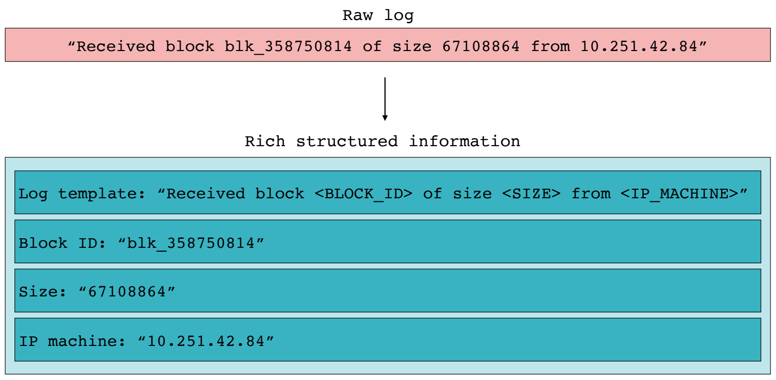

Raw log
"Received block blk_358750814 of size 67108864 from 10.251.42.84"
Rich structured information
Log template: "Received block <BLOCK_ID> of size <SIZE> from <IP_MACHINE>"
Block ID: "blk_358750814"
Size: "67108864"
IP machine: "10.251.42.84"

Figure 4.2: Example with an instance of *rich structured information*. For a given application, there is a predefined set of fields, which are then identified in runtime logs. It can be the case that for a given log, certain entities need not be matched, and for specific keys in the dictionary to remain empty.

to be of utmost importance, as it is misleading for both researchers and practitioners to assume that log parsing is designed for transforming raw logs into actual structured information. Thus, to remove the unclarity around the claimed "structured information", we define *rich structured information* as a dictionary that contains key-value pairs that correspond to predefined fields of interest in a particular software system, and their respective values matched in runtime logs. We visualize an instance of *rich structured information* in Figure 4.2. Here, we observe that a raw log is transformed into a dictionary, where the keys are predefined and the values are extracted from the raw log.

However, compared to only identifying variables and constants, it is harder to create *rich structured information*, as this implies adding an extra layer of complexity to the problem of log parsing, which has yet to be solved in the first place. Specifically, as the literature has yet to produce clear winners for solving log parsing, the key question that begs to be asked is: "why to tackle a problem that is actually more complex, instead of solving the less complex one first?". While this question is reasonable, we still believe that there is merit in trying to generate *rich structured information*, instead of focusing on solving log parsing. Specifically, we argue that solving the problem of generating *rich structured information* would bring incomparably more benefits than solving log parsing, as it would allow for abstracting away from raw logs, toward a representation that is incomparably more meaningful. Actually, by solving the problem of generating *rich structured information*, log parsing would be automatically solved, as the former encompasses the latter. Lastly, from the perspective of generating *rich structured information*, there is no point in solving log parsing first, as that would not move us closer to the initial goal of creating structure from raw log data.

Nevertheless, creating *rich structured information* involves great challenges, associated with modern systems' log data heterogeneity and difficulty in accessing systems' source code. Specifically, logs generated by modern systems are highly heterogeneous, and identifying fields of interest requires solutions that account for a tremendous number of log formats. Secondly, to create solutions that are fine-tuned to systems' intricacies, access to source code might be required. However, accessing source code can be problematic for systems that have a highly complex underlying software infrastructure, and sometimes even infeasible as the subsumed software components are hard to identify and analyze. To address these challenges, (1) solutions that are robust to log data heterogeneity have to be created, and (2) source code access has to be provided, or potential ways to bypass that have to exist.

Tackling generating *rich structured information* is undoubtedly worthwhile, as long as solutions consider leveraging (1) a general understanding of what entities are by means of training on existing log corpora, and (2) source code, for fine-tuning to systems' intricacies. Specifically, to create solutions that are both robust and fine-tuned for particular use-cases, we firmly believe that there is no way around tackling the tremendous log heterogeneity of systems without leveraging (1) the tremendous amount of publicly available log data, and (2) systems' source code. Consequently, we would like to discuss addressing the limitations of the previous log parsing paradigm by tackling the aforementioned challenges. Thus, in this chapter we answer the following research questions:

> **RQ4:** How is the log parsing paradigm aligned with industry goals, and how can that be improved?
>
> **RQ4.1:** How to discover representative entity types for variables present in logging statements?

RQ4.2: How can we generate a dataset that operates in the entity parsing paradigm?

RQ4.3: How to create an entity parsing dataset that is representative of industry?

RQ4.4: How to tackle entity parsing to provide an accurate and scalable solution?

4.1 Current limitations of log parsing

The claim that log parsing transforms raw logs into structured information is misleading, as in actuality the structure obtained is limited to a binary classification of logs' textual content, namely only to identifying static and dynamic parts in log messages. Consequently, log parsing is unable to provide *rich structured information* output, as the current paradigm is not designed to account for differences between identified variables. Although there are contexts where output that does not differentiate between types of variables can be useful, for example, when mining frequent patterns in a given batch of data, downstream tasks benefit from a feature-rich representation of the input data, and, as log parsing cannot facilitate that, the current paradigm is incomplete.

RQ4. Log parsing is not aligned with what is desirable in industry, namely abstracting away from raw logs toward structured information. Specifically, the claim according to which log parsing transforms raw logs into structured information is misleading, as its output fails to provide actual structure, being unable to separate between the various entities present in log messages, and rather only being able to differentiate between their static and variable textual content. For example, log parsing is not able differentiate between IP addresses, integers, software components' identifiers, etc., toward mapping entities that are of interest in log messages. In industry, it is desirable to account for the differences between these, as this (1) generates a feature-rich representation of the input for downstream tasks, and (2) may lead to acquiring critical insights from the logs. Thus, log parsing is not aligned with what is necessary in practice, namely abstracting away from raw data toward structured information. As a consequence, its output is incomplete, which makes its applicability questionable.

Improving the current paradigm means addressing the limitations of log parsing's output. Specifically, improving log parsing means generating *rich structured information* output, transforming raw logs into a structure that contains key-value pairs that correspond to predefined fields of interest for a particular system, and their respective values matched in runtime logs. We define this way of transforming raw logs into structured information as *entity parsing* – a paradigm in which the focus is to identify various entities/fields of interest, and in turn generating *rich structured information*.

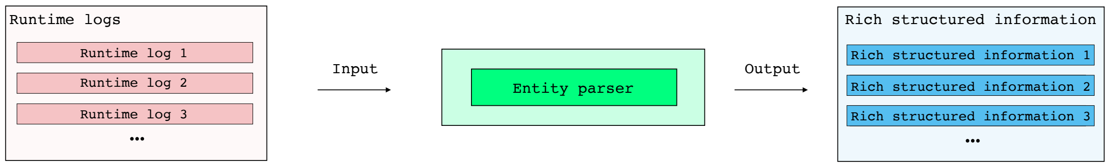

Runtime logs
Runtime log 1
Runtime log 2
Runtime log 3
...
Input
Entity parser
Output
Rich structured information
Rich structured information 1
Rich structured information 2
Rich structured information 3
...

Figure 4.3: Entity parsing's goal. Runtime logs are transformed into *rich structured information*.

4.2 What is the entity parsing paradigm?

Log parsing is not designed to generate *rich structured information* from raw logs, making the nature of its output incomplete. Furthermore, log parsing fails to provide quality log templates for downstream tasks, demonstrated by the poor performance obtained on public and industry data. To address these limitations, we introduce *entity parsing*, a paradigm that abstracts away from raw logs, generating *rich structured information*, while assuring high quality log templates. We visualize *entity parsing*'s main goal in Figure 4.3.

Entity parsing may leverage (1) a general understanding of what entities are by means of using existing log datasets for pretraining, and (2) systems' source code for fine-tuning to their intricacies. Specifically, methods that operate in the *entity parsing* paradigm may leverage (1) public log data, and (2) systems' log data, to create solutions that are able to generate *rich structured information*. We visualize the *entity parsing* paradigm in Figure 4.4. Here, an "entity parser"* leverages both (1) a dataset intended for acquiring a general understanding of entities in log messages, and (2) a dataset intended for fine-tuning, comprised of logs generated by a particular system. To generate the latter, access is required to both (1) runtime logs generated by the particular system and (2) the system's source code for generating ground truth labels.

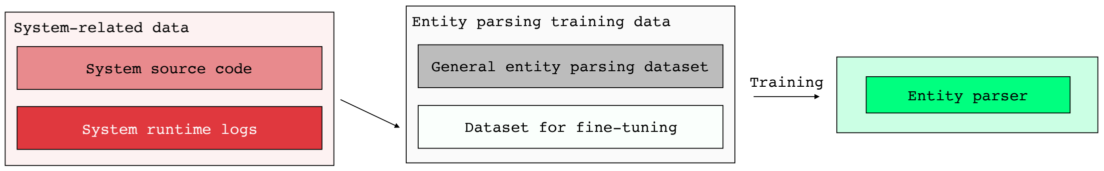


Figure 4.4: Entity parsing paradigm. An *entity parser* can operate in this paradigm by pretraining on a general *entity parsing* dataset, and subsequently fine-tuning to the intricacies of a system by means of using a dataset that contains log data generated within its infrastructure.

*"Entity parser" refers to a method that operates in the *entity parsing* paradigm.

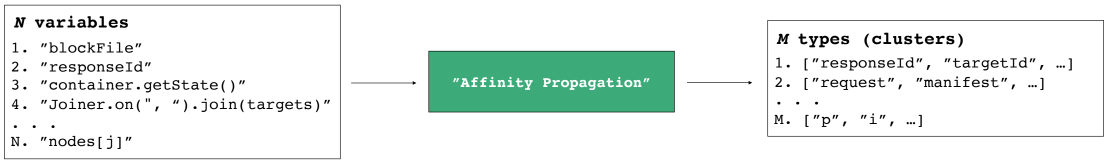


Figure 4.5: Discovering types of variables using *Affinity Propagation*. We mine variables from software repositories, and subsequently cluster these together to discover generalizable entity types.

4.3 Representative fields in rich structured information

Generating *rich structured information* within the *entity parsing* paradigm implies recognizing various entities in log messages, such as IP addresses, software components' identifiers, system paths, etc. However, considering generalizable entities is challenging, as entity types in log messages are something inherently subjective, thus open to interpretation. For example, it might be reasonable for one to consider "system path" as an entity type, as this type of variable can generally be found in software systems' logging statements, whereas for another person to consider it irrelevant. Nevertheless, there is merit in deciding on general types of variables to generate *rich structured information*, and, in case particular types are not applicable in the context of a specific application, allowing for reconsidering entities.

Defining relevant types is challenging, as the right trade-off between generality and specificity has to be considered. However, if the right trade-off is considered, the benefits of recognizing entities are tremendous, enabling downstream tasks to use *rich structured information*. Thus, to tackle this, we mine variables' names logged in publicly available software code, and analyze them by means of clustering, assuming that there is a connection between variables' names and entity types. However, as the clustering of variables' names may change with the inclusion or exclusion of data, this means that one might be influenced in dropping/adding types that were initially relevant/irrelevant. Furthermore, as it might be the case for specific types to be irrelevant for particular systems, there needs to exist the possibility of reconsidering types if (1) more data is added to the analysis, or if (2) specific types are not applicable to the particularities of an application.

To cluster variables' names, we leverage an unsupervised clustering algorithm, designed for clustering textual data, namely *Affinity Propagation*. We choose this algorithm as (1) the desired number of clusters is not known beforehand, and (2) as the objective is to cluster words, instead of sentences, or large pieces of text. In Figure 4.5 we visualize *Affinity Propagation*'s workflow, namely for a given list of N words it is able to generate M types, which are called "exemplars". Consequently, after mining variables in software repositories, we run *Affinity Propagation*, and analyze[†] its output to decide on types that might generalize

[†]Code and data are publicly available at https://github.com/spetrescu/affinity-propagation-entity-types.

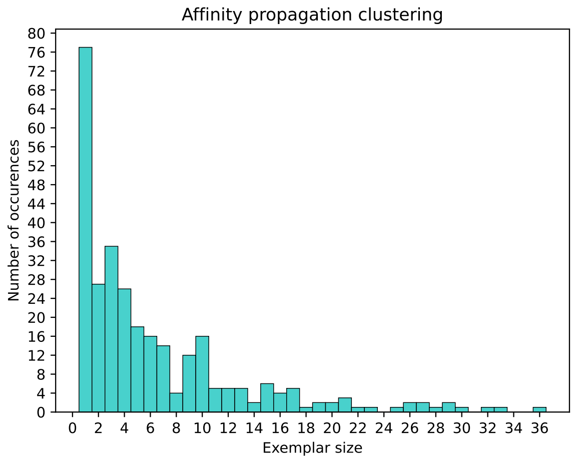


Figure 4.6: Example of clustering of variables example using *Affinity Propagation*. The results seem to follow a power law distribution, which means that only a few clusters have a large size, in comparison to a high number clusters which have a rather low size. To generate this histogram we used the following hyper-parameters: `random_state=5`, `affinity="precomputed"`, `damping=0.5`, `max_iter=200`.

Table 4.1: Types of log templates identified.

No.	Entity type	Example(s)
1	`Generic Type`	`specs, range, targetAddr, nnc, kvstart, avg`
2	`Path`	`basePath, dataPath, filePath, filePath2, fileSrcPath, fullPath`
3	`Id`	`reduceId, resID1, responseId, , results, , shellId, threadId`
4	`File`	`destFiles, editFile, hostsFile, keytabFile, outputFile`
5	`Priority`	`avgRespTimePriority, callVolumePriority, appPriority, priority`

to other systems.

After analyzing the exemplars with a high number of variables, we decide on five entity types, and display these in Table 4.1. In Figure 4.6 we plot the distributions of unique number of exemplar sizes and number of occurrences. Many exemplars are rather low in size, whereas there a rather low number of exemplars are high in size – being representative of a high number of variables.

RQ4.1. To discover representative entity types for variables present in logging statements, the variables of seven software code repositories have been analyzed. Specifically, we have tackled the problem of finding entity types by leveraging *Affinity Propagation*. We consider *five* variable types and assume these to generalize across other systems.

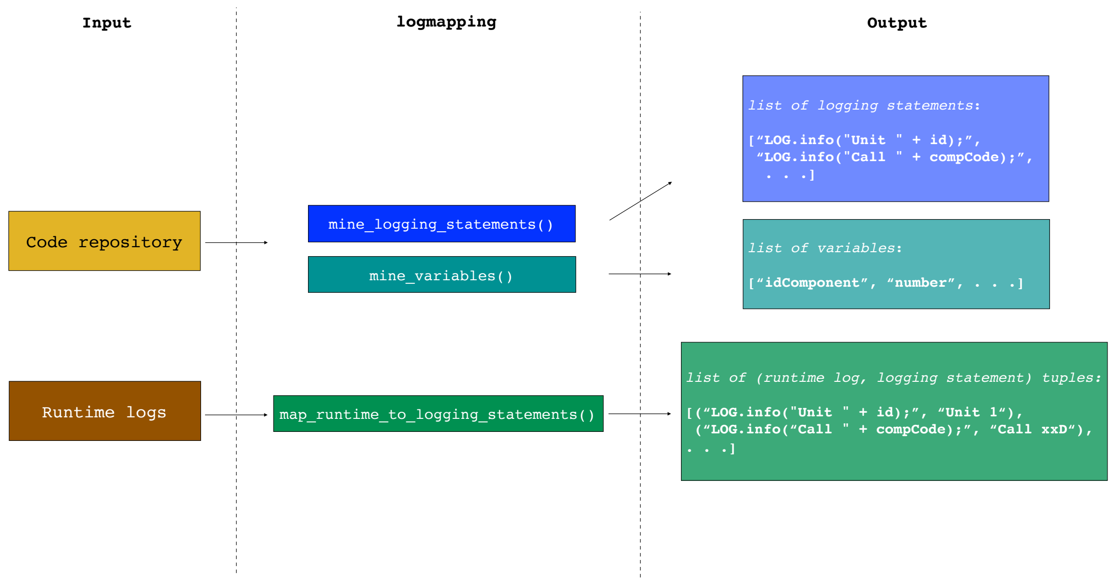

Input
logmapping
Output
Code repository
Runtime logs
mine_logging_statements()
mine_variables()
map_runtime_to_logging_statements()
list of logging statements:
[“LOG.info("Unit " + id);”,
“LOG.info("Call " + compCode);”,
. . .]
list of variables:
[“idComponent”, “number”, . . .]
list of (runtime log, logging statement) tuples:
[(“LOG.info("Unit " + id);”, “Unit 1“),
(“LOG.info(“Call " + compCode);”, “Call xxD“),
. . .]

Figure 4.7: Workflow of out tool that creates `(‘runtime log’, ‘ground truth logging statement’) mappings.`

4.4 A dataset that operates in the entity parsing paradigm

Creating a dataset – the *Entity* dataset – that operates in the *entity parsing* paradigm, implies mapping runtime logs to their underlying logging statements. For humans, it might be straightforward to analyze a logging statement and decide if a particular runtime log could have been generated by it, but the process would be too slow, thus requiring automating it. However, mapping runtime logs automatically to their underlying logging statements is challenging, because of the various number of formats and types of logs that systems contain. As a consequence, we need to address the challenges associated with (1) creating a mapping between runtime logs and their underlying software code logging statements, and (2) generating the ground truth labels for the mapped `(‘runtime log’, ‘logging statement’)` tuples. To tackle these challenges, we create a tool able to (1) create a representation of the logging statements that allows for comparisons with runtime logs, (2) generate the ground truth templates for runtime logs based on the discovered mapping. In Figure 4.7 we visualize the functionality of the tool, namely (1) mining logging statements, (2) mining variables present in logging statements, and (3) creating the mapping from runtime logs the underlying logging statements. Specifically, the tool is created as a Python library[‡], and in Figure 4.8 we display its design, in terms of separation of concerns.

[‡]Code is available at https://github.com/spetrescu/logmapping.

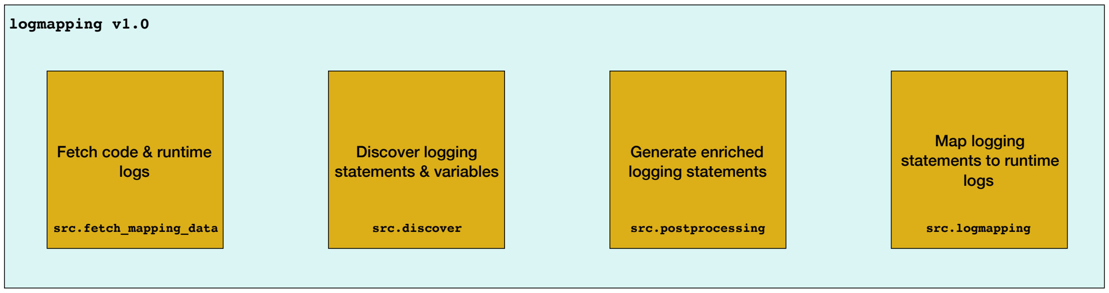

logmapping v1.0
Fetch code & runtime logs
src.fetch_mapping_data
Discover logging statements & variables
src.discover
Generate enriched logging statements
src.postprocessing
Map logging statements to runtime logs
src.logmapping

Figure 4.8: Tool's design in terms of separation of concerns.

Table 4.2: Structure of each entry in the *Entity* dataset.

Runtime log	Log template	Entity log template	(Array) Variable-Entity tuples

Consequently, after running the tool's processes, each `(‘runtime log’, ‘logging statement’)` tuple discovered is saved, and appended to the *Entity*[§] dataset. The dataset is comprised of entries that contain four attributes, namely (1) the runtime log, (2) the ground truth logging statement, (3) the entity log template (that replaces variables with specific entity types), and (4) an array of `(“variable’s name in logging statement”, “variable’s entity type”)`. For creating the *Entity* dataset, we mine six available systems, for which runtime logs are publicly available, and we display these in Table 4.3.

Table 4.3: Systems to be identified.

System	Type	Code available
Hadoop	Distributed System	GitHub
Spark	Distributed System	GitHub
Zookeeper	Distributed System	GitHub
OpenStack	Distributed System	GitHub
Linux	Operating System	GitHub
Apache HTTP Server	Server Application	GitHub

4.4.1 `mine_logging_statements()`

To create a mapping with runtime logs, we identify all the logging statements in a given software repository. Specifically, we analyze the software code repository and iterate through

[§]The *Entity* dataset is available at https://github.com/spetrescu/entity-dataset.git all the files to discover logging statements. Based on the specific programming language of the software codebase, we leverage a series of regular expressions to mine the logging statements. As most logging statements span over multiple lines, we ensure that we capture entire logging statements. Lastly, the output of this function is a list of discovered logging statements.

4.4.2 `mine_variables()`

For the analysis of representative fields in rich structured information, we create a function responsible with mining all the variables found in a given list of software code logging statements. Specifically, it discovers variable names present in logging statements. For example, for the logging statement: `"LOG.info("Call " + compCode);"` it extracts `"compCode"`. Thus, the output of this function is a list of discovered variables.

4.4.3 `map_runtime_to_logging_statements()`

For mapping runtime logs to logging statements, we tokenize logging statements, and leverage this representation for discovering potential candidates. Specifically, for each runtime log, we check for matching all tokens of a particular logging statement, and subsequently, based on heuristics, decide if the runtime log belongs to the underlying logging statement or not. Below, in **Algorithm 1**, we list the steps followed for mapping a runtime log to its underlying logging statement.

Algorithm 1 Procedure for mapping runtime logs to their underlying logging statements

```
1. Discover raw logging statements in a software code repository
2. Process the discovered raw logging statements (remove syntax)
3. Tokenize the transformed raw logging statements
4. Remove META information from runtime log messages and tokenize
5. Check for match between logging statements' tokens and current runtime logs
6. Choose match (if applicable) from a potential list of candidates
```

RQ4.2. A dataset that operates in the log parsing paradigm can be generated by creating a tool that automatically creates a mapping between logging statements and runtime logs. Specifically, by mining the logging statements in a given software repository and by using the respective runtime logs, using a series of transformations, a mapping between these two can be found, and then appended as an entry to the dataset.

4.5 Generalisability of the entity parsing dataset

As demonstrated in the previous chapter, combining various homogeneous data (generated by standalone applications) results in obtaining similar properties to industry data. Subsequently, we apply a similar strategy to construct the *Entity* dataset, namely to combine logs from the systems analyzed. In Figure 4.9, Figure 4.10, and Figure 4.11 we visualise how the proposed *Entity* dataset clusters together with the previously analyzed log datasets, and as the datasets considered in the previous analysis[¶] contain *2k* entries, we subsequently match that number by randomly sampling *2k* entries from the *Entity* dataset. For sampling, to enable reproducibility, we use a `random_state=1`, for `pandas.DataFrame.sample`.

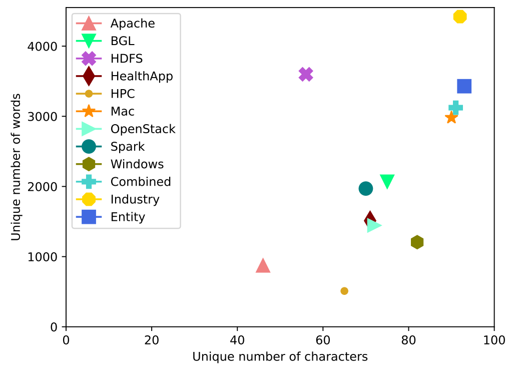


Figure 4.9: Unique number of log lines' character length versus unique number of characters.

We observe that in terms of the *unique number of words*, *unique number of characters*, *unique number of log lines' character length*, the dataset is closely aligned with the *Industry* and *Combined* datasets, confirming once again that combining data from various systems resulting in obtaining similar properties to industry data. To understand the differences, in Table 4.4 we display the measurements.

Table 4.4: Statistics for the log dataset analyzed, with the addition of the *2k* randomly sampled entries from the *Entity* dataset.

Dataset	Apache	BGL	HDFS	HealthApp	HPC	Mac	OpenStack	Spark	Windows	Combined	Industry	Entity
Word level	874	2068	3599	1512	510	2981	1445	1970	1206	3123	4421	3429
Character level	46	75	56	71	65	90	72	70	82	91	92	93
Log length level	9	114	59	55	50	186	50	63	66	157	181	181

[¶]See Chapter 3, section 3.5.3.

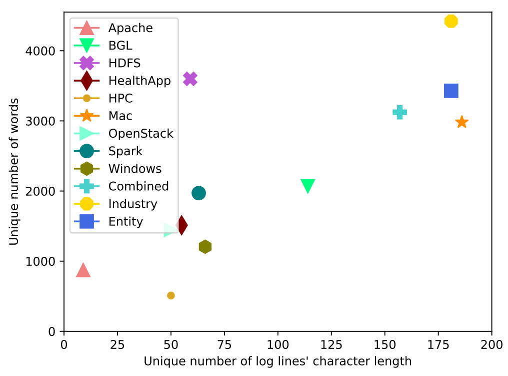


Figure 4.10: Unique number of characters versus unique of words.

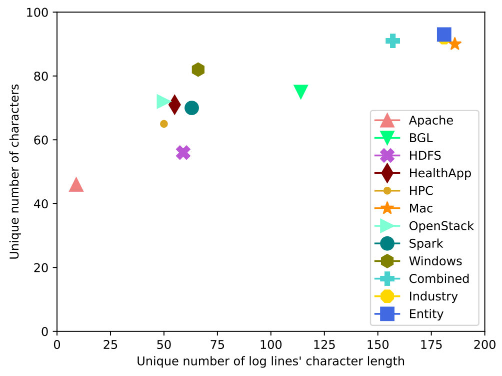


Figure 4.11: Unique number of log lines' character length versus unique number of words.

RQ4.3. Modern systems subsume a tremendous amount of software components that generate heterogeneous log data. During the log collection phase, these systems centralize a variety of log formats, which are then subjected to analysis to distil critical insights. Consequently, creating a dataset that is representative of industry implies using a similar strategy, namely centralizing logs from various systems. As a consequence, the resulting data is heterogeneous, confirmed by the rather high *number of unique characters*, *unique number of words*, and *unique number of log lines' character length*, compared to public and industry data. Thus, we would expect that if more and more logs produced by different systems would be added to the dataset, its generalisability would increase, in turn enabling training on more and more log formats, thus increasing possible methods' robustness.

4.6 EntityLog - A method for tackling entity parsing

Entity parsing transforms raw logs into *rich structured information*, facilitating input for downstream tasks. However, (1) log data heterogeneity and (2) the tremendous data volume generated make it challenging for creating accurate and scalable *entity parsing* solutions. Nevertheless, for drawing actionable insights in real world scenarios, solutions need to move away from hard-coded rules and heuristics, while parsing input data in real time for downstream tasks. To contribute toward creating such solutions, we study the performance of three machine learning models on the data gathered in the previous chapter, in terms of accuracy and scalability.

We tackle *entity parsing* as a Named Entity Recognition (NER) [46] task. NER is typically applied in natural language processing, where supervised models are able to recognize named entities in unstructured text. Although logs are not natural language, they still need

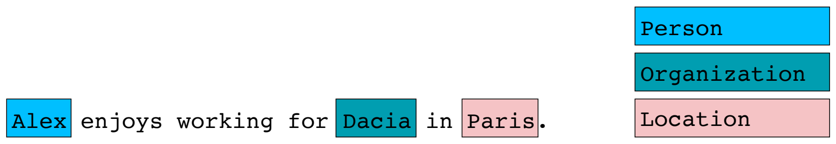


Figure 4.12: NER in natural language.

to be interpreted by humans, and their structure resembles it (natural language) closely, thus making it feasible to formulate the *entity parsing* as NER task. In Figure 4.12 we provide an example of NER's capabilities, namely identifying a predefined set of entities in unstructured text, and in Figure 4.13 we display the *EntityLog*'s[‖] workflow, namely transforming raw logs into *rich structured information*. Instead of accounting for entities that are usually present in natural language, such as names, locations, organizations, country, etc. we account for the entities of interest in software systems. Thus, we leverage the *Entity* dataset which allows for recognizing five entity types, namely `Generic Type`, `Path`, `Id`, `File`, `Priority`. Consequently, we consider three machine learning models that are used usually in NER, and train them to solve *entity parsing*. Subsequently, we evaluate their performance in terms of accuracy and scalability. 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)

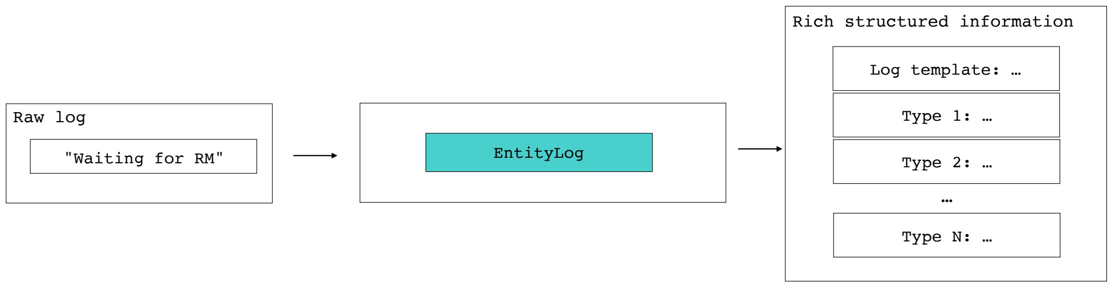


Figure 4.13: EntityLog's workflow. Each incoming raw log is parsed into *rich structured information*.

[‖]Code for *EntityLog* is available at https://github.com/spetrescu/entity-log

4.7 EntityLog's performance

We implemented three machine learning models, namely a *Linear Chain CRF*, a shallow *Long short-term memory* (LSTM) neural network, and a shallow *Bidirectional LSTM*. The code for these models has been made publicly available[**][††][‡‡]. We visualize the architecture of the shallow LSTM model in Figure 4.14, and in Figure 4.15 we visualize the architecture of the shallow Bi-LSTM model.

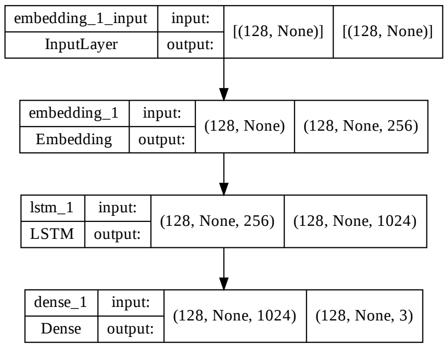


Figure 4.14: Shallow LSTM architecture used in our experiments.

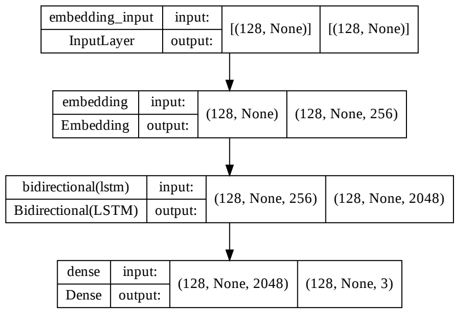


Figure 4.15: Bi-LSTM architecture used in our experiments.

[**]Code for the LSTM model is available at https://github.com/spetrescu/lstm-ner-log-parsing
[††]Code for the Bi-LSTM model is available at https://github.com/spetrescu/bi-lstm-ner-log-parsing
[‡‡]Code for the CRF model is available at https://github.com/spetrescu/crf-ner-log-parsing

In Table 4.5 we display the results of our experiments. We observe that the CRF model obtains better results in terms of accuracy, compared to the LSTM and Bi-LSTM models. However, it scales poorly in comparison to the other two models.

Table 4.5: Models' performance. We used an 80/20 train/test split. Batch size is 128, except for the CRF model as it cannot do batch inference due to its Viterbi decoding.

Model	Accuracy	Throughput
LSTM	0.64	13890 pred/s
Bi-LSTM	**0.77**	**7172 pred/s**
CRF	0.94	100 pred/s

We consider the Bi-LSTM model to be the "winner", as it obtains a better accuracy than the LSTM model, while still being able to process a relatively high number of logs. Specifically, when evaluated on the *Entity* dataset, with a 80/20 train/test split, the chosen model achieves an accuracy of 77% in terms of correctly identified entities, and the model's throughput is 7172 predictions per second. Thus, for using *EntityLog* in practice, one can either use the already trained model, or fine tune by providing more data. In comparison to how log parsing methods were previously evaluated, namely without a train/test split, we address this and test on unseen data that was not used during training. Consequently, in terms of the generalizability of our results, we believe that estimates for solving *entity parsing* are strictly dependent to the data used for training.

4.8 Discussion.

Log parsing methods perform poorly, which makes their output unusable for downstream tasks. Additionally, log parsing is not designed to generate templates that contain variable types, which is something desirable in practice. To address this, we propose a novel paradigm, within which the goal is to (1) identify entity types, whilst (2) generating high quality templates. Entity types are defined by leveraging variables found in logging statements, in various software repositories. Subsequently, we construct a dataset by mapping logging statements to their respective runtime logs. This dataset – the *Entity* dataset – can serve as benchmark for training and evaluating future methods that wish to operate within this paradigm. Specifically, the dataset is comprised of entries that contain (1) the runtime log, (2) the ground truth logging statement, (3) the entity log template (that replaces variables with specific entity types), and (4) an array of `("variable's name in logging statement", "variable's entity type")` tuples.

Subsequently, as parsing log data within modern systems is infeasible using hard coded rules and heuristics, it is desirable to parse logs using a method that is robust to logs' heterogeneity. To do so, we propose a method that leverages a neural network architecture to parse logs, and that has the ability to operate within the *entity parsing* paradigm, hence able to differentiate between different types of variables in log messages in a general sense. To decide on which model to use for the proposed method – *EntityLog*, we tested three models, namely an LSTM, an Bi-LSTM, and a CRF. Subsequently, we chose the one that yielded the best performance in terms of accuracy and scalability (throughput, namely predictions per second), as we have leveraged the dataset created in the previous chapter to train and evaluate the model. Specifically, after running experiments, we chose the Bi-LSTM, as it was able to obtain the best performance in terms of accuracy and scalability, namely 77% accuracy and 7172 predictions per second.

To summarize, in this chapter, we have presented the *entity parsing* paradigm and have contributed by answering research questions regarding (1) how to consider types of log template variables that are general enough and also specific enough, (2) how can a dataset that operates within this paradigm be generated, (3) how generalizable is the dataset, (4) how can *entity parsing* be tackled by evaluating three machine learning models in terms of accuracy and scalability.

5 Discussion & Future Work

In this work we have discovered research gaps within the field of log parsing, and proposed solutions to address these. In the following sections we (1) summarize our contributions, (2) provide answers to the research questions that were tackled throughout this manuscript, and (3) discuss future work.

5.1 Discussion

The amount of data generated within modern software systems is tremendous, making it infeasible to rely only on manual analyses. Thus, implementing automated log analyses is necessary to distil actionable insights from these low signal-to-noise ratio data. Thus, to contribute toward tackling automating the log analysis processes, we have looked at one of its most important steps, namely log parsing. We have investigated the various ways in which it has been approached, and evaluated the 14 most recognized log parsing methods found in literature. In the process, we have discovered that the field has adopted incomplete evaluation metrics, and in turn we have considered metrics that address the limitations of the former. Furthermore, in an attempt to apply log parsing within the infrastructure of large international financial institution, we have conducted an investigation of how log parsing methods perform on industry data. As a consequence, we have discovered that log parsing is not ready for industry, as its paradigm is not aligned with industry's requirements. To address this, we have proposed a novel log parsing paradigm, namely *en-*

tity parsing, which aims to identify entities in log messages, toward generating structured information from raw logs. Subsequently, we have constructed a dataset intended to serve as a benchmark for the field, for training and evaluating possible entity parsing methods. Lastly, we have proposed a method able to operate in the *entity parsing* paradigm, designed to move away from hand-crafted rules and heuristics. Specifically, we have proposed *EntityLog*, a method that uses a Bi-LSTM to transform raw logs into structured information, able to obtain an accuracy of 77% (for identifying entities) and a throughput of 7172 predictions per second.

To summarize our conclusions, we present the answers to the research questions tackled throughout this manuscript.

RQ1: How has log parsing been approached?

We discover that in the literature there are currently 25 methods that propose solving log parsing. We observe that these can be categorised in two main branches, online and offline methods. By studying the trends in the log parsing literature, we outline the most preferred underlying algorithms for log parsing methods, namely clustering and frequent pattern mining.

RQ2: What is the goal of log parsing and how is that evaluated in literature?

To remove the ambiguity around the goal of log parsing once and for all, we define log parsing as the task of identifying log templates in a runtime log message, to extract the static parts that were present in the original software logging statements. Specifically, log parsing structures raw log data into (1) the underlying log templates corresponding to the static part of the logging statements in the software, (2) their respective parameters corresponding to the dynamic part of the logging statements, and (3) log meta information.

However, in the literature, evaluation metrics have a somewhat different focus. Specifically, instead of evaluating the quality of the output produced by log parsing, the main trend is to evaluate log parsing's ability to classify logs. Thus, in the literature there is a misalignment between log parsing's goal and the way it is currently evaluated.

RQ2.1: How can log parsers be evaluated to best align with the goal of log parsing?

To ensure that evaluation reflects the goal of log parsing, we consider two evaluation metrics that align best with the goal of log parsing, namely *log template accuracy* and *edit-distance*. *Log template accuracy* is defined as the ratio of the number of correctly parsed logs, over the total number of logs, and *edit-distance* is defined as the Levenshtein distance. *Log template accuracy* and *edit-distance* aim to asses the quality of templates generated, as they are intended to compare parsed logs against their respective ground truth templates. Consequently, this aligns perfectly with the goal of log parsing. Finally, we argue that it is necessary for the field to adopt metrics based on textual similarity, as they quantify the ability of parsers to produce templates that are evaluated against the ground truth labels, thus providing estimates for the quality of the output generated.

RQ2.2: What is the performance of log parsers on publicly available data?

Our findings indicate that parsers are not able to generate templates that match their respective ground truth templates, and compared to the previous estimates in the field, we discover that the actual performance differs by a large margin, making applying current log parsing methods impractical. We observe *NuLog* as being the only exception, an approach that proved to be robust to the various datasets considered for the experiments, compared to the rest of the methods. We argue that this is a consequence of being intrinsically designed to move away from hand-crafted rules, which is something that the field should adopt moving forward.

RQ3: What is the performance of log parsing methods in the context of modern software ecosystems?

The performance of log parsing methods in the context of modern software ecosystems is subpar, and in the following paragraphs we outline an error analysis of the results. s **Data heterogeneity/diversity.** One of the factors that generated the poor results is data heterogeneity. Specifically, we find data diversity in modern systems to be a significant issue for parsers, and makes them unable to identify underlying templates correctly. We discover that the similarity between the industry dataset and the combined dataset is higher than the similarity between the industry dataset and the individual homogeneous datasets. Specifically, the properties of the data found in the industry dataset are very similar with the properties of the combined dataset, as logs originate similarly from different data distributions (systems), as a consequence of being centralized. In comparison to the combined dataset, the log diversity found in the industry dataset is higher, but the properties of the dataset are intrinsically the same (clusters of log data generated by different systems). Compared to the individual datasets, the log diversity found in the industry dataset is incomparably higher, as it is generated by a incomparably larger number of software components. Consequently, this makes it extremely difficult for parsers to discover the underlying templates on industry data, which is reflected in the *log template accuracy* results. Thus the problem is arguably harder than expected from the results obtained on the combined dataset. In terms of *edit-distance* we observe a drop in performance, which can also be attributed to the aspects discussed in the next paragraphs.

Jargon and high information denseness. Another factor that hinders parsers' performance is the jargon present in data, and the high information denseness. Compared to publicly available data, for a production log, templates and parameters are hard to separate and identify. For example, parameters can contain various alphanumeric characters, but also symbols that make it hard for parsers to generate templates that match the corresponding ground truth labels. Specifically, parsers rely on heuristics that prove not working on industry data. Additionally, another troublesome aspect is the way in which log messages are cascaded. For example, it might be that an error occurs on a specific infrastructure resource, which then sends that information to other resources which concatenate and display similar information. In these cases, parsers would have to be robust against nested templates.

Labeling industry log data. Creating labeled industry data is a hard task, which in turn affects negatively estimating the performance, as subjectivity is added during the labeling process. Due to data heterogeneity and jargon, labeling industry log data is a harder task than labeling publicly available data, even for experts, as getting access to the ground truth code logging statements is something very difficult and, in some cases, even impossible. Specifically, we found that the biggest issue with labeling industry log data is that its nature makes deciding between what is a constant and variable in a log message very hard. Most messages are formatted in ways that generate ambiguity when identifying the templates. For example, messages can contain many concatenated symbols, or have duplicated text concatenated, which makes it difficult to decide on a ground truth.

RQ4: How is the log parsing paradigm aligned with industry goals, and how can that potentially be improved?

Log parsing is not aligned with what is desirable in industry, namely abstracting away from raw logs toward structured information. Specifically, the claim according to which log parsing transforms raw logs into structured information is misleading, as its output fails to provide actual structure, being unable to separate between the various entities present in log messages, and rather only being able to differentiate between their static and variable textual content. For example, log parsing is not able differentiate between IP addresses, integers, software components' identifiers, etc., toward mapping entities that are of interest in log messages. In industry, it is desirable to account for the differences between these, as this (1) generates a feature-rich representation of the input for downstream tasks, and (2) may lead to acquiring critical insights from the logs. Thus, log parsing is not aligned with what is necessary in practice, namely abstracting away from raw data toward structured information. As a consequence, its output is incomplete, which makes its applicability questionable.

Improving the current paradigm means addressing the limitations of log parsing's output. Specifically, improving log parsing means generating *rich structured information* output, transforming raw logs into a structure that contains key-value pairs that correspond to predefined fields of interest for a particular system, and their respective values matched in runtime logs. We define this way of transforming raw logs into structured information as *entity parsing* – a paradigm in which the focus is to identify various entities/fields of interest, in turn generating *rich structured information*.

RQ4.1: How to discover representative entity types for variables present in logging statements?

To discover representative entity types for variables present in logging statements, the variables of seven software code repositories have been analyzed. Specifically, we have tackled the problem of finding entity types by leveraging Affinity Propagation. We consider five variable types, namely *Generic Type*, *Path*, *Id*, *File*, *Priority*), and assume these to generalize across other systems.

RQ4.2: How can we generate a dataset that operates in the entity parsing paradigm?

A dataset that operates in the log parsing paradigm can be generated by creating a tool that automatically creates a mapping between logging statements and runtime logs. Specifically, by mining the logging statements in a given software repository and by using the respective runtime logs, using a series of transformations, a mapping between these two can be found, and then appended as an entry to the dataset.

RQ4.3: How to create an entity parsing dataset that is representative of industry?

Modern systems subsume a tremendous amount of software components that generate heterogeneous log data. During the log collection phase, these systems centralize a variety of log formats, which are then subjected to analysis to distil critical insights. Consequently, creating a dataset that is representative of industry implies using a similar strategy, namely centralizing logs from various systems. As a consequence, the resulting data is heterogeneous, confirmed by the rather high number of *unique characters*, *unique number of words*, and *unique number of log lines' character length*, compared to public and industry data. Thus, we would expect that if more and more logs produced by different systems would be added to the dataset, its generalisability would increase, in turn enabling training on more and more log formats, thus increasing possible methods' robustness.

RQ4.4: How to tackle entity parsing to provide an accurate and scalable solution?

Entity parsing can be tackled by formulating it as a We tackle as a Named Entity Recognition (NER) task. NER is typically applied in natural language processing, where supervised models are able to recognize named entities in unstructured text. Although logs are not natural language, their structure resembles it closely, thus making it feasible to formulate the *entity parsing* as NER task. Instead of accounting for entities that are usually present in natural language, such as names, locations, organizations, country, etc. we account for the entities of interest in software systems. Thus, we leverage the *Entity* dataset which allows for recognizing five entity types, namely `Generic Type`, `Path`, `Id`, `File`, `Priority`. Consequently, for discovering the best solution in terms of accuracy and scalability, we consider three machine learning models that are used usually in NER, and train them to solve *entity parsing*. Subsequently, the best performance is obtained by a Bi-LSTM model, namely 77% accuracy and 7172 predictions per second.

5.2 Future work

Given that we are moving toward a more and more digital world, systems complexity will continue to increase, thus requiring improving the current ways of automating the log analysis process.

For future work, we believe that is important to consolidate the *Entity* dataset created, by (1) augmenting the existing data, and (2) analyzing more systems to create more mappings from runtime logs to underlying logging statements. Furthermore, we believe that more functionalities could be added to our proposed tool, for example enabling the possibility of removing specific entities from training data, or enabling data augmentation processes for training data. Moreover, it would be beneficial to investigate other machine learning models, for optimizing the accuracy and scalability results of our method, thus investigating other potential solutions that move away from hard-coded rules and heuristics. Additionally, we believe that it would be beneficial to test the paradigm in various industry scenarios, to investigate if there are any additional challenges ahead for generating structure from raw logs. Lastly, it would be interesting to test with other data representations, for example, discovering ways in which raw logs can be represented, that might be of more value to downstream tasks, compared to only extracting fields of interest.

Bibliography

[1] Agrawal, A., Karlupia, R., & Gupta, R. (2019). Logan: A distributed online log parser. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (pp. 1946–1951).

[2] Ankerst, M., Breunig, M. M., Kriegel, H.-P., & Sander, J. (1999). Optics: Ordering points to identify the clustering structure. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99 (pp. 49–60). New York, NY, USA: Association for Computing Machinery.

[3] Dai, H., Li, H., Shang, W., Chen, T.-H., & Chen, C.-S. (2020). Logram: Efficient log parsing using n-gram dictionaries.

[4] Devlin, J., Chang, M., Lee, K., & Toutanova, K. (2018). BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

[5] Di Francesco, P., Lago, P., & Malavolta, I. (2019). Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, 150, 77–97.

[6] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). *Microservices: Yesterday, Today, and Tomorrow*, (pp. 195–216). Springer International Publishing: Cham.

[7] Du, M. & Li, F. (2019). Spell: Online streaming parsing of large unstructured system logs. *IEEE Transactions on Knowledge and Data Engineering*, 31(11), 2213–2227.

[8] El-Masri, D., Petrillo, F., Guéhéneuc, Y.-G., Hamou-Lhadj, A., & Bouziane, A. (2020). A systematic literature review on automated log abstraction techniques. *Information and Software Technology*, 122, 106276.

[9] Fu, Q., Lou, J.-G., Wang, Y., & Li, J. (2009). Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 Ninth IEEE International Conference on Data Mining* (pp. 149–158).

[10] Gholamian, S. & Ward, P. A. S. (2021). A comprehensive survey of logging in software: From logging statements automation to log mining and analysis. *CoRR*, abs/2110.12489.

[11] Hamooni, H., Debnath, B., Xu, J., Zhang, H., Jiang, G., & Mueen, A. (2016). Logmine: Fast pattern recognition for log analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16 (pp. 1573–1582). New York, NY, USA: Association for Computing Machinery.

[12] He, P., Zhu, J., He, S., Li, J., & Lyu, M. R. (2018). Towards automated log parsing for large-scale log data analysis. *IEEE Transactions on Dependable and Secure Computing*, 15(6), 931–944.

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

[14] He, S., He, P., Chen, Z., Yang, T., Su, Y., & Lyu, M. R. (2021). A survey on automated log analysis for reliability engineering. *ACM Comput. Surv.*, 54(6).

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

[16] Huang, S., Liu, Y., Fung, C., He, R., Zhao, Y., Yang, H., & Luan, Z. (2020). Paddy: An event log parsing approach using dynamic dictionary. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium* (pp. 1–8).

[17] Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., & Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3), 24–35.

[18] Jiang, Z. M., Hassan, A. E., Flora, P., & Hamann, G. (2008). Abstracting execution logs to execution events for enterprise applications (short paper). In *2008 The Eighth International Conference on Quality Software* (pp. 181–186).

[19] Kobayashi, S., Fukuda, K., & Esaki, H. (2014). Towards an nlp-based log template generation algorithm for system log analysis. In *Proceedings of The Ninth International Conference on Future Internet Technologies*, CFI '14 New York, NY, USA: Association for Computing Machinery.

[20] Lewis, J. & Fowler, M. (2014). Microservices: a definition of this new architectural term. *MartinFowler. com*, 25, 14–26.

[21] Liu, J., Zhu, J., He, S., He, P., Zheng, Z., & Lyu, M. R. (2019). Logzip: Extracting hidden structures via iterative clustering for log compression.

[22] Liu, Y., Zhang, X., He, S., Zhang, H., Li, L., Kang, Y., Xu, Y., Ma, M., Lin, Q., Dang, Y., Rajmohan, S., & Zhang, D. (2022). UniParser: A unified log parser for heterogeneous log data. In *Proceedings of the ACM Web Conference 2022*: ACM.

[23] Maier, D. (1978). The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2), 322–336.

[24] Makanju, A., Zincir-Heywood, A. N., & Milios, E. E. (2012). A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering*, 24(11), 1921–1936.

[25] Messaoudi, S., Panichella, A., Bianculli, D., Briand, L., & Sasnauskas, R. (2018). A search-based approach for accurate identification of log message formats. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18 (pp. 167–177). New York, NY, USA: Association for Computing Machinery.

[26] Mi, H., Wang, H., Zhou, Y., Lyu, M. R.-T., & Cai, H. (2013). Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 24(6), 1245–1255.

[27] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. In Y. Bengio & Y. LeCun (Eds.), *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*.

[28] Mizutani, M. (2013). Incremental mining of system log format. In *2013 IEEE International Conference on Services Computing* (pp. 595–602).

[29] Nagappan, M. & Vouk, M. A. (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)* (pp. 114–117).

[30] Nedelkoski, S., Bogatinovski, J., Acker, A., Cardoso, J., & Kao, O. (2020). Self-supervised log parsing. *CoRR*, abs/2003.07905.

[31] Ning, X., Jiang, G., Chen, H., & Yoshihira, K. (2013). Hlaer : a system for heterogeneous log analysis.

[32] Oprea, A., Li, Z., Yen, T.-F., Chin, S. H., & Alrwais, S. (2015). Detection of early-stage enterprise infection by mining large-scale log data. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (pp. 45–56).

[33] Sedki, I., Hamou-Lhadj, A., & Mohamed, O. A. (2021). AWSOM-LP: an effective log parsing technique using pattern recognition and frequency analysis. *CoRR*, abs/2110.15473.

[34] Shima, K. (2016). Length matters: Clustering system log messages using length of words.

[35] Studiawan, H., Sohel, F., & Payne, C. (2018). Automatic log parser to support forensic analysis. In *16th Australian Digital Forensics Conference*.

[36] Studiawan, H., Sohel, F., & Payne, C. (2020). Automatic event log abstraction to support forensic investigation. In *Proceedings of the Australasian Computer Science Week Multiconference*, ACSW '20 New York, NY, USA: Association for Computing Machinery.

[37] Tang, L., Li, T., & Perng, C.-S. (2011). Logsig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11 (pp. 785–794). New York, NY, USA: Association for Computing Machinery.

[38] Tao, S., Meng, W., Chen, Y., Zhu, Y., Liu, Y., Du, C., Han, T., Zhao, Y., Wang, X., & Yang, H. (2021). Logstamp: Automatic online log parsing based on sequence labelling. *Interface*, 19(03), 03.

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

[40] Vaarandi, R. (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)* (pp. 119–126).

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

[42] Wikipedia contributors (2022a). Conditional random field — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Conditional_random_field. [Online; accessed 12-July-2022].

[43] Wikipedia contributors (2022b). Jaccard index — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Jaccard_index. [Online; accessed 12-July-2022].

[44] Wikipedia contributors (2022c). Levenshtein distance — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Levenshtein_distance. [Online; accessed 12-July-2022].

[45] Wikipedia contributors (2022d). Logging (software) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Logging_(software). [Online; accessed 12-July-2022].

[46] Wikipedia contributors (2022e). Named-entity recognition — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Named-entity_recognition. [Online; accessed 12-July-2022].

[47] Wikipedia contributors (2022f). Payment system — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Payment_system. [Online; accessed 12-July-2022].

[48] Wikipedia contributors (2022g). Self-supervised learning — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Self-supervised_learning. [Online; accessed 12-July-2022].

[49] Xiao, T., Quan, Z., Wang, Z.-J., Zhao, K., & Liao, X. (2020). Lpv: A log parser based on vectorization for offline and online log parsing. In *2020 IEEE International Conference on Data Mining (ICDM)* (pp. 1346–1351).

[50] Zawawy, H., Kontogiannis, K., & Mylopoulos, J. (2010). Log filtering and interpretation for root cause analysis. In *2010 IEEE International Conference on Software Maintenance* (pp. 1–5).

[51] Zhao, N., Chen, J., Wang, Z., Peng, X., Wang, G., Wu, Y., Zhou, F., Feng, Z., Nie, X., Zhang, W., Sui, K., & Pei, D. (2020). *Real-Time Incident Prediction for Online Service Systems*, (pp. 315–326). Association for Computing Machinery: New York, NY, USA.

[52] Zhu, J., He, S., Liu, J., He, P., Xie, Q., Zheng, Z., & Lyu, M. R. (2019). Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (pp. 121–130).