A Systematic Literature Review on Automated Log Abstraction Techniques

Article *in* Information and Software Technology · February 2020

DOI: 10.1016/j.infsof.2020.106276

5 authors, including:

El-Masri Diana
Polytechnique Montréal
1 PUBLICATION **2** CITATIONS
SEE PROFILE

Fabio Petrillo
University of Québec in Chicoutimi
58 PUBLICATIONS **258** CITATIONS
SEE PROFILE

Abdelwahab Hamou-Lhadj
Concordia University Montreal
152 PUBLICATIONS **1,513** CITATIONS
SEE PROFILE

Anas Bouziane
Polytechnique Montréal
1 PUBLICATION **2** CITATIONS
SEE PROFILE

A Systematic Literature Review on Automated Log Abstraction Techniques

Diana El-Masri[a,*], Fabio Petrillo[b], Yann-Gaël Guéhéneuc[c], Abdelwahab Hamou-Lhadj[c], Anas Bouziane[a]

[a] *Département de génie informatique et génie logiciel, Polytechnique Montréal, Montréal, QC, Canada*
E-mail: {diana.el-masri,anas.bouziane}@polymtl.ca
[b] *Départment d'Informatique et Mathématique, Université du Québec à Chicoutimi, Chicoutimi, QC, Canada*
E-mail: fabio@petrillo.com
[c] *Department of Computer Science & Software Engineering, Concordia University, Montréal, QC, Canada*
E-mail: {yann-gael.gueheneuc, wahab.hamou-lhadj}@concordia.ca

Abstract

Context: Logs are often the first and only information available to software engineers to understand and debug their systems. Automated log-analysis techniques help software engineers gain insights into large log data. These techniques have several steps, among which log abstraction is the most important because it transforms raw log-data into high-level information. Thus, log abstraction allows software engineers to perform further analyses. Existing log-abstraction techniques vary significantly in their designs and performances. To the best of our knowledge, there is no study that examines the performances of these techniques with respect to the following seven quality aspects concurrently: mode, coverage, delimiter independence, efficiency, scalability, system knowledge independence, and parameter tuning effort.
Objectives: We want (1) to build a quality model for evaluating automated log-abstraction techniques and (2) to evaluate and recommend existing automated log-abstraction techniques using this quality model.
Method: We perform a systematic literature review (SLR) of automated log-abstraction techniques. We review 89 research papers out of 2,864 initial papers.
Results: Through this SLR, we (1) identify 17 automated log-abstraction techniques, (2) build a quality model composed of seven desirable aspects: coverage, delimiter independence, efficiency, system knowledge independence, mode, parameter tuning effort required, and scalability, and (3) make recommendations for researchers on future research directions.
Conclusion: Our quality model and recommendations help researchers learn about the state-of-the-art automated log-abstraction techniques, identify research gaps to enhance existing techniques, and develop new ones. We also support software engineers in understanding the advantages and limitations of existing techniques and in choosing the suitable technique to their unique use cases.

Keywords: Log Abstraction Techniques, Log Analysis, Log Mining, Log Parsing, Software Analysis, Software Log, Systematic literature review, Systematic survey.

1. Introduction

Logs contain a wealth of data that can help software engineers understand a software system run-time properties [1, 2]. However, modern systems have become so large and complex, especially with the emergence of the Internet of Things (IoT) and Cloud computing, that they produce too huge amounts of log data for software engineers to handle manually. Google systems, for example, generate hundreds of millions of new log entries every month, which account for tens of terabytes of log data daily [3, 4]. Also, logs come in different formats, hindering the analyses of their content and making their uses yet more complex [4, 3].

To tackle these problems, software engineers have at their disposable a wide range of Automated Log Abstraction Techniques (ALATs) that they can use to reduce the amount of data to process. These techniques implement different log-abstraction algorithms, designed for various purposes, *e.g.,* performance optimization, information security, anomaly detection, business reporting, resource utilization, or users' profiling [1].

However, there is a gap between industry and academia. First, software engineers are not aware of all existing ALATs developed in academia and the characteristics of their algorithms. To the best of our knowledge, there is no work that presents a comprehensive view on state-of-the-art ALATs and software engineers cannot afford to undertake the cumbersome and time-consuming task of searching through the large body of literature to identify the best suited ALAT. Second, software engineers do not have the time and resources to study and understand the characteristics of each ALAT. The gap is further spread because researchers focus on enhancing accuracy (defined

*Corresponding authors
Email addresses: diana.el-masri@polymtl.ca (Diana El-Masri), fabio@petrillo.com (Fabio Petrillo)

in Section 6) when proposing new ALATs whereas software engineers are also interested in comparing the ALATs in terms of other useful aspects.

To reduce this gap, this paper helps researchers and software engineers as follows:

- It provides a SLR to inform software engineers of existing state-of-the-art ALATs in Section 5
- It collates and combines ALATs' characteristics identified through the SLR into seven desirable quality aspects on which it builds a quality model to evaluate ALATs, explained in Section 6
- It presents a comparison of 17 ALATs according to our quality model, identifies research gaps, and makes recommendations for researchers on future research directions, in Section 7.
- It helps software engineers understand the advantages and limitations of existing ALATs and select the most suitable for their use cases, in Section 7.

We review 89 research papers out of 2,864 initial papers, identified using a SLR, following the guidelines proposed by Kitchenham *et al.* [5, 6]. We selected these papers after searching all the papers related to log analysis in the digital resource Engineering Village. Two authors independently read and evaluated the papers. We performed backward and forward snowballing through SCOPUS. Based on our inclusion/exclusion criteria and quality assessment, we obtained 89 papers, in which we identified 17 unique ALATs.

We evaluated these ALATs and showed that (1) researchers worked on improving the efficiency of ALATs by adopting diverse algorithms, while distributed architectures seem most promising; (2) parameter tuning for large-scale log data is challenging and requires major effort and time from software engineers, researchers should consider techniques for automatic and dynamic parameters tuning; (3) due to confidentiality issues, log datasets are rare in the community while all existing unsupervised ALATs depend on these datasets for training, so we recommend researchers to investigate new ALATs that do not rely on training data; (4) practitioners must make compromises when selecting an ALAT because there is not one ALAT that can satisfy all quality aspects even if online ALATs (*e.g.,* Spell, Drain) or ALATs based on heuristic clustering approaches and implementing a parallelization mechanism (*e.g.,* POP, LogMine) satisfy most combinations of quality aspects; (5) supervised ALATs based on Natural Language Processing techniques (NLP) are accurate if the models are trained on large amounts of data and researchers should build and share their logs to benefit the research community.

He *et al.* [7] provided an ad-hoc comparison of four ALATs using accuracy and efficiency as quality aspects. Also, parallel to this work, Zhu *et al.* [8] measured the performance 13 ALATs on 16 log datasets and reported interesting results in terms of accuracy, robustness, and efficiency. Differently, we conduct a systematic literature review (SLR) from which we identify, study, summarize, and compare 17 ALATs based on seven desirable quality aspects identified from the literature: mode, coverage, delimiter independence, efficiency, scalability, system knowledge independence, and parameter tuning effort.(defined in Section 6). Furthermore, we provide practitioners with direct references and summarize/group the researchers' findings, so practitioners benefit from their experience with ALATs. Our results are based on a thorough review of ALAT development contexts and algorithmic characteristics, detailed in Section 5 and Table 1, and on the results of empirical experiments and experiences shared in the literature. Our results are not based on review of any source code released.

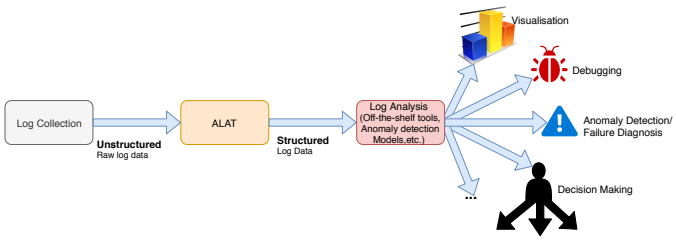

Figure 1: Log Mining Pipeline

The remainder of the paper is as follows. Section 2 provides a background on log abstraction. Section 3 motivates the use of ALATs by practitioners and researchers. Section 4 describes our study design. Section 5 groups and summarizes the 17 state-of-the-art ALATs identified through a SLR. Section 6 presents the ALATs quality model based on seven quality aspects identified in literature. Sections 7 provides the results of our study and promising directions for researchers and software engineers, respectively. Section 8 discusses threats to the validity of our results. Section 9 concludes the paper with future work.

2. Log Mining Process

To perform log-mining tasks, such as failure diagnosis, performance diagnosis, security, prediction, and profiling [1], a typical log-mining process is composed of three steps: log-collection, log-abstraction, and log-analysis (Figure 1).

The raw log data collected during the log-collection step contains log entries describing system states and runtime information. Each log entry includes a message containing a free-form natural-language text describing some event. Based on the log-mining task at hand, the log-analysis step implements the most suitable automated log-analysis technique (*i.e.,* anomaly detection, model inference, etc.), which usually requires structured input-data that can be encoded into numerical feature vectors. As shown in Figure 2, during the log-abstraction step, ALATs transform the raw log-data into structured events lists required by the automated log-analysis techniques. Thus, ALATs are essential in the pre-processing step for efficient

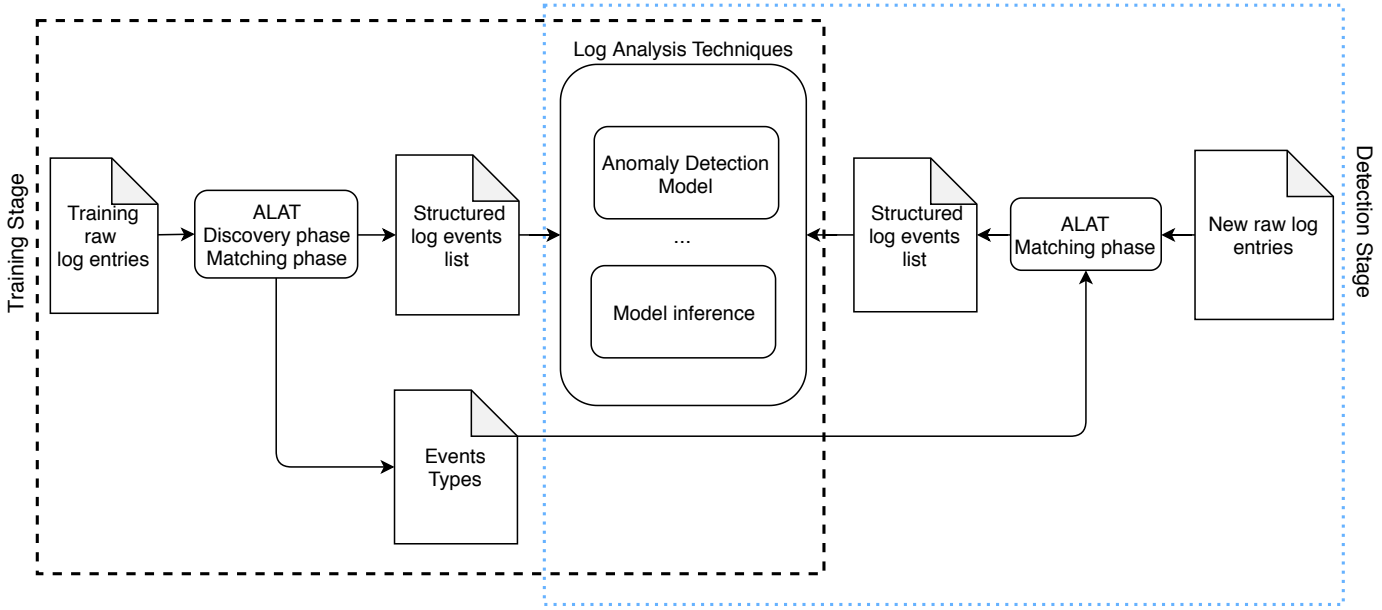

Figure 2: Log Mining Process

log-mining (*e.g.,* searching, grouping, etc.), a foremost step for most automatic log-analysis techniques and a useful step for managing logs in a log management system [9].

2.1. Log Format

Logs are generated by logging statements inserted by software engineers in source code to record particular events and track run-time information. For example, in the logging statement:

```
logger.info("Time taken to scan block pool {} on
{} {}", map.get("pool"), path, executionTime )}
```

`logger` is the logging object for the system, `info` is the chosen verbosity level, `Time taken to scan block pool` and `on` are static messages fixed in the code, which remain the same at runtime, and `poor`, `path`, and `executionTime` are dynamic parameters varying each time this statement is executed, which can thus generate different log entries, such as the example in Figure 3.

Each **log entry** in a raw log-file represents a specific event. As shown in Figure 3, a log entry is generally composed of a log header and a log message containing run-time information associated with the logged event. The logging-framework configuration determines the fields of the log-header. Usually, they include data such as a time-stamp, a severity level, and a software component [10, 11]. Therefore, these fields are structured and can easily be parsed and abstracted.

As illustrated in Figure 4, the **log message** of a log entry is written in a free-form text in the source code, typically as a concatenation of different strings and–or a format string, which is difficult to abstract because it does not have a "standard", structured format. Log messages are composed of static fields and dynamic fields. Dynamic

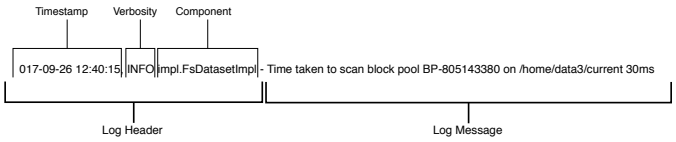

Figure 3: Log Entry Sample

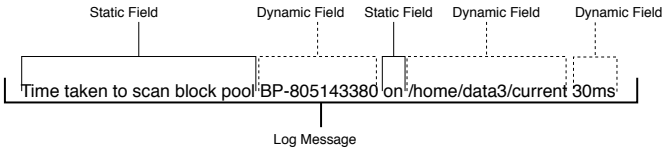

Figure 4: Log Message Fields

fields are the variables assigned at run-time. Static fields are text messages that do not change from one event occurrence to another and denote the event type of the log message. Log fields can be separated by any delimiter *e.g.,* white-space, brackets, comma, semicolon, etc.

2.2. Log Abstraction

Log-abstraction structures and reduces the amount of log entries in the raw log-file while keeping the provided information. The goal of ALATs is to separate the static fields from the dynamically-changing fields, to mask the dynamic fields (usually by an asterisk *), and to abstract each raw log message into a unique event type that is the same for all occurrences of the same event. For example, the log message in Figure 4 could be abstracted by the following event type:

```
Time taken to scan block pool * on * *
```

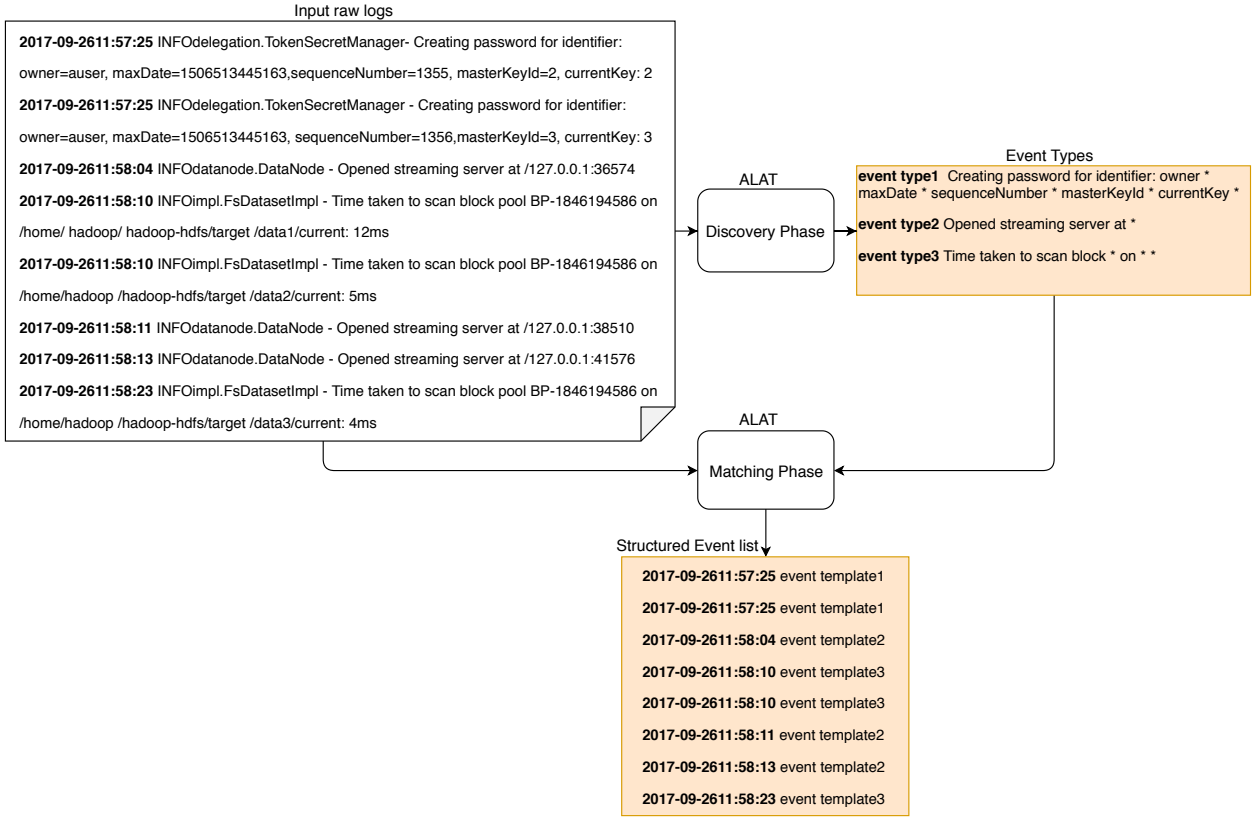

Figure 5: Log-Abstraction Phases

ALATs include two phases: **discovery** and **matching**. As shown in Figure 5, during the discovery phase, ALATs take as input a batch of training raw log entries and output the abstracted event types for all log entries of the same event. Once the event types are generated, they serve as a basis for matching new log entries in batch or stream processing.

2.2.1. Challenges

Abstracting logs for complex and evolving systems requires ALATs to tackle several challenging issues. We now summarise these challenges.

Heterogeneity of Log Data. Log messages have various formats. They are produced by different software layers/components and can be written by hundreds of developers all over the world [3, 9]. Therefore, practitioners may have limited domain knowledge and may not be aware of the original purpose and characteristics of the log-data [3].

Updating of Event Types. Log messages change frequently (*e.g.,* hundreds of logging statements are added in Google systems each month [3]). Practitioners must update event types periodically via the discovery phase to ensure abstraction accuracy for the matching phase [12, 13].

Manual Parameter Tuning. During the discovery phase, practitioners must manually tune ALATs parameters, which is challenging: (1) some are not intuitive and impact the ALATs internal algorithms; (2) others must change according to the systems because each system has different log-data characteristics; and, (3) tuning ALATs parameters on large data is time-consuming. Usually, practitioners tune parameters on a small sample [14], hoping to obtain the same accuracy on large log-files [3].

Log Entries Lengths. Some ALATs, such as Drain, IPLOM, or POP, assume that log messages of the same event type have the same lengths (*i.e.,* number of tokens in their messages). However, log messages of a same type may have different lengths, *e.g.,* `User John connected` (length: 3) vs. `User John David connected` (length: 4) for the type `User * connected`.

2.3. Log-Analysis

Log-analysis is a rich research field. We give a brief overview of some its purposes and their influences on ALATs.

2.3.1. Anomaly Detection

Anomaly Detection analyzes log data (*e.g.,* system logs, security logs) to identify in a timely manner abnormal behaviors that deviate from typical, good behaviors to diagnose failures [15] or security [16] and performance issues [17] and, thus, mitigate their effects [1, 18, 19, 20].

Anomaly detection typically uses machine-learning techniques (supervised, such as SVM and decision tree or unsupervised methods, such as PCA, clustering, and invariant mining), which use as input a numerical feature vector for each event sequence generated from the structured events list provided by an ALAT. Therefore, ALATs are a prerequisite for anomaly detection to provide the structured event lists needed to train the anomaly-detection model and to abstract log entries during the detection [21].

2.3.2. Model Inference

Model inference mines systems logs (*e.g.,* execution logs, transaction logs, events logs) to infer a model of the system behavior (*e.g.,* finite sate machines). The model is then used to detect deviation from the expected behavior and infer the faults that produced the abnormal behaviour. Model inference is useful for understanding complex and concurrent behaviour and predict failures. For example, Beschastnikh *et al.* [22] generated finite state machines to provide insights into concurrent systems. Salfner *et al.* [23] generated Markov models for failure prediction. Therefore, ALATs are a prerequisite for model inference (1) to abstract log messages into structured event lists from which to generate numerical feature vectors and (2) to remove log messages that are irrelevant and–or too frequent, keeping only messages useful to build a model [24, 25].

3. Motivation

Organisations, regardless of their sizes, find log data to be invaluable. They use this data in various ways. However, the log-abstraction components offered in off-the-shelf automated log-analysis tools (*e.g.,* Loggly, Prelert, or Splunk) and open-source automated log-analysis tools (*e.g.,* GrayLog, Logstash) do not satisfy the challenges of modern systems, because they abstract log messages using domain-expert predefined regular expressions and, thus, depend on human knowledge and manual encoding, which are error-prone, non-scalable, and non-evolutive.

In organisations adopting Cloud technology, practitioners have logs coming from logic-tiered servers, multiple Web servers, and database servers. They also have logs generated by Docker containers and other virtual machines. They must treat all these logs as a whole and aggregate them via a log shipper (*e.g.,* Logstash or Apache Flume) to a centralized server where an ALAT and a log-analysis tool are installed. Practitioners managing centralized logs need an ALAT with a strong focus on efficiency, heterogeneity, scalability, and independence from the servers. Furthermore, in organisations adopting continuous software delivery (*e.g.,* Facebook pushes tens to hundreds of commits every few hours), practitioners face streams of log statements being continuously added and updated (*e.g.,* Google systems introduce tens of thousands of new logging statements every month, independent of the development stage [26]). Therefore, they require an ALAT updating its parameters automatically without the need to retrain/retest.

There is a wide range of ALATs among which to choose in the literature. Practitioners should select the ALAT with quality aspects that best suite their unique use cases and–or address the prerequisites of their log-analysis techniques. For example, for anomaly detection, an ALAT must have a high coverage and abstract rare events to avoid false positive [16]. The ALAT should handle the evolving nature of logs and discover/refine event types dynamically without interrupting the anomaly detection process by an offline discovery phase [16, 9]. In contrast, for model inference, an ALAT must allow practitioners to perform a pre-processing step to remove irregular/irrelevant log messages to make their analysis more effective [25, 1, 27]. Furthermore, predictions depend on whether the log granularity used to create the model matches the decision-making granularity and the ALAT must allow practitioners to change it as they see fit [1, 27].

4. Study Design

We follow the guidelines by Kitchenham *et al.* [5, 6] for an SLR. We divide our research method into five main steps: (1) research questions (RQs), (2) search strategy, (3) selection procedure and quality assessment, (4) reporting of the results and answers to the RQs in Section 5 and Section 6, and (5) comparing ALATs to guide software engineers in Section 7.

4.1. Research Questions

We want to answer the following RQs to understand the current state of automated log-abstraction techniques along with the existing challenges. We use the answers to these questions to propose a quality model for evaluating existing techniques and tools.

- RQ1. What are the state-of-the-art automated techniques for log abstraction analysis?
- RQ2. What are these techniques' quality aspects in addition to accuracy?

4.2. Search Strategy

We used papers from conferences and journals, written in English, and published between 2000[1] to 2018. We

[1]We chose to start at the year 2000 because the ALAT SLCT proposed by Vaarandi *et al.* in 2003 represents one of the first log data clustering algorithms [28]. We decided upon a tolerance of 3 years before 2003

conducted the literature search through the digital metalibrary Engineering Village[2] that offers access to 12 engineering literature and patent databases and provides coverage from a wide range of engineering sources including: ACM library; EI Compendex; IEEE library; Inspec-IET; and, Springer.

We conducted the snowballing using Scopus, the largest database of abstracts and citations of peer-reviewed literature[3]. We used Scopus to cover a larger range of papers, combining seed papers from Engineering Village and snowballing papers from Scopus. We searched in the titles, abstracts, and keywords of the papers with the following queries[4]:

```
(("log analysis") WN ALL)
```

and:

```
(("log parsing" OR (log AND "message type")
 OR (log AND "message formats") OR "log message"
 OR ("signature extraction" AND logs)
 OR ("log format") OR "log template"
 OR "log event type") WN ALL).
```

4.3. Literature Selection Procedure

We passed the papers through three stages of screening. The filtering steps are (1) general criteria (language, paper type, time frame, domain category), (2) inclusion and exclusion criteria, and (3) overall quality of the papers.

Inclusion criteria are:

- Paper must be in conference proceeding or journal.
- Paper must be published between 2000 and 2018.
- Paper must be written in English.
- Paper must be on log analysis, log abstraction, log mining, or log parsing.
- Paper must pertain to software engineering.
- Paper must propose, explain, or implement an automated log-analysis technique.

Exclusion criteria are:

- Papers with identical or similar contributions (duplicates).
- Papers not publicly available.
- Papers focusing on end-user experience.
- Papers focusing on logging practices (*i.e.,* how to write logs).
- Papers using off-the-shelf tools (*e.g.,* ElasticSearch, Logstash, Kibana stack (ELK)).
- Papers focusing on log-analysis component architecture (*i.e.,* logging pipeline-architecture).
- Papers requiring access to source code of the system.

Quality assessment answers the following questions:

- Is the paper based on research?
- Is the research method clear enough?
- Is there a description of the context in which the research was carried out?
- Does the proposed method address the objectives set by the researchers?
- Is there an evaluation of the proposed method?

Figure 6 shows our search and selection process, which we detail in the following.

Seed papers. We first performed an automatic search by running our search queries through Engineering Village. The initial search returned 2,864 papers. After filtering these papers based on the inclusion and exclusion criteria, we obtained 121 papers. Then, two of the authors reviewed the titles and abstracts of these papers independently and classified each paper as "include" or "exclude". We collated the results: any papers in disagreement was discussed with all the authors until an agreement was reached. We obtained 31 seed papers.

Candidate papers. We then obtained a set of 738 papers by merging the sets of paper obtained (1) by running the second search string in Engineering Village and (2) by searching in SCOPUS for all papers referencing the 31 seed papers (forward snowballing) and all references in the seed papers (backward snowballing). Two of the authors reviewed independently the titles and abstracts of each of the 738 papers and kept 106 papers. Finally, we grouped these 106 papers and the 31 seed papers into the set of 137 candidate papers.

Selected papers. Independently, two authors read in details the 137 candidate papers. They evaluated each paper based on our inclusion/exclusion criteria and our quality assessment. Again, we collated both authors' decisions and obtained the set of 89 selected papers.

4.4. Data Extraction and Synthesis

Data extraction. Independently, two authors reviewed in detail the 89 selected papers and extracted data regarding:

- State-of-the-art ALATs approaches, algorithms, and techniques.
- Desired ALATs' characteristics/quality aspects, their definitions and classification criteria.

First, the authors compared the data and resolved disagreements by consensus. Then, they collated the data extracted on ALATs characteristics/quality aspects, which they consolidated into seven industry desired quality aspects (*i.e.,* unified the naming, typical question, definition, and classification criteria) to compose our quality model. They also extracted the main results and evaluations of the ALATs in terms of the identified quality aspects.

[2] https://blog.engineeringvillage.com/about

[3] https://www.elsevier.com/solutions/scopus

[4] The full queries are available in the replication package at http://www.ptidej.net/downloads/replications/ist19a/.

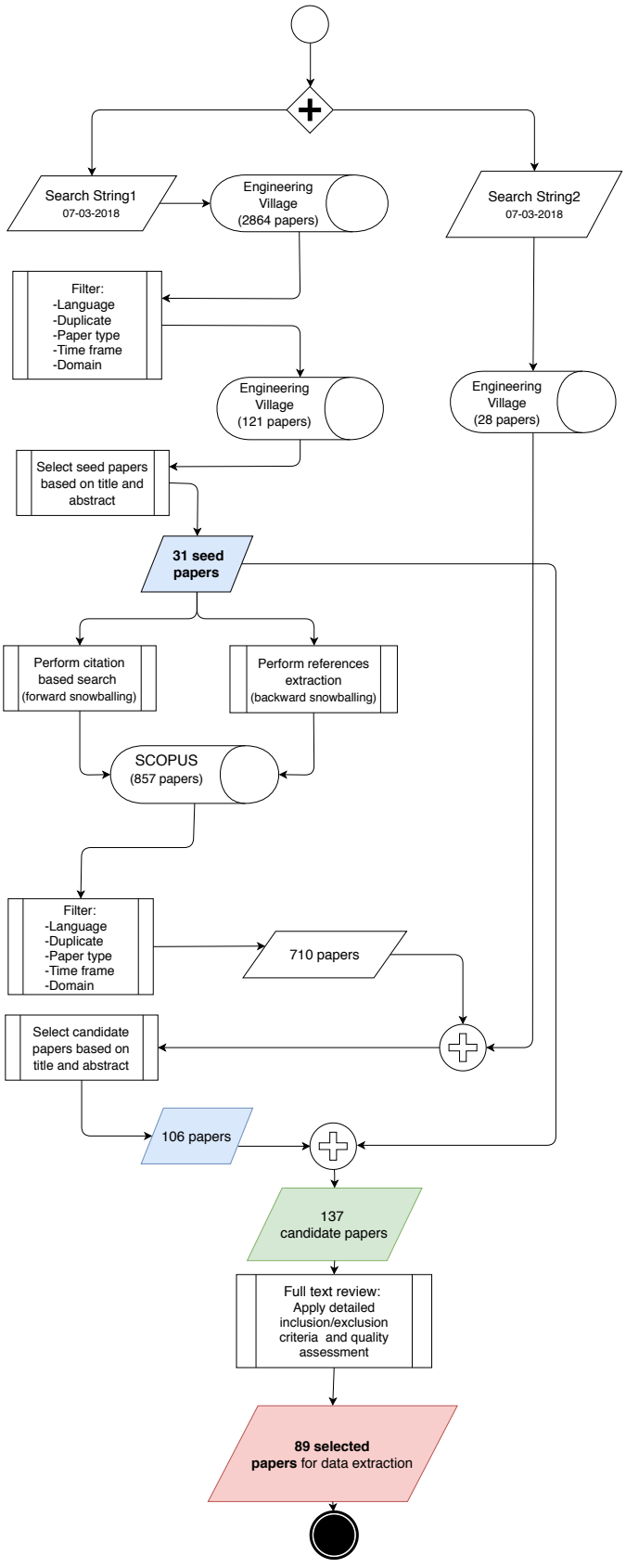

Figure 6: Papers Selection Process

Data Synthesis. We summarized and collated the extracted data. We did not identify inconsistencies within the data. We present the data synthesis and our findings in Section 2 with a catalogue of state-of-the-art ALATs, in Section 6 for their quality aspects, and in Section 7 for their evaluations in terms of the identified quality aspects.

Study Scope. We considered open- and closed-source ALATs used in industry and prototype techniques and tools released by researchers. Only three articles briefly described the matching phase of their proposed ALATs. Therefore, we focused on the discovery phase. Finally, we do not check the source code of the tools to assess the correctness of their implementations, which would require access to all of their source code and extensive resources out of the scope of this paper.

5. Automated Log Abstraction Tools

We now discuss the 17 ALATs that we identified through our SLR explained in Section 4.4. We group these techniques according to the approaches and the algorithms that they adopt for abstracting raw log messages. Figure 7 categorizes the 17 ALATs and Table 1 summarizes key characteristics of their algorithms.

5.1. Online ALATs

Online ALATs abstract raw log entries one after another from streams of entries without any requirement of doing offline processing first [9]. This approach is particularly important for Web services management and system on-line monitoring and processing [4], for which the volumes of logs increase constantly and model training is time consuming using some existing logs [4].

Drain [4] abstracts log messages into event types using a fixed depth parse-tree to guide the log event analysis process, which avoids constructing a profound and unbalanced tree and encodes specially-designed parsing rules in the parse tree nodes. Drain algorithm consists of five steps. During the first step, Drain pre-processes raw log messages using user-defined regular expressions based on domain knowledge to identify and remove trivial dynamic fields (*e.g.,* IP addresses, numbers, and memory). In a second step, Drain assumes that logs with the same event type have the same length (number of tokens) and selects the node corresponding to the log length. For example, with the pre-processed log message *User John connected*, Drain selects a path to a first layer node *length: 3*. In the third step, Drain assumes that tokens in the beginning positions of a log message are more likely to be static fields. It selects the leaf node linked to the second layer node *User*. During the fourth step, Drain calculates the similarity between the log message and the event type of each log group in the leaf node and adds the logID of the log message to the

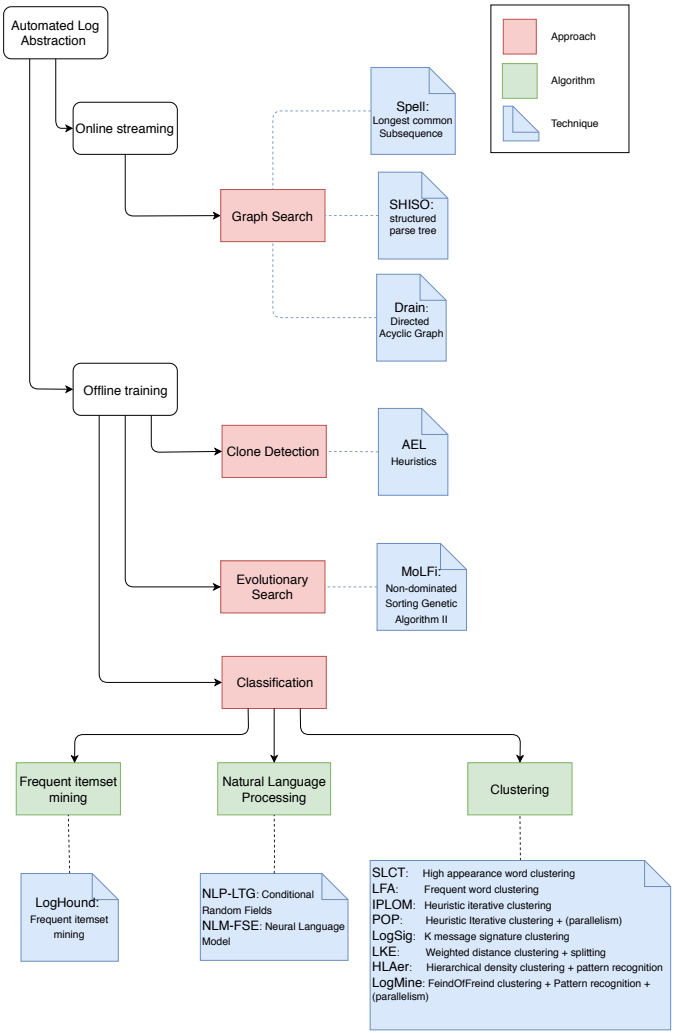

Figure 7: Automated Log Abstraction Techniques.

most suitable log group. Finally, Drain updates the parse-tree by scanning the tokens in the same position of the log message and the event type. He *et al.* enhanced Drain [13] with a new algorithm based on a Directed Acyclic Graph (DAG) to guide the log event analysis process. The DAG also encodes specially-designed heuristic rules and its depth is also fixed to accelerate the parsing process. Another enhancement is the implementation of an automated parameter-tuning mechanism. Drain-DAG can initialize and update its parameters automatically and dynamically according to the incoming log messages. Drain provides an optional mechanism to merge similar log event types in a post-processing step to address the "variable log entry length" cases (detailed in section 2.2) that invalidate the event-size-heuristic assumed in step(2). The time complexity of Drain for the event type search process is $O(n)$ where n is the number of log messages. Its complexity for calculating the similarity between a log message and a candidate event type is $O(m_1 + m_2)$ where m_1 and m_2 are their respective numbers of tokens.

Spell (Streaming Parser for Event Logs using an LCS) [9] is an online streaming ALAT, which convert log messages into event types and parameters. It uses an approach based on the longest common subsequence (LCS). Spell view the output of the logging print statements in the source code as a sequence containing a majority of static fields, and assumes that when two sequences are produced by the same logging print statement, their longest common sequence represents their event type. Therefore, the LCS of the two sequences is likely to be static fields representing an event type. Thus, Spell starts with an empty LCSMap and transforms each incoming log message into a "token" sequence using user-defined delimiters. Then, Spell compares the new token sequence with the LCSseq of all LCSObjects in the LCSMap and adds its ID to the corresponding LCSObjects or creates a new LCSObject if it cannot find a suitable LCSObject. Spell uses a pre-filtering step, with a prefix tree to find if the event type already exists and to prune away candidates. The time complexity of Spell is $O(n)$ where n is the number of raw log messages [13].

SHISO (Scalable Handler for Incremental System log) [12] performs online classification and incremental mining of event types and their parameters. The algorithm builds on the fly a structured tree with a user-defined number of children. First, SHISO splits the new log message into a word list W using common delimiters (*e.g.,* blank characters, "=", ";", etc.) without separating Web addresses and file paths. Second, SHISO creates a new node that has the word list W and puts it in a tree structure. During the third step, SHISO searches for a log format for W. Finally, SHISO adjusts and refines existing formats continuously in real-time. SHISO conducts the search phase for each log entry. It conducts the adjustment phase when it creates a new format or updates an existing format. The time complexity of SHISO is $O(n)$ where n is the number of raw log messages [13]. However, SHISO algorithm only limits the number of children for each node and, thus, might build a very deep and unbalanced tree, which might increase its running time and impact its ability to handle system logs with lots of event types [13].

5.2. Offline ALATs - Discovery Phase

The following offline ALATs use batch processing and need all the log data to be available during discovery [9].

5.2.1. Clone Detection

AEL (Abstracting Execution Logs) [29] assumes that log messages generated by a same event have textual similarities. Thus, abstracting log messages consists of detecting and grouping similar log messages. To abstract log messages, AEL uses clone detection based on a similarity measure. AEL divides in four steps. The anonymization step uses hard-coded heuristics based on system knowledge to identify dynamic fields in the log messages (*e.g.,* IP addresses, numbers, memory) and replaces them with a generic token (`$V`). Then, the tokenization step divides the anonymized log message into different bins according

to their numbers of words and estimated parameters. For example, the log message `creating password, user is John, masterKeyid=2` and the heuristics `word=value` and `is-are-was-were value` lead AEL to transform the log message into:

```
creating password, user is $V, masterKeyid=$V
```

and to put it in the bin $(7, 2)$. Next, the categorization step compares log messages in each bin and abstracts them into execution event. The anonymization step may miss some dynamic fields because it relies on heuristics, *e.g.,* `creating password for user John`. The reconciliation step mitigates these cases by re-examining the event types and merging similar ones. AEL does not require any user-given parameters but uses a heuristic to merge event types, which users may have to re-define for different logs. The time complexity of the AEL algorithm is $O(n)$ where n in the number of raw log messages. However, in the categorization step, the algorithm compares each log message in a bin with all the existing execution events in the same bin. Therefore, its running time depends on the format of the log dataset, which might generate a large bin size.

5.2.2. Evolutionary Search-based

MolFi (Multi-objective Log message Format Identification) [3] is based on the observation that any ALAT should meet two conflicting objectives: (1) abstract event types that match as many log messages as possible (high frequency in matching log messages) and (2) abstract event types that correspond to particular events types (high specificity). These objectives conflict: event types may be too generic or match only a few log messages. Therefore, MolFi uses multi-objective optimization, with frequency and specificity as its objectives. MolFI includes a preprocessing step to filter duplicated log messages, identify trivial dynamic fields using regular expressions (as Drain), tokenize the log messages, and group messages of same lengths. Then, MoLFi generates log event types that meet the two objectives based on the evolutionary search-based approach NSGA-II and a trade-off analysis. MolFi does not require any parameters tuning. The time complexity of Molfi evolutionary search-based algorithm NSGA-II is $O(mn^2)$ where m is the number of objectives and n is the number of raw log messages.

5.2.3. Classification - Frequent Itemsets Mining

LogHound [30, 31] considers event logs as transaction databases and views each log message as a transaction that consists of items. For example, it views the log message `Password created for User John` as a transaction with the items `(Password,1)`, `(for,2)`, `(User,3)`, `(John,4)`, `(created,5)`. Then, it recasts the task of identifying event types as a task of mining frequent itemsets. It implements a breadth-first algorithm similar to the Apriori algorithm, with heuristics to control memory usage and itemsets sizes. First, it identifies frequent items (words) in the log file based on a user-given "support threshold". Second, it considers rare items as dynamic fields. It keeps an itemset containing only frequent items from each transaction. For example, the itemset above is `(Password,1), (for,2), (User,3), (created,5)`. Third, it stores the frequent itemsets in a cache trie using a "cache trie support". Fourth, it builds a reduced itemset trie using correlations among frequent items. Finally, it abstracts event types from the frequent itemsets in the itemset-trie node. The example above corresponds to the event type `Password created for * John`. It is lightweight and requires little resource.

5.2.4. Classification - Clustering

LKE (Log Key Extraction) [32] assumes that log messages printed by the same logging statements in the source code tend to be very similar to one another, while log messages printed by different statements are often quite different. Therefore, it uses clustering techniques to group log messages from the same statements and considers their common part as event types. Similar to AEL, LKE is a mixture of heuristic rules-based approach and hierarchical clustering. Furthermore, LKE algorithm is designed to allow for easy implementation in parallel mode. It consists of four steps. The first step is to mitigate the problem of having clusters of log messages based on similar dynamic fields, it erases trivial dynamic parameters (*e.g.,* URL, IP addresses, etc.) from raw log messages according to predefined empirical rules and obtains raw log key. The second step is to cluster similar raw log key according to a weighted edit-distance, giving more weight to words at the beginning of raw log key and using space as word delimiters. The third step is to repeatedly split the clusters until each raw log key in the same cluster belongs to the same log key. Finally, LKE extracts the common parts of the raw log key from each cluster to generate event types. The time complexity of LKE algorithm in $O(n^2)$ where n is the number of raw log messages [14].

SLCT (Simple Logfile Clustering Tool) [30, 33] views each log message as a data point with words as categorical attributes. It formulates the log-abstraction task as a data clustering task and. It processes log datasets at the word level abd it clusters log data with common frequent words. It does not use a traditional distance-based clustering approach (*e.g.,* LKE) but a density-based clustering algorithm. It differs from LogHound by considering only one word when creating frequent itemsets. SLCT is a three-step process and makes two passes over the data. It makes the first pass over the log file and builds a word vocabulary. It considers the words positions in the log messages and uses user-given threshold s, "support threshold". It then makes a second pass over the log file and, with every log message containing frequent words, it builds a cluster candidate. For example, if the log message is *Password for User John created* and the words *(Password) (for) (User) (created)* are frequent, then the cluster candidate is *Password for User * created* . If the cluster candidate

already exists, then it increments its "support". In a last step, SLCT selects the cluster candidates containing more log messages than s and abstracts them as event types. Log messages outside clusters are outliers reported during an optional data pass. SLCT requires the user-given parameter s. It is prone to overfitting with low "support threshold" values. The time complexity of SLCT is $O(n)$ where n is the number of raw log messages [12, 14].

LFA (Log File Abstraction) [34] is built on the observation that SLCT does not report the event types of all the log messages in the data but report the event types of frequently occurring log messages. The SLCT algorithm discovers dynamic parts in log messages by relying on word frequency across all log lines in the log file, whereas LFA finds clusters within log lines. Hence, LFA can find all event types, not just the ones that occur more than the supported thresholds required by SLCT. LFA also makes two passes over the data. In the first pass, it builds a frequency table that has the number of times a particular word occurs in a particular position in the log line and, in the second pass, it extracts the frequency of each word at each position. Then, LFA determines the frequency threshold and consider constant words in a log message if their frequency is not less than this frequency threshold. The time complexity of LFA algorithm is $O(n)$ where n is the number of words when considering all log messages in the log dataset [34].

LogCluster [15] is a new version of the SLCT algorithm. It addresses two shortcomings of SLCT. First, the SLCT algorithm cannot detect dynamic variables after the last word in an event type. For example, if we have the log message *password for user John authenticated* and *password for user Sam rejected* with a support threshold set to 2. Then SLCT would report the cluster *(password,1), (for,2), (user,3)* as the event type *password for user* although users might prefer *password for user * **. Second, SLCT is sensitive to word position and delimiter noise. For example, if we have the log message *User John Micheal authenticated* SLCT reports the event type *User * * authenticated*, although users would prefer to have the event type *User * authenticated*. Similar to SLCT, LogCluster makes the first pass over the data to identify frequent words based on user-given support threshold s. However, it does not consider the word position in the log message in the first pass. During the second pass, LogCluster extracts all frequent words from the log message and arranges them into a tuple then splits the log file into clusters that contain at least s log messages. Now, all log messages in the same cluster match the same pattern of frequent words and wildcards. Each wildcard has the form *m,n and matches at least m and at most n words. For example, if we have the log messages *User John authenticated* and *User John Micheal authenticated* with a support threshold of 2. Then, LogCluster will create the cluster candidate *(User, authenticated)* and report the event type *User * 1,2 authenticated*. Similar to SLCT, log messages without a cluster are regarded outliers and reported during an optional data pass. Both SLCT and LogCluster make two passes over the data and their time complexity is $O(n)$ where n is the number of raw log messages. However, Logcluster is slower than SLCT due to the simpler cluster candidate generation procedure of SLCT [15].

IPLOM (Iterative Partitioning Log Mining) [31, 35] finds all event types in the log file, not only the frequent ones and it clusters log messages as an entity starting with the entire log data as a single partition. It employs a heuristic-based hierarchical clustering algorithm and discovers event types using a 4-steps iterative partitioning; First, IPLOM assumes that all log messages corresponding to the same event have the same length and then partitions all log messages using an event-size heuristics. Second, it splits each partition using a token position with the highest number of similar words. Third, it uses a bijective relationship considering strong correlation between two tokens at the same position. Finally, it considers the leaf nodes of the hierarchical partitioning as clusters and event type. Finally, IPLOM partitioning of the database is practically a decomposition of the log abstraction problem, which makes IPLOM a good candidate for using parallel processing. The time complexity of IPLOM is $O(n)$ where n is the number of raw log messages [4].

LogSig [36] assumes that the words present in a logging statement are fixed in the source code and do not change. It considers them as signatures of event types. It also assumes that the positions of the fixed words can change because the length of dynamic parameters may vary for a same event (see Section 2.2). LogSig uses a message signature-based algorithm to identify event types using a set of message signatures. LogSig has three steps and requires a number of clusters k. First, it converts log messages into pairs of terms while preserving the order of terms. For example, logSig converts *User John authenticated* to the pairs *(User, John), (User, authenticated), (John, authenticated)*. Second, it creates k random log-message clusters. Then, it iterates and moves log messages among clusters using a local search-strategy. It stops when no log message is moved. Third, it scans every log message in each of the k clusters and selects the terms appearing in more than half of all the log messages in a cluster. Finally, it builds the event types using the most frequent common term pairs. LogSig has an optional domain-specific preprocessing step to improve its performance. LogSig scales linearly with the number of raw log messages $O(n)$. However, its running time also increases linearly with the number of events, which can lead to relatively longer parsing time [7]. Although the time complexity of LogSig is $O(n)$, it must convert each log message into a set of term pairs, which can be time consuming.

HLAer (Heterogeneous Log Analyzer) [28] HLAer assists log abstraction based on a hierarchical clustering approach and pattern recognition. HLAer does not require any specific knowledge about the analyzed system, it does not make any assumption on the word delimiter used in the log entry, and it does not require users to specify these

delimiters. Instead, HLAer tokenizes all the words and special symbols in the log entry by adding the space as the delimiter between them. For example, HLAer tokenizes the log message *Password created, user=john* to *Password created , user = john*. Then, it performs a hierarchical clustering of heterogeneous logs messages based on a pair-wise similarity using the density-based clustering algorithm OPTICS [37]. After finding all the clusters, it performs a bottom-up log pattern recognition within the hierarchical structure to find a pattern that represents the log messages in each cluster. HLAer uses Smith–Waterman algorithm for pattern generation from pairs and the UPGMA [38] to generate a pattern from a set of patterns, thus the event types. HLAer has a $O(n^2)$ memory requirement because it implements OPTICS, an expensive clustering algorithm for large datasets, which calculates the MinPts-nearest neighbors for each object, requiring $O(n^2)$ pair-wise distance calculations for n raw log messages. Also, with HLAer, the time complexity of running UPGMA is $0(n^2 \times l^2)$ where n is the number of log messages and l is the average number of fields in each log message [39]. Ning *et al.* indicate that HLAer might be a good candidate for parallel processing because the OPTICS algorithm can easily be made parallel and the alignment task for each cluster is fully decoupled and thus can run in parallel.

LogMine [39] overcomes some limitations of HLAer. Hamooni *et al.* found HLAer robust to heterogeneity but not efficient enough when abstracting large log files, due to its memory requirement and communication overhead. LogMine uses MapReduce and does not require user input and does not assume any property of the log messages. Similar to HLAer, LogMine can abstract heterogeneous log messages generated from various systems and its algorithm also consists of a hierarchical clustering module and pattern recognition module with one pass over log messages. First, LogMine is similar to HLAer: it tokenizes every word and symbols by adding spaces (or a given delimiter). Then, it applies an optional (yet recommended) pre-processing step to improve efficiency. It uses regular expressions based on domain knowledge to detect a set of user-defined types (*e.g.,* date, time, IP, and numbers). Then, it replaces the real value of each field with its name. For example, it replaces *2018-04-25* with *Date*). Third, it clusters similar log messages using a one-pass version of the friends-of-friends clustering algorithm and exploits several optimization techniques to improve performance. Fourth, it uses a hybrid version of UPGMA [38] to merge log messages in clusters and identify one event type per cluster. Finally, it iterates through Steps 3 and 4 until it reaches the *Max Pattern Limit* provided by the user or until it encounters the event type containing only wildcards. The time complextiy of LogMine is $O(n)$ where n is the number of raw log messages and its memory complexity is $O(number of clusters)$.

POP (Parallel Log Parsing) [14] observes that most ALATs fail to complete in a reasonable time (less than 1 hour) when log data grows to production levels (around 200 million log messages). ALATs are limited by the computing power and memory of a single computer. POP uses a parallel ALAT implemented on top of SPARK. POP is similar to IPLOM. It uses both heuristic rules and a clustering algorithm. POP abstracts log messages in a three-step process with one pass over log messages. First, it pre-processes log messages with pre-defined regular expressions, based on domain knowledge. The second step is similar to IPLOM, POP clusters the log messages based on the event size heuristic. In the third step, POP recursively partitions the clusters based on token position heuristic. This heuristic assumes that if log messages in a same cluster have a same event type then the tokens in the same positions should be similar. Fourth, it leverages log messages in each cluster and generates an event type from each cluster. Finally, to avoid over-parsing (*i.e.,* suboptimal parameter settings) and to mitigate the event size heuristics in Step 2, POP clusters similar groups based on their event types using hierarchical clustering. It merges the groups in the same cluster and generates event types by calculating the Longest Common Subsequence. The time complexity of POP is $O(n)$ where n is the number of raw log messages.

5.2.5. Supervised classification

NLP-LTG (Natural Language Processing–Log Template Generation) [40] considers event template extraction from log messages as a problem of labeling sequential data in natural language. It uses Conditional Random Fields (CRF) [41] to classify words as a static/dynamic part of the log messages. The training data consists of log messages. To construct the labeled data (the ground truth), it uses human knowledge in the form of regular expressions.

NLM-FSE (Neural language Model-For Signature Extraction) [42] trains a character-based neural network to classify static/dynamic part of log messages. It constructs the training model through four layers. (1) The embedding layer transforms the categorical character input into a feature vector. (2) The Bidirectional-LSTM layer [43] allows each prediction to be conditioned on the complete past and future context of a sequence. (3) The dropout layer avoids over-fitting by concatenating the results of the bi-LSTM layer, and (4) the fully connected, feed-forward neural network layer predicts the event template using the Softmax activation function.

6. ALATs Quality Model

Deissenboeck *et al.* [44] state that, in an Assessment Quality Model (AQM), the assessment of the quality aspects (QAs) is either qualitative or quantitative and aspects not directly measurable are described qualitatively. Following our SLR's methodology detailed in Section 4.4, we identified, grouped, and combined characteristics of the ALATs in the literature into seven desirable quality aspects. For example, we identified that Hamooni *et al.*

Table 1: Characteristics of ALATs Discovery Phase Algorithm ("NA" means that the algorithm does not require this characteristic, "NS" means that the ALAT does not specify a solution to efficiently update event types, "MI" means missing information, text in italic highlights information deduced to the best of our understanding and not specifically specified in the reviewed papers.)

ALAT	Pre-processing	Parallel mode	Time complexity	Event types update
MolFi	y	n	$o(n^2)$	*re-train, offline (batch mode)*
AEL	y	n	o(n)	*re-train, offline (batch mode)*
IPLOM	n	n(parallel mode ready)	$o(n)$	*re-train, offline (batch mode)*
POP	y	y	$o(n)$	*re-train, offline (batch mode). Offers solution for new logs detection*
HLAer	n	n(parallel mode ready)	$o(n^2)$	online
LogMine	optional	y	$o(n)$	re-train, offline (batch mode)
LKE	y	n	$o(n^2)$	*incremental; generates new event type*
LogSig	optional	n	$o(n)$	*re-train, offline (batch mode)*
LogHound	n	n	MI	*re-train, offline (batch mode)*
LogCluster	n	n	$o(n)$	*re-train, offline (batch mode)*
LFA	n	n	$o(n)$	*re-train, offline (batch mode)*
SLCT	n	n	$o(n)$	*re-train, offline (batch mode)*
Drain	y	NA	$o(n)$	online
Spell	n	NA	$o(n)$	online
SHISO	n	NA	$o(n)$	online
nlp-ltg	y	NA	$o(n)$	*require re-labelling and re-training*
nlm-fse	n	NA	$o(n)$	*incremental re-training*

[39] defined four desirable properties that ALATs should have: no-supervision, heterogeneity, efficiency, and scalability. While, Jiang *et al.* [29] defined four aspects regarding ALATs: interpretability, system knowledge, effort, and coverage. He *et al.* [4] addressed challenges in tuning the ALATs parameters and the importance of considering the parameters tuning effort. Furthermore, He *et al.* [14] indicated that, although an ALAT might have high accuracy, it also must be robust in handling large log datasets. Mizutani *et al.* [12] indicated the importance of abstracting log message immediately for troubleshooting. Finally, Makanju *et al.* [35] conveyed the importance of having high coverage and discovering rare events.

We now propose an AQM for evaluating the discovery phase of ALATs. Our AQM defines seven desirable quality aspects identified and collated through our SLR: mode, coverage, delimiter dependence, efficiency, scalability, system knowledge dependence, and parameter tuning effort. We provide a typical question, a definition, and classification criteria.

Accuracy determines the capacity of an ALAT to identify correctly the static and dynamic fields of a log message and abstract it to correct event type during the discovery phase [3]. The accuracy of a same ALAT varies according to the log formats and origins. For example, He *et al.* and Zhu *et al.* evaluated various ALATs in terms of their accuracy (F-measure) across the same log datasets generated from 16 systems and showed that some ALATs have high accuracy on certain files but low on others [7, 8]. Practitioners obtain insights from the accuracy values in the literature but must evaluate it on their own log datasets.

6.1. Mode

Typical Question: *Can the ALAT dynamically abstract incoming log messages into event types without prior knowledge obtained from an offline discovery phase?*
Definition: Existing studies [9, 4, 13, 12] define two modes for ALATs: offline and online. Offline ALATs require batch processing (collect logs for a certain time) before abstracting log messages into event types, *i.e.,* applying an offline discovery phase. Then, they use the previously discovered events types to match new incoming logs in batch or stream. Online ALATs abstract log messages into event types on the fly, dynamically updating discovered event types. Online ALATs do not need a batch mode discovery phase.
Classification Criteria: We study the algorithm of each ALAT. If an ALAT does not require a batch discovery phase then we classify it as "*online*". Otherwise, we classify it as "*offline*".
Domain: Offline, online.

6.2. Coverage

Typical Question: *Can the ALAT abstract all input log messages?*
Definition: Coverage indicates the ability of an ALAT to abstract each log message to an appropriate event type [29]. Log entries of event types for troubleshooting and anomaly detection are rare and there is a risk that the

ALAT removes them during the discovery phase if occurring below a threshold. For example, Jiang *et al.* [29] found that many log entries are not abstracted to any event type by SLCT because they do not occur often enough for a frequent pattern to emerge.

Classification Criteria: We study the algorithm of each ALAT, if its algorithm presents a solution to abstract all input log messages, then we classify it as "*all messages*" coverage, else as "*frequent messages*" coverage.

Domain: All messages, frequent messages.

6.3. Efficiency

Typical Question: *Does the ALAT execute the discovery phase in a reasonable running time and reasonable resource utilization?*

Definition: SquaRE [45] defines performance efficiency as the degree to which a system can complete its functions with a running time and resource utilization that meet requirements. Correspondingly, most of the reviewed articles (*e.g.,* [14, 36, 39]) evaluated the efficiency of ALATs by measuring the running time for completing the discovery phase on different log datasets (different formats and sizes). Only few articles, *e.g.,* [39, 35, 30], measured the memory requirement of their proposed ALATs.

Classification Criteria: We reviewed the experiments conducted on each ALAT in the selected articles and extracted the recorded efficiency (running time/memory requirements) of the ALAT wrt. a specific log dataset. We classified an ALAT as having *low* efficiency if it fails to complete the discovery phase on a given log dataset in a reasonable time. We classified an ALAT as having *high* efficiency if it completes the discovery phase on a dataset in a reasonable time. We considered the time that a discovery phase takes as being reasonable if it runs in less than one hour as shown by He *et al.* in [14].

Domain: High efficiency, low efficiency

6.4. Scalability

Typical Question:*Can the ALAT discovery phase handle an increasing large volume of log messages?*

Definition: Scalability is defined in ISTQB [46] as the capability of a software product to be upgraded to accommodate increasing loads. Similarly, Hamooni *et al.* [39] defined the scalability of ALATs as the ability to process increasing large batches of log messages without incurring CPU and memory bottlenecks. Typically, researchers evaluate the performance of their ALATs on small sample log datasets. However, ALATs deployed in production must scale and complete their discovery phase on large log datasets (approximately 200 million log messages [47]) in a reasonable time. For example, IPLOM scaled linearly with the number of log messages on BGL2K and HDFS2k. However, its memory requirement and running time rapidly increased when abstracting larger log datasets (HDFS30m and BGL30m), and it even failed to complete on HDFS(100m) [13]. Consequently, He *et al.* [7, 14] concluded that offline ALATs must implement a parallelization mechanism not to be limited by the computing power and memory of a single computer.

Classification Criteria: We study the algorithm of each ALAT, and we classify an ALAT as "*scalable*" to production log dataset if it accelerates the discovery process by implementing a parallelization mechanism, or if it is an online ALAT (process log messages one after another). We classify an ALAT as *potentially scalable* if the authors explicitly specify that their algorithm can be easily implemented in parallel mode.

Domain: Scalable, potentially scalable, not scalable.

6.5. System Knowledge Independence

Typical Question: *Does the ALAT require any prior manual hard-coded rules, regexps, heuristics based on experts' domain knowledge?*

Definition: Jiang *et al.* specifies that this QA pertains to the amount of knowledge needed about a system for an ALAT to work [29]. Some ALATs require a domain expert to encode rules/regexps based on their experiences. For example, the first step of AEL extracts the dynamic fields in raw log messages based on rules hard-coded by a domain expert (*e.g.,* textttword=value and `is-are-was-were value`). Similarly, the first step of LKE prunes the obvious dynamic fields in raw log messages (*e.g.,* numbers, URIs, IP addresses) based on regexps hard-coded by a domain expert to describe those parameter values. Even though these rules/regexps are simple, they make ALATs infrastructure-dependent, require further manual configuration, and could be difficult to maintain and evolve [28, 48]. In the literature, this QA is also referred to as *no-supervision* [39], domain knowledge [36, 28, 14], needed system knowledge [29], or infrastructure-dependency [48].

Classification Criteria: We study the algorithm of each ALAT, and we classify it as *dependent of system knowledge* if it explicitly specifies that (1) it requires users to define hard-coded rules and/or (2) it requires users to set empirical regular expressions manually.

Domain: Independent of system knowledge, dependant of system knowledge.

6.6. Delimiter Independence

Typical Question: *Can the ALAT abstract various formats of raw log datasets with different delimiters?*

Definition: Heterogeneous log datasets are generated from different systems and components and have a different formats and, thus, different delimiters (*e.g.,* ":", "|", "=", ";", etc.). Ning *et al.* [28] reported that using pre-defined, popular delimiters on heterogeneous log datasets might lead to incorrect abstractions.

For example, IPLOM [35] assumes that spaces delimits words. It abstracts *"creating password:* ***user = (John)****", "creating password:* ***user =(John)****", "creating password:* ***user= (John)****", "creating password:* ***user = (John)****",

etc. into *"creating password: *"* while the correct abstraction is *"creating password: user *"*. Conversely, HLAer algorithm does not assume any predefined delimiters and separates all special symbols and words in a log message. Consequently, all the examples above have the same tokenization: *"creating password : user = (John)"*.
Classification Criteria: We study the algorithm of each ALAT and report an ALAT as *pre-defined* if its algorithm explicitly specifies that it uses a predefined/popular set of delimiters. We classify an ALAT as *user-defined* if it explicitly allows users to modify its set of delimiters. We classify an ALAT as *independent* if its algorithm does not rely on any predefined/popular delimiters.
Domain: Pre-defined, user-defined, idndependent.

6.7. Parameters Tuning Effort

Typical Question: *Does the ALAT require user-defined parameters?*
Definition: Most ALATs require users to fine-tune their parameters, usually following a trial-and-error process [3]. If the parameters are not tuned correctly, then the performance of the ALATs are sub-optimal [3]. Parameter-tuning effort reflects the knowledge required to set and update the parameters of an ALAT [47].
Classification Criteria: We study the algorithm of each ALAT and present the parameter(s).
Domain: The set of parameters.

7. ALATs Comparison

We revisit the ALATs presented in Section 5, according to the aspects of our quality model described in Section 6.

Our results are based on a thorough review of ALAT development contexts and algorithmic characteristics, detailed in Section 5 and Table 1, and on the results of empirical experiments and experiences shared in the literature (see Section 4.4).

Table 2 resume the results and present references to the papers that evaluated the ALATs in terms of the QAs. We provide practitioners with direct references and summarize/group the findings so practitioners benefit from the researchers' experience with ALATs.

In the following, we discuss our findings and provide brief conclusions and promising research directions. Detailed summary of each ALATs algorithm and characteristics is presented in section 5 and Table 1

7.1. Mode

Table 2 shows that Drain, SHISO, and Spell are online ALATs. Online ALATs adjust their event types gradually, they do not need access to all the log data, and they do not require an offline discovery phase. Empirical experiments conducted by Du *et al.* [9] and He *et al.* [13], which compared the accuracy and efficiency of online ALATs to offline ALATs (*e.g.,* LKE, LogSig, IPLOM) on several log files showed that online ALATs are competitive with offline techniques in terms of accuracy and efficiency.

Observation. ALATs can be online without compromising accuracy and efficiency.

Promising Direction. There are few instances of online ALATs in the literature and researchers could investigate further online ALATs. While offline ALATs are desirable to work with a previously-known set of abstracted event types, online ALATs are useful for Web services and systems online monitoring and processing, in which the volumes of logs increase and evolve rapidly, making batch event types discovery time-consuming [9, 13]. Online ALATs could be a valuable addition to modern log-management systems (*e.g.,* ElasticSearch or Splunk), which collect logs in streams.

7.2. Coverage

Jiang *et al.* [29] reported that ALATs based on frequent itemsets accurately extract frequently occurring events but might fail to identify rare events. Further, Zhu *et al.* [8] found that SLCT and LogCluster cannot recognise low repetition events while ALATs based on iterative partitioning, clone detection, or clustering techniques enjoyed high coverage. Nagappan *et al.* [34] indicated that LogHound, LogCluster, and SLCT are designed to abstract frequent log messages and may not abstract log messages that occur less than a user-given threshold. For example, Nagappan *et al.* [34] showed that with thresholds of 50%, 25%, 10%, 5% and even 1%, SLCT could not abstract all log-messages. Zhu *et al.* [8] found that LFA is based on frequent-pattern clustering and has a high coverage because it identifies event types by generating clusters within each log message, unlike LogCluster, LogHond, and SLCT that find clusters within the whole log dataset [34].

Observation. ALATs based on frequent-itemset mining (*e.g.,* LogCluster) tend to have low coverage. ALATs with low coverage might be inadequate for troubleshooting or anomaly detection because, during the analysis phase, they would produce false positives [29].

Promising Direction. ALATs based on frequent mining (*i.e.,* LogCluster, LogHound, and SLCT) are simple, command-based tools that can be easily integrated with other systems and pipelines. They help users to build a model of their log files. A promising direction would be to combine these ALATs with anomaly detection techniques to provide full-scale log-analysis capability.

7.3. Efficiency

Researchers conducted case studies and empirical experiments to evaluate and compare their ALATs with others. We group and report their findings and shared experiences on the efficiency of each ALAT. In the following, we only report on ALAT's efficiency as reported in the reviewed papers and do not measure the ALAT's efficiency ourselves. Also, we do not study the worst-case scenario of each ALAT, which we leave for future work.

Table 2: Automated Log Abstraction Techniques Aspects ("System Independence" and "Tuning" correspond to "System knowledge independence" and "Parameter tuning effort". "Off." and "On." means "Offline" and "Online". "PR" means Proprietary, "PX" means Proxifier, and "Zk" means Zookeeper. "MI" means missing information " Training" means that it depends on the training data. * The effort is for labeling the data.)

Tool	Technique	Mode	Coverage	Efficiency		Scalability	System independence	Delimiter independence	Tuning
				Class	Datasets				
MolFi	Evolutionary Search	Off.	All	Low	Android, BGL, HDFS, HPC, Zk, PX	✗[3]	✓[3]	✗ pre-defined [3]	NA [3]
AEL	Clone detection	Off.	All	High	Android, HDFS	✗[29]	✗ [48, 29]	NA (rules) [48, 34]	merge threshold [29]
				Low	BGL(1GB)				
IPLOM	Iterative Partition	Off.	All	High:	HPC, Thunderbird, Zk, Hadoop, Spark, Windows, Linux, Apache, PX, SysLog, Access, Error, System, Rewrite	potentially [28, 4, 13]	✓ [35, 31]	✗ pre-defined [29, 35]	*FS, PS, UB, LB, CG* [31]
POP	Iterative Partition	Off.	All	High	BGL, HDFS, HPC, PX, Zk	✓[14]	✗[14]	✗ pre-defined [14]	*GS, splitAbs, splitRel, maxDist* [14]
HLAer	Hierarchical Clustering + PR	Off.	All	Low	PR(10m)	potentially [39]	✓[28]	✓[39]	*MinPts*, ϵ [28]
LogMine	Hierarchical Clustering + PR	Off.	All	High	PR(10m)	✓[39]	✓optional [39]	✓[39]	*maxPatternLimit* (optional) [39]
LKE	Hierarchical Clustering	Off.	All	Low	BGL, HDFS, HPC, Zk, PX	✗[14, 49, 3]	✗ [48, 32]	✗ pre-defined [48, 32]	ν, ϱ, ζ (optional) [32]
LogSig	Clustering	Off.	All	High	Thunderbird, Zk, Hadoop, Spark, Windows, Linux, Apache, PX	✗[14, 49, 13]	✓optional [36]	MI	k [36, 14]
				Low	HDFS, BGL, HPC				
LogHound	Frequent itemsets mining	Off.	Freq.	High	SysLog, Windows, Access, Error, System, Rewrite	✗[31, 35]	✓[30]	✗ user-defined [30]	s [31]
				Low	HPC(11.4MB)				
LogCluster	Frequent pattern clustering	Off.	Freq.	High	Nagios, Unix deamon, Mail server	✗[15]	✓[15]	✗ user-defined [15]	s [30]
				Low	Authorization, Network device, application, Web Porxy(16GB)				
LFA	Frequent pattern clustering	Off.	All	MI		✗[34]	✓[34]	MI	NA [34]
SLCT	Frequent pattern clustering	Off.	Freq.	High	BGL, HDFS, HPC, Zk, PX, SysLog, Windows, Access, Error, System, Rewrite	✓[14, 29]	✓[30]	✗ user-defined [30]	s [31, 14]
Drain	DAG	On.	All	High	BGL, HPC, Thunderbird, HDFS, Zk, Hadoop, Spark, Windows, Linux, Apache, PX	✓[13, 4]	✗[13, 4]	✗ pre-defined [13]	NA [13, 4]
Spell	LCS	On.	All	High	BGL, HPC, Thunderbird, HDFS, Zk, Hadoop, Spark, Windows, Linux, Apache, PX	✓[13]	✓[9]	✗ user-defined [9]	τ (optional) [9]
SHISO	Parse tree	On.	All	High	HPC, Thunderbird, Zk, Hadoop, Spark, Windows, Linux, Apache, PX	✗[13]	✓[12]	✓[12]	*c, ts, tm, tr* [12, 9]
				Low	BGL, HDFS				
NLP-LTG	Conditional Random Fields	Off.	All	Training	✓	✗	NA (manually labeled)	High*	
NLM-FSE	bi-LSTM (character-based)	Off.	All	Training	✓	✗	NA (Character based)	High*	

MolFi: Messaoudi *et al.* [3] compared the running time of MolFi to Drain and IPLOM on a benchmark composed of BGL (2K le [5]), BGL (100K le), HDFS (2K le), HDFS (60K le), HPC (2K le), Proxifier (2K le), and Zookeeper (2K le). They reported that (1) MolFi surpassed Drain on BGL (100K le) and (2) it was the slowest on all the other log datasets. Zhu *et al.* [8] evaluated the running time of MolFi on Android, BGL, and HDFS. They showed that (1) MolFi failed to abstract BGL (500MB) and Android (100MB) in a reasonable time (*i.e.,* 6+ hours); (2) it could abstract HDFS (1GB) because HDFS log dataset has a low number of events. They attributed the low efficiency of MolFi to its iterative algorithm NSGA-II, in $O(n^2)$, because a larger population size requires more fitness computations [3] and execution time increases rapidly as size grows.

LKE: Du *et al.* [9] found that LKE is significantly slower than IPLoM and Spell on BGL and HPC and that it failed to complete on BGL (5m le) and HPC (1.5K le). He *et al.* [14, 7, 4] found that LKE failed to complete on several log datasets, such as BGL (40K le), HDFS (100K le), HPC (75K le), Proxifier (9600 le), and Zookeeper (32K le). They attributed the low efficiency of LKE to its quadratic time complexity.

HLAer and **LogMine**:Hamooni *et al.* [39] compared HLAer and LogMine on three proprietary and three public log datasets. They showed that (1) HLAer memory consumption is approximately five times higher than LogMine due to its OPTICS algorithm quadratic memory consumption (2) LogMine running time is 500 times faster than HLAer, (3) HLAer failed to complete on an industrial-proprietary log dataset (10GB), and (4) LogMine pattern-recognition running time stays constant for different log datasets of the same size because its algorithm scans the data once, irrespective of how many event types exist in the dataset. In contrast, HLAer pattern-recognition algorithm UPGMA has a quadratic time complexity.

AEL: Zhu *et al.* [8] measured the running time of AEL on Android (1GB), BGL (1GB), and HDFS (1GB). The results show that AEL is very efficient (*i.e.,* approximately tens of minutes) on Android and HDFS log datasets. Yet, its running time increased rapidly on BGL. They attributed the low efficiency of AEL on BGL to the fact that AEL compares each log-message in a cluster with all the event types in the same cluster. Therefore, AEL running time increases on log datasets that yield large clusters, *e.g.,* BGL.

LogHound: LogHound entails high computational cost during candidate generation on datasets containing log-messages of high length (*e.g.,* HPC, BGL). For instance, Makanju *et al.* [35] evaluated the running time and the memory consumption of LogHound on seven log datasets Access, Error, HPC, Rewrite, SysLog, System, and Windows. Although LogHound performed well on six datasets, it crashed and could not complete on HPC (11.4MB). They attributed the low efficiency of LogHound on HPC to the large number of itemsets generated for each log message.

LogSig: He *et al.* [14, 7, 4, 13] found that LogSig completed the discovery phase on Proxifier (9,600 le), Windows (2K le), Zookeeper (64K le), Thunderbird (2K le), Hadoop (2K le), Spark (2K le), Linux (2K le), and Apache (2K le) but was slower than SLCT and/or IPLOM. LogSig failed to complete in a reasonable time on HDFS (10m le) (*i.e.,* days), HPC (375K le) (*i.e.,* 16+ hours), and BGL(4m le) (*i.e.,* days). They attributed the low efficiency of LogSig, particularly on dataset containing lengthy log messages, to its slow clustering iterations in which LogSig converts each log-message to a set of term pairs.

IPLOM: Makanju *et al.* [35] compared the efficiency of IPLOM to SLCT and LogHound on seven log datasets HPC, SysLog, Windows, Access, Error, System, and Rewrite. The results show that IPLOM is efficient in terms of running time and memory utilization on all datasets. He *et al.* [14, 4] evaluated the running time of IPLOM on HPC (375K le), BGL (4m le), HDFS (10m le), Zookeeper (64K le), Proxifier (9600 le), Thunderbird (2K le), Hadoop (2K le), Spark (2K le), Windows (2k le), Linux (2K le), and Apache (2K le) and they reported that IPLOM is efficient on all datasets (*e.g.,* HDFS (10m le) in 5 min). Du *et al.* [9] compared IPLOM to Spell on HPC and BGL log datasets and reported that IPLOM was the fastest on HPC and slightly slower than Spell on BGL. They attributed the efficiency of IPLOM to the fact that it is not affected by long patterns and low support threshold because of its partitioning algorithm.

SLCT: Makanju *et al.* [35] evaluated the running time and the memory consumption of SLCT on seven datasets HPC, SysLog, Windows, Access, Error, System, and Rewrite. Although LogHound crashed on HPC, SLCT was not affected by the long log-message lengths in HPC because it only generates 1-itemsets. He *et al.* [14, 7] evaluated the running time of SLCT on HPC, BGL, HDFS, Zookeeper, and Proxifier and showed that SLCT is faster than POP and IPLOM on all datasets and that its running time scales linearly with the number of log messages. Mizutani *et al.* [12] compared SLCT, IPLOM, and SHISO on a public security log dataset (60K le)) and showed that SLCT outperformed IPLOM and was slightly slower than SHISO.

LogCluster: Although LogCluster is an improved implementation of SLCT, Vaarandi *et al.* [15] found that SLCT completed faster than LogCluster on seven different log datasets: UNIX daemon (740MB)), Web proxy (16GB), Authorization messages (3GB), Nagios (391MB), Mail server (246MB), Network device (4GB), and application messages (9GB) due to its simpler candidate genera-

[5] "le" stands for log entries in opposition to sizes in bytes.

tion procedure. LogCluster failed to complete in a reasonable time (*i.e.,* 2+ hours) on Web proxy messages of 16GB.

POP: He *et al.* [14] evaluated the running time of POP on HPC, BGL, HDFS, Zookeeper, and Proxifier: (1) POP is efficient on all datasets; (2) POP is slower than SLCT and IPLOM because Pop parallelization mechanism entails more running time for setting up the nodes and for communications among nodes; (3) POP enjoys an almost constant running time as the log size grows; and, (4) POP becomes even faster than IPLoM on HDFS (10m le).

SHISO: Mizutani [12] evaluated SHISO on Public Security Log (670K le) and found that SHISO was faster than SLCT and IPLOM. Yet, He *et al.* [4, 13] found that SHISO was the slowest between IPLOM, Spell, and Drain on HPC (375K le), Zookeeper (64K le), Proxifier (9600 le), Hadoop (2K le), Windows (2K le), Spark (2K le), Linux (2K le), and Apache (2K le) and failed to complete in reasonable time on BGL (4m le) (*i.e.,* days) and HDFS (10m le) (*i.e.,* 18+ hours). They attributed the low efficiency of SHISO to its tree construction algorithm that may create an unbalanced tree.

Spell: Du *et al.* [9] compared Spell and IPLOM on BGL5m and HPC400K. They found that Spell completed in 9 seconds on HPC, slightly slower than IPLOM, and outperformed IPLOM on BGL. He *et al.* [4] evaluated Spell on BGL (4m le), HPC (375K le), HDFS (10m le), Zookeeper (64K le) and Proxifier (9600 le). The results showed that Spell is faster than SHISO on all datasets and tends to become as fast as Drain and IPLOM on large log datasets (*i.e.,* BGL and HDFS). The authors attributed the increase of Spell efficiency as the log dataset size grows to the fact that most log messages directly find an event type match in the prefix tree. The computation cost of calculating the LCS between two log-messages is considerably reduced.

Drain: He *et al.* [13] compared its proposed ALAT, Drain, to IPLOM, SHISO, and Spell on 11 log datasets HPC(375K le), BGL (4m le), HDFS (10m le), Zookeeper (64 le), Proxifier (9600 le), Thunderbird (2K le), Hadoop (2K le), Spark (2K le), Windows (2K le), Linux (2K le), and Apache (2K le). The results showed that Drain required the least running time on all datasets (*e.g.,* BGL (4m le) in 2 min and HDFS (11m le) in 7 min) and had similar running time compared with the offline ALAT IPLoM. Messaoudi *et al.* [3] reported that Drain is efficient on BGL, HDFS, HPC, and Zookeeper, but Drain crashed on a proprietary dataset (300K le).

Observation. We observe that: (1) some ALATs are highly efficient on certain types of log datasets, but fail to complete on others; (2) the ALATs efficiency depends on the characteristics of the log datasets [47] (*e.g.,* numbers of event types, length of log-messages, length of patterns, number of log-messages, etc.); and, (3) few articles measured/reported the memory consumption of the ALATs during their empirical experiments.

Promising Direction. (1) Researchers could investigate novel ways to enhance efficiency, such as the use of distributed architecture, *e.g.,* installing ALATs on several nodes rather than on a unique computer [13]; (2) Researchers could establish a "common" log dataset on which all ALATs should be evaluated, which would ease their comparison; (3) resource utilization is an important aspect of evaluating the efficiency of a system and studies should evaluate and report the memory consumption of their proposed ALATs; and (4) practitioners can obtain insights from the reported efficiency of each ALAT but should evaluate the ALATs efficiency on their own log datasets. With respect to (2), we recommend researchers to test their ALATs on datasets with many event types and long log message lengths, BGL and HPC, to validate their efficiency.

7.4. Scalability

The running time of sequential, offline ALATs with time complexity $O(n)$ scales linearly with log size. However, He *et al.* [14] showed that such ALATs can have a steep gradient and, thus, an long running time on production logs. They evaluated three such ALATs, LogSig, SLCT, and IPLOM, on two large, synthetic log datasets: BGL (200m le) and HDFS (200m le). They showed that (1) LogSig could not complete HDFS10m and BGL10m in a reasonable time; (2) SLCT completed HDFS200m in about 30 min and BGL200m in about 18 min; and, (3) IPLOM failed to complete on HDFS150m and incurred 16+GB memory consumption for BGL30m and HDFS30m. However, SLCT running time increased rapidly due to a single thread of control and IPLOM memory requirement and running time increases because IPLOM is limited by the memory of a single computer and loads all log-messages into memory. Makanju *et al.* [35] reported that LogHound crashed on HPC (11.4MB) as the virtual memory had raised to 4GB and the resident memory consumption became 1.6GB. Also, Zhu *et al.* [8] reported that AEL running time increased rapidly on a large BGL dataset.

LogMine and POP implement parallel computing architectures and stay efficient when handling large-scale data. In particular, POP is built on top of Apache Spark, and LogMine scans log-messages only once using a MapReduce-based mechanism. He *et al.* [14] found that; (1) POP abstracts large-scale log dataset HDFS200m in about 7 min and BGL200m in about 20 min; (2) POP is faster than SLCT on HDFS (200m le); (3) POP is slower than SLCT on BGL but POP enjoys the slowest increase in its running time which becomes similar to SLCT as log size increases to 200m log entries. Hamooni *et al.* [39] compared the running times of sequential and MapReduce

implementations of LogMine and found that the MapReduce implementation runs up to five times faster than the sequential implementation and can abstract millions of log messages in a few minutes.

As detailed in Section 5, authors [35, 28] specified that IPLOM and HLAer algorithms are designed to be easily parallelized to handle large-scale data which make them potentially scalable.

Online ALATs, Drain, SHISO, and Spell are not limited by the memory of a single computer and abstract log-messages one at a time. He *et al.* [13] found that Drain and Spell are suitable for large-scale log datasets (*e.g.,* Spell completed HDFS (10m le) in approximately 11 min). However, SHISO was not scalable and took 3 hours to complete BGL (4m le) and approximately 2 hours to complete on HDFS (10m le). This mainly because SHISO uses deep parse trees while Drain uses a fixed DAG with a cache mechanism and Spell uses prefix trees.

Observation. (1) Most of the sequential, offline ALATs are efficient on sample log datasets and their running times increase linearly wrt. the number of log messages [12, 7, 9]. (2) Efficient sequential, offline ALATs may not handle large log datasets because they are limited by the memory and the computing power of a single computer. (3) Offline ALATs that implement parallel computing architectures (*i.e.,* LogMine and POP) stay efficient when handling large log datasets. (4) Online ALATs might not complete in a reasonable time if their data structure construction algorithm builds a deep and unbalanced data structure of log messages (*e.g.,* SHISO).

Promising Direction. In production, ALATs must scale to abstract large log files in reasonable times. Researchers could propose algorithms that implement parallelization algorithms and–or that decouple individual tasks. Another direction is to add data/task parallelization algorithms to existing ALATs, such as IPLOM and HLAer.

7.5. Independence of System Knowledge

AEL algorithm relies on domain experts hard-coded rules (*e.g., "is-are-was value"*) to identify dynamic fields in raw log messages (*e.g.,* IP addresses, numbers, memory) and replaces them with a generic token ($V) then divides log messages into different bins according to their numbers of words and generic token ($V) [29].

LKE reduces the influence of the dynamic fields in raw log messages on its clustering algorithm by pruning typical dynamic fields according to empirical regexps provided by a domain expert [32].

MolFi enhances the accuracy of its algorithm by replacing typical dynamic fields with a special token `#spec#` using regexps hard-coded by a domain expert. In later stages of the search, MolFi ignores the special tokens `#spec#` [3].

POP and **Drain** algorithms implement a "pre-processing by domain knowledge" step to enhance their accuracy. Thus, they require experts to write regexps based on their domain knowledge. The algorithm uses these regexps to prune the dynamic fields in raw log messages. POP also allows users to optionally provide regexps to specify the characteristics of relevant log events [14].

LogSig [36] algorithm does not require domain experts to hard-code rules or regexps manually. However, Tang *et al.* [36] presented two optional approaches to improve the accuracy of LogSig algorithm by relying on domain experts' regexps or constraints.

LogMine [39] implements a *type detection step* to enhance the performance of LogMine, which requires experts to write a set of regexps to pre-define types such as date, time, IP, and number. Then LogMine replaces the real value of each field with the name of the field. For example, LogMine replaces 2019/10/25 with `date` and 192.168.10.15 with `IP`. This step is not mandatory.

HLAer, IPLOM, LFA, LogCluster, LogHound, SLCT, SHISO, and Spell do not rely on rules or regexps for their algorithm to work.

Observation. (1) There is a trade-off between independence of system knowledge and performance. (2) Modern systems are more and more complex, which makes adequate and up-to-date detailed domain knowledge difficult to obtain [28]. (3) ALATs that rely on domain experts' manual effort may not be flexible, heterogeneous, and evolutive.

Promising Direction. Researchers should investigate ALATs that do not rely on domain knowledge, such as ALATs that dynamically updates their rules. If not possible, they should propose ALATs that allow experts to use their domain knowledge but without requiring it.

7.6. Delimiters Independence

To determine the log-messages lengths (*i.e.,* number of words) during the first partitioning of log messages, Drain, IPLOM, and POP assume that tokens/words are delimited by *white spaces*. However, as detailed in Section 6.6, this assumption is incorrect and may produce inaccurate partitions.

To tokenize log messages, LKE uses the pre-defined delimiter *white space* and MolFi *white space*, *parentheses*, and *punctuation characters*. Ning *et al.* [28] indicated that pre-defined delimiters limit the analysis of log datsets with different delimiters.

To identify frequent words in the log dataset, LogCluster, LogHound, and SLCT use customizable delimiters using the *–separator* option for LogCluster and *-d* option for LogHound and SLCT (*white space* by default). Spell uses a customizable set of delimiters but Mi *et al.* [47] found that user-defined delimiters may not be applicable across systems and users should redefine them for each log dataset format.

HLAer, LogMine, SHISO do not rely on pre-defined or user-defined delimiters to identify tokens/words in the log entry. They consider each word, number, and symbol in the log entry as a token and separate them by a white space. For example `017-09-26 12:40:15, INFO impl.FsDatasetImpl - Time taken to scan block pool BP-805143380 on /home/data3/current 30ms` is tokenized into `017 - 09 - 26 12 : 40 : 15 , INFO impl . FsDatasetImpl - Time taken to scan block pool BP - 805143380 on / home / data3 / current 30 ms`. Therefore, these ALATs can tokenize heterogeneous log datasets without any delimiter definition from the user.

Observation. Supporting multiple log formats is challenging but required for the industrial adoption of ALATs. In the absence of a standard and common log format, ALATs must support new formats and thus be independent of delimiters or allow user-defined delimiters.

Promising Direction. Most ALATs depend on token delimiters and predefined rules. A standard and common log format would promote sharing of data and synergy among ALATs. Such log format could draw inspiration from the attempts to develop a standard language for execution traces (a structured form of logs) [50, 51], in which meta-modeling techniques allow scalable and expressive trace formats.

7.7. Parameters Tuning Effort

We now present the user-given parameters required by each ALAT and the experience shared by researchers when tuning them.

Drain initializes automatically and updates dynamically its parameters for each log file. [13].

LFA and **MolFi** do not require the user to set any parameter and use internal heuristics to set the threshold(s) of their algorithms automatically. For example, LFA sets its word-frequency threshold to the lowest word frequency in the cluster with the most number of words.

Spell has one parameter *message type threshold* τ to compare a new log message sequence with existing LCSseq in LCSMap (as detailed in Section 2). Users can optionally set this parameter; otherwise, Spell sets τ to a default value of half the length of the new sequence [9].

LogMine has one parameter *Max Pattern Limit* to determine the desired level in the hierarchy of patterns. Users can set this parameter, else LogMine iterates until it reaches the most generic pattern (*i.e.,* the event type containing only wildcards [39].

AEL does not explicitly specify that it requires users to set a parameter. However, Jiang *et al.* [29] used a threshold of 5 to prevent merging of similar yet different event types. This value was adequate for their case study but users must adjust this threshold based on the content of their log datasets.

LogCuster, **LogHound**, and **SLCT** require users to set a *support threshold* s. They identify itemsets/words that occur more frequently than s in the log datasets and generate cluster candidates from these frequent itemsets. They only output clusters with a support value equal to or greater than$s$. Vaarandi *et al.* [30] found that setting s to an appropriate value is a challenging task because the identification of runtime parameters depends on it. Empirical experiments [15, 33, 48, 35] showed that a high support threshold value generates generic event types and anomalous messages could go undetected while a low threshold yields too specific event types.

IPLOM requires users the set five parameters. (1) *File Support threshold (FS)* controls the number of clusters produced. Clusters that have a lower support value than this threshold are discarded and increasing this value decreases the number of found event types. (2) *Partition Support threshold (PS)* controls backtracking during the partitioning step. Setting this threshold to 0 means that no backtracking will be done. (3) *Upper Bound (UB)* and *Lower Bound (LB)* controls the 1-to-M and M-to-1 relationships during the partitioning by bijection to decide if the M side represents constants or dynamic fields. (4) *Cluster Goodness threshold (CG)* controls the partitioning level computed for each new cluster. IPLOM will not partition a cluster if its cluster-goodness is higher than *CG* [31]. Makanju *et al.* [35] found that IPLoM remains stable when changing its parameters values and is mainly sensitive to *FS*.

IPLOM *FS* is equivalent to LogHound and SLCT *support threshold*. Unlike SLCT and LogHound, IPLOM does not use its *support threshold* to identify frequent itemsets and to generate clusters/partitions but removes the partitions that fall below *FS* at the end of each partitioning. This step is optional and setting *FS* to 0 indicates that no partition pruning is done.

POP requires four parameters *Group Support (GS)*, *splitAbs*, *splitRel*, and *maxDistance*. (1) *GS* is equivalent to IPLOM *cluster goodness* and controls the partitioning level. Complete clusters/partitions that have a cluster-goodness value higher than *GS* avoid further partitioning. (2) *splitAbs*

and *splitRel* are used to identify constant and dynamic fields in log messages. A split token position with an absolute threshold and relative threshold higher than *splitAbs* and *splitRel* is considered to be a dynamic field. (3) *maxDistance* is used to merge two clusters if their Manhattan distance is smaller than its value.

LogSig requires users to set one parameter, the number of clusters k (*i.e.,* number of event types to be generated). The choice of an appropriate value for k depends on the user's domain knowledge to the log dataset [15].

LKE requires three parameters. (1) *Edit distance weight* ν. LKE measures the similarity between log messages by the *weighted edit distance* and ν controls the weight function that computes words weights at different positions in a log message. (2) *Private content threshold* ϱ. LKE counts the number of different values at each token position within the same cluster of log messages. If the number is less than ϱ, then LKE considers this token position a constant field and further split the cluster at this token position. (3) *Cluster threshold* ζ. LKE automatically sets this threshold via k-means clustering. Du *et al.* [9] found that setting this threshold manually to a value calculated for a smaller log dataset significantly improved the runtime of LKE.

SHISO requires four user-given parameters: (1) the maximum number of children per node c; (2) the similarity threshold *ts* to find the most suitable log group for each new log message; (3) the format merge threshold *tm*; and, (4) the format lookup threshold *tr* both used during the adjustment phase. He *et al.* [13] indicated that tuning SHISO parameters require a lot of effort because they must be tuned for each log file.

HLAer requires two parameters: (1) the minimum number of event types in the final clusters, *MinPts*, and (2) the maximum distance between any two event types in a cluster, ϵ. Authors in [39, 28] found that HLAer parameters must be set for each log dataset by an expert via empirical experiments.

Observation. Parameters tuning is challenging and requires effort, especially when the parameters are not intuitive [3]. Pre-defined parameters might become ineffective and limit the robustness of ALATs against new logging statements [13]. Using the same parameters for different log datasets might lead to inaccuracy [13]. Drain mitigates these challenges by setting automatically and dynamically its parameters.

Tuning parameters for large log datasets is even more challenging because a trial-and-error approach is impossible. Practitioners may then tune parameters on small log datasets and apply them to large log datasets. He *et al.* [14] applied five offline ALATs (IPLOM, LKE, LogSig, POP, and SLCT) on several log datasets and found that only IPLOM and POP performed consistently on large log datasets after parameter tuning on small log datasets.

Promising Directions. Researchers should invest into techniques to tune dynamically the algorithms parameters.

7.8. Quality Aspects of Supervised ALATs

ALATs discussed in this section were all unsupervised. We identified two supervised ALATs: NLP-LTG and NLM-FSE. These ALATs face additional challenges that we briefly discuss now. Supervised ALATs based on NLP have shown to be accurate but have limitations, such as their ability to generalize, the need for expert knowledge to label data, and their dependence on training data. Due to the confidential nature of industry log files, research suffers from a lack of access to large, industrial log files to train and evaluate their ALATs [42, 40].

Observation. The main limitation of these ALATs is the need for large, industrial log files to train and test models.

Promising Direction. Researchers should build and share log files to the benefit of the research community.

8. Threats To Validity

We now discuss the threats to the validity of our results and recommendations.

Construct Validity. Construct validity threats concern the accuracy of the observations with respect to the theory. We considered a quality model that covers key aspects of log abstraction tools. We extracted these aspects from the literature through a systematic review of 89 papers. Thus, we argue that there is no threat to the construct validity of our results and recommendations besides the threat to any systematic-literature review: we may have missed a few relevant papers. We accepted this threat and followed best practices to perform an SLR, *e.g.,* [5], as well as our experience with previous SLRs, *e.g.,* [52].

Internal Validity. Internal validity threats concern the factors that might influence our results. The selection of the research papers is one possible threat. We may have missed relevant papers. We mitigated this threat by using the Engineering Village, which is one of the main sources of research papers in this field of study. In addition, we performed a snowballing process to reduce the risk of missed papers. Two of the authors also reviewed the selected papers thoroughly to ensure that they fit this study. Another threat concerns the tools that we selected. We may have missed some tools or misunderstood some of their aspects described in the corresponding papers. We mitigated this threat by following a systematic survey process. We also made sure that at least two authors reviewed the tools and their features. Another threat to internal validity is that

our analysis of the characteristics of the ALATs is based on the information provided in the selected research papers. We did not check the source code of the tools to assess the correctness of their implementation which would require access to all of their source code and extensive resources out of the scope of this paper.

Conclusion Validity. Conclusion validity threats correspond to the correctness of the obtained results. We classified the selected tools based on a detailed review of the literature that describes them and our own experience using them. When a tool used an algorithm that was published in other studies, we made every effort to review these studies to ensure that we properly interpret the tool aspects. Table 2 shows the papers that describe how a given quality aspects is implemented in a tool. We strive to provide as many details as possible to allow the assessment and reproducibility of our results.

Reliability Validity. Reliability validity concerns possibility of replicating this study. We studied 17 ALATs but we cannot claim that these are representative of all tools. Based on the literature review that we conducted, we argue that these tools are representative of existing log abstraction techniques. Moreover, we put on-line material to ease assessing and reproducing our study.

External Validity. External validity is related to the generalizability of the results. We performed our study on 17 ALATs that cover a wide range of log abstraction techniques. We do not claim that our results can be generalized to all ALATs, in particular industrial, proprietary ALATs to which we did not have access. They are, however, representative of ALATs in the scientific literature. Besides, the seven quality aspects of our quality model can be used to classify any ALAT.

9. Conclusions and Future Work

Logs contain a wealth of data that can help software engineers to understand a system's run-time properties. However, modern systems have become so large and complex that they produce too huge amounts of log data to analyze. Also, logs often come in different formats, hindering the analyses of their content and making their uses even more complex [4, 3]. To tackle these problems, software engineers have at their disposable Automated Log Abstraction Techniques (ALATs) that they can use to reduce the amount of data to process trough their log-abstraction algorithms.

However, there is a gap between the industry and academia. First, software engineers are not aware of all existing ALATs developed in academia and the characteristics of their algorithms. Second, software engineers do not have the time and resources to study and understand the characteristics of each ALAT algorithm. To reduce this gap, we conducted a thorough study in which we grouped, summarized, and compared 17 ALATs based on seven quality aspects identified from the literature: mode, coverage, efficiency, scalability, independence of system knowledge, heterogeneity, and parameter tuning effort required.

In this paper, we reported on our systematic review of the literature on ALATs. From 2,864 papers, we thoroughly reviewed 89 papers to identify unique ALATs and quality aspects relevant to software engineers. Then, we proposed a quality model with seven industry relevant quality aspects for evaluating ALATs. We also identified, compared, and evaluated 17 ALATs using our quality model. We observed that there is not one ALAT that can address all requirements and practitioners must make compromises. The results in Section 7 bridge the gap between industry and academia: practitioners do not have to spend valuable time investigating state-of-the-art ALATs. Instead, they can focus on experimenting with a subset of candidate ALATs and decide which fits best with their particular use cases. To the best of our knowledge, this is the first and only extensive study of ALATs and recommender for ALATs based on a quality model.

Researchers can use our model and recommendations to learn about the state-of-the-art ALATs, understand research gaps, enhance existing ALATs, and–or develop new ones. Software engineers can use our model and recommendations to understand the advantages and limitations of existing ALATs and to identify the ones that best fit their need.

Future Work. During our study of all available ALATs, we made several observations that could be the basis for future works.

Salfner *et al.* [53] proposed a new log format to make log datasets more expressive and comprehensive. They recommended adding Event IDs in the log format to ease automatic log analysis and accurate abstraction. Adding information to logging statements could help in the analysis of complex modern systems. This information should be outputted automatically for consistency. Some logging frameworks (log4j, log4j2, and SL4J/logback) provide (semi-)automatically this information each log entry. However, this feature is not yet available in all systems and, sometimes, may dramatically and unnecessarily increase the size of log datasets.

Moreover, the values of some dynamic variables may be useful for log analyses. For example, they can serve as identifiers for a particular execution, such as *block-id* in an HDFS log and *instance-id* in an OpenStack log [16]. When matching new log entries, ALATs could allow software engineers to keep or ignore dynamic variables.

Researchers evaluate their proposed ALAT on different log datasets, which make them harder to compare. The community should establish a "common" dataset on which all ALATs should be compared. Moreover, most ALATs depend on log datasets availability to train. Future work include collecting and curating such a dataset.

We also plan on measuring the efficiency of each ALATs on various log datasets and reporting the worst-case scenario for each ALAT.

Furthermore, we intend to study how different practitioners use ALATs in practice through surveys and experiments with practitioners from industry. Also, we want to extend this work to catalogue log-analysis techniques, another component of log mining and to investigate the requirements for ALATs by different characteristics of log input datasets and different log-mining goals. We thus want to assist software engineers with log management.

Finally, there is a need to reconcile the areas of logging and tracing. Logs are user-defined, whereas traces usually contain executable code snippets, with function calls, etc. Many techniques for trace abstraction and modeling exist, *e.g.,* [50, 54, 55]. Researchers should study how these techniques could apply to log datasets.

Acknowledgements

This work has been supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] A. Oliner, A. Ganapathi, W. Xu, Advances and challenges in log analysis, Communications of the ACM 55 (2) (2012) 55–61.

[2] A. Miransky, A. Hamou-Lhadj, E. Cialini, A. Larsson, Operational-log analysis for big data systems: Challenges and solutions, IEEE Software 33 (2) (2016) 55–59.

[3] S. Messaoudi, A. Panichella, D. Bianculli, L. Briand, R. Sasnauskas, A search-based approach for accurate identification of log message formats, in: Proceedings of the 26th IEEE/ACM International Conference on Program Comprehension (ICPC'18), ACM, 2018, pp. 167–177.

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

[5] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, S. Linkman, Systematic literature reviews in software engineering–a systematic literature review, Information and software technology 51 (1) (2009) 7–15.

[6] B. Kitchenham, Procedures for performing systematic reviews, Keele, UK, Keele University 33 (2004) (2004) 1–26.

[7] P. He, J. Zhu, S. He, J. Li, M. R. Lyu, An evaluation study on log parsing and its use in log mining, in: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, 2016, pp. 654–661.

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

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

[10] D. Yuan, S. Park, Y. Zhou, Characterizing logging practices in open-source software, in: Proceedings of the 34th International Conference on Software Engineering, IEEE Press, 2012, pp. 102–112.

[11] B. Chen, Z. M. J. Jiang, Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation, Empirical Software Engineering 22 (1) (2017) 330–374.

[12] M. Mizutani, Incremental mining of system log format, in: Services Computing (SCC), 2013 IEEE International Conference on, IEEE, 2013, pp. 595–602.

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

[14] P. He, J. Zhu, S. He, J. Li, M. R. Lyu, Towards automated log parsing for large-scale log data analysis, IEEE Transactions on Dependable and Secure Computing.

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

[16] M. Du, F. Li, G. Zheng, V. Srikumar, Deeplog: Anomaly detection and diagnosis from system logs through deep learning, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2017, pp. 1285–1298.

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

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

[19] M. Islam, K. Wael, A. Hamou-Lhadj, Anomaly detection techniques based on kappa-pruned ensembles, IEEE Transactions on Reliability 67 (1) (2018) 212–229.

[20] W. Khreich, B. Khosravifar, A. Hamou-Lhadj, C. Talhi, An anomaly detection system based on variable n-gram features and one-class svm, Elsevier Journal of Information & Software Technology 91 (2017) 186–197.

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

[22] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, Inferring models of concurrent systems from logs of their behavior with csight, in: Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 468–479.

[23] F. Salfner, M. Malek, Using hidden semi-markov models for effective online failure prediction, in: 2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007), IEEE, 2007, pp. 161–174.

[24] S. Kobayashi, K. Fukuda, H. Esaki, Mining causes of network events in log data with causal inference, in: 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), IEEE, 2017, pp. 45–53.

[25] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, G. Jiang, Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs, ACM SIGPLAN Notices 51 (4) (2016) 489–502.

[26] W. Xu, L. Huang, M. I. Jordan, Experience mining google's production console logs, in: Proceedings of the Workshop on Managing Systems via Log Analysis and Machine Learning Techniques, 2010, p. 5.

[27] X. Xu, L. Zhu, I. Weber, L. Bass, D. Sun, Pod-diagnosis: Error diagnosis of sporadic operations on cloud applications, in: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, IEEE, 2014, pp. 252–263.

[28] X. Ning, G. Jiang, Hlaer: A system for heterogeneous log analysis, in: SDM Workshop on Heterogeneous Learning, 2014, p. 1.

[29] Z. M. Jiang, A. E. Hassan, G. Hamann, P. Flora, An automated approach for abstracting execution logs to execution events, Journal of Software Maintenance and Evolution: Research and Practice 20 (4) (2008) 249–267.

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

[31] A. A. Makanju, A. N. Zincir-Heywood, E. E. Milios, Clustering event logs using iterative partitioning, in: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2009, pp. 1255–1264.

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

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

[34] M. Nagappan, M. A. Vouk, Abstracting log lines to log event types for mining software system logs, in: Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on, IEEE, 2010, pp. 114–117.

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

[36] L. Tang, T. Li, C.-S. Perng, Logsig: Generating system events from raw textual logs, in: Proceedings of the 20th ACM international conference on Information and knowledge management, ACM, 2011, pp. 785–794.

[37] M. Ankerst, M. M. Breunig, H.-P. Kriegel, J. Sander, Optics: ordering points to identify the clustering structure, ACM SIGMOD Record 28 (2) (1999) 49–60.

[38] P. Sneath, R. Sokal, Unweighted pair group method with arithmetic mean, Numerical Taxonomy (1973) 230–234.

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

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

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

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

[43] A. Graves, N. Jaitly, A.-r. Mohamed, Hybrid speech recognition with deep bidirectional lstm, in: Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on, IEEE, 2013, pp. 273–278.

[44] F. Deissenboeck, E. Juergens, K. Lochmann, S. Wagner, Software quality models: Purposes, usage scenarios and requirements, in: Software Quality, 2009. WOSQ'09. ICSE Workshop on, IEEE, 2009, pp. 9–14.

[45] I. Iec25010, systems and software engineering–systems and software quality requirements and evaluation (square)–system and software quality models, International Organization for Standardization 34 (2011) 2910.

[46] Istqb glossary, `https://glossary.istqb.org/en/search/`, (Accessed on 10/21/2019).

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

[48] L. Huang, X. Ke, K. Wong, S. Mankovskii, Symptom-based problem determination using log data abstraction, in: Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research, IBM Corp., 2010, pp. 313–326.

[49] L. Li, Y. Man, M. Chen, A method of large-scale log pattern mining, in: International Conference on Human Centered Computing, Springer, 2017, pp. 76–84.

[50] A. Hamou-Lhadj, T. Lehtbridge, A metamodel for the compact but lossless exchange of execution traces, Springer Journal on Software and Systems Modeling (SoSym) 11 (1) (2019) 77–98.

[51] F. Hojaji, B. Zamani, A. Hamou-Lhadj, T. Mayerhofer, E. Bousse, Lossless compaction of model execution traces, Springer Journal on Software and Systems Modeling (SoSym) (2019) 1–32doi:https://doi.org/10.1007/s10270-019-00737-w.

[52] Z. Sharafi, Z. Soh, Y.-G. Guéhéneuc, A systematic literature review on the usage of eye-tracking in software engineering, Inf. Softw. Technol. 67 (C) (2015) 79–107. doi:10.1016/j.infsof.2015.06.008.
URL `https://doi.org/10.1016/j.infsof.2015.06.008`

[53] F. Salfner, S. Tschirpke, M. Malek, Comprehensive logfiles for autonomic systems, in: 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings., IEEE, 2004, p. 211.

[54] A. Hamou-Lhadj, T. Lethbridge, Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system, in: Proceedings of the 14th International Conference on Program Comprehension, IEEE, 2006, pp. 181–190.

[55] B. Cornelissen, Z. Andy, A. v. Deursen, L. Moonen, R. Koschke, Lossless compaction of model execution traces, IEEE Transactions on Software Engineering 35 (5) (2009) 684–702.