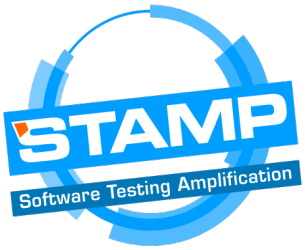

Horizon 2020
Communications Networks, Content & Technology
Cloud & Software
Research & Innovation Action
Grant agreement n° 731529

Title:	WP3 – D3.1 – Survey on logging practices and tools v1
Date:	May 18, 2017
Writer:	Joop Aue (TU Delft) Mauricio Aniche (TU Delft) Arie van Deursen (TU Delft) Andy Zaidman (TU Delft)
Reviewers:	INRIA, TellU

Table Of Content

1. Executive Summary

Using logs to detect and diagnose problems in software systems is no longer a feasible human process. The ever increasing amount of logs produced by present-day systems calls for more advanced techniques to enable log analysis. A great deal of log research has been focused on abstracting over log messages, clever analysis techniques and best practices for logging. An overview of the field, however, has not yet been composed. This makes it difficult for practitioners to identify what is relevant for them, and for researchers to determine interesting angles to explore. To this end we present a literature survey on the field of log analysis. In this survey we outline the different techniques and practices introduced by research in the field. We find that the results of the various works do not lend themselves well to comparison, and suggest future steps of research to overcome this lack of clarity. Furthermore, we suggest areas of improvement in terms of the applicability and feasibility of analysis techniques.

This literature survey started as a tour d'horizon of the scientific literature pertaining to the matter. Subsequently, during the May meeting of the STAMP consortium, the industrial members of STAMP were asked whether the scientific view that we constructed resembled the current state-of-the-art as they experienced it. There was clear agreement that this was the case.

2. Revision History

Date	Version	Author	Comments
18-May-2017	1.00	Joop Aue (TU Delft) Mauricio Aniche (TU Delft) Arie van Deursen (TU Delft) Andy Zaidman (TU Delft)	First version.
31-May-2017	1.10	Joop Aue (TU Delft) Mauricio Aniche (TU Delft) Arie van Deursen (TU Delft) Andy Zaidman (TU Delft)	Fianl draft

3. Objectives

The objective of this deliverable is to provide an overview of logging approaches that are known in scientific literature and to verify that what is present in scientific literature resembles the industrial state-of-the-art, or rather, whether this is state-of-the-practice.

4. Introduction

Logging in the field of computer science is the practice of recording events that provide information about the execution of an application. Log messages are an effective way to infer what has happened during the execution of a production system to diagnose problems effectively [Chen et al., 2004, Yuan et al., 2010]. In a test setting a developer can debug an application to locate the origin of a failure. However, in a live execution environment debugging is no longer feasible due to the ever increasing complexity of systems. In addition, making core dumps to diagnose problems in large applications is not feasible anymore for the same reason. This underlines the importance of logs and their application is almost any system.

As applications and system components are becoming bigger, more complex, and faster, the accompanying increase in log data produced is evident. While in earlier days human operators were able to diagnose problems by hand using the logs, this no longer applies to today's systems. The ever increasing size of systems has resulted in present-day applications that easily generate millions of log messages on a daily basis [Xu et al., 2010]. It has been stated that human interpretation is no longer feasible on this scale and automated approaches have become a necessity [Kc and Gu, 2011, Makanju et al., 2009]. In the literature a wide range of techniques, methods and guidelines have been proposed as today's approaches to facilitate effective log analysis. Among these approaches is enabling diagnosis to deal with millions of log messages. In addition to this, efforts are made to automatically detect failures and anomalies. However, there do not seem to be solutions that are widely used in practice and there is no evidence of a standard and effective way to apply logging and analysis in general. Moreover, there is no overview of what has been investigated in the field of log analysis. Although papers that focus on a specific aspect (e.g. log clustering or log parsing) mention related work that introduce a similar perspective, the bigger picture has not yet been compiled into a complete overview. This makes it difficult for practitioners to identify what is relevant for them. Also the lack of overview makes it harder for researchers to identify interesting and useful angles of research to pursue.

To this end we present an overview of the current state of the art of log analysis in research. With this overview we attempt to give insights into the work concerned with dealing with large numbers of logs. Moreover, the objective is to outline the possibilities to extract the desired information from logs, e.g., failures or anomalies. In addition, we study best practices to improve log statements. For future research we make an effort to identify what is missing in current log research and what should be considered to tackle the shortcomings.

This survey is further organized as follows: Section 5 provides a background on logging in general and logging terminology. Section 6 introduces our literature survey research methodology, which we apply in Section 7. Techniques employed to deal with reducing the number of logs to simplify log analysis are discussed in Section 8. In Section 9 we outline failure detection, anomaly detection, and failure diagnosis techniques. Practices for log enhancements are discussed in Section 10 and in Section 11 we present the discussion of our findings and we summarize.

5. Background

Applications have logged system events since the beginning of the computer era. It is however unclear what the first evidence of logging is. Logs have an application in a wide range of different fields. For this reason, logging is a generic topic even when scoped to application logging. We encountered a variety of terms to denote logs that are either synonyms or overlapping in meaning. To remove some ambiguity and confusion we go through the differences and use this to scope the survey.

In the communication security field *audit logs* are used to record the interactions of individual users and systems to enable the reconstruction of events [Schneier and Kelsey, 1999]. Audit logs, also called audit trails, are also used to record financial transactions, and in online proofing and telecommunication. *Transaction logs* have an application in database systems to recover from crashes and to maintain consistency [TechNet, 2016]. *Query logs*, also referred to as transaction logs, are used to capture user behaviour in search engines [Jansen, 2006]. These logs contain information-searching behaviour of users, and can be used to optimize search engine results and for content collection. *Event logs* refer to records of events taking place in the execution of a system [Cinque et al., 2013]. They are also referred to as *software logs* [Moreta and Telea, 2007], *computer logs* [Takada and Koike, 2002], *application logs* [Sarkar et al., 2014], *system logs* [Mariani and Pastore, 2008], *console logs* [Xu et al., 2009], and *execution logs* [Beschastnikh et al., 2011]. We reason that these terms are used interchangeably and therefore make no distinction between them. This survey focuses on event logs and will not consider audit, transaction and query logs. Other terms we encountered are *error logs* [Salfner and Tschirpke, 2008], which are a subset of event logs that report errors, and *webserver logs*, which characterize logs specific to webservers only [Liu and Kesˇelj, 2007].

In the field of software engineering the following terms are commonly used: *fault*, *error* and *failure*. Within the scope of logging and log analysis this is not different. Because logs are often linked to failures we clarify the related terminology. We define faults, errors and failures according to the IEEE Standard Glossary on Software Engineering

Terminology (IEEE Standard 610.12-1990) [Radatz et al., 1990]:

fault an incorrect step, process or data definition in a program

error the difference between the observed and the expected outcome.

failure the inability of a system or component to perform its required function.

A fault is a root cause, which can produce an error. Errors do not necessarily have side-effects visible to the user and may be handled by the system. Errors that propagate further and result in a crash, incorrect functionality or bad results are referred to as failures. For example, a piece of code that checks whether there is enough balance to make a payment on a bank account returns false for every amount. This is a fault as it is possible that there is enough balance. When someone tries to make a transfer an exception is thrown because of the fault. This results in the user not being able to make a transfer; a failure. The distinction between these terms is used throughout the rest of this survey.

Besides the different types of logs, the log message content itself may also differ. A typical log entry contains a timestamp, a severity and a free text message. A timestamp usually represents the Unix time that an event was recorded and can be used to compare different records. In addition it provides a way to reconstruct sequences of events. A severity level is used to demonstrate the degree of impact that an event has on the operation of a system. Commonly used severity levels are *error*, *warn*, *info* and *debug*[1]. In their book Gu¨lcu¨ and Stark [2003] describe the severity levels as follows. *Debug* messages are usually only used for diagnostic purposes by developers and in some cases system operators. *Info* messages in general contain useful execution information and are not of interest under normal circumstances. Potential events that may be a problem and that a system can recover from on its own are labeled with the *warn* level. *Errors* are fatal to an operation and require the attention of a system operator. Depending on the error recovery mechanisms in place, the application can often continue, but may misbehave. Error log messages are more likely to be linked to a failure or related to an anomaly, and are for that reason often subject of interest in log analysis research. Research does however not limit itself to just *error* level messages. The free text message provides a way for developers to explain what has happened in a specific execution path and for system operators this information can be crucial for diagnosing problems. A typical message containing a timestamp, severity level and free text message may look like this:

2016-12-25 09:44:01 WARN - Problem processing modification request 8345

Different log libraries, however, may produce different messages for the same type of application. The variance in application purposes also has an effect on the composition of log message. E.g., a webserver log focused on user interaction will log differently compared to a super computer with thousands of nodes that communicate with each other.

1Log4j log levels - https://logging.apache.org/log4j/2.x/log4j-api/apidocs/org/apache/logging/log4j/Level.html

7. Literature Survey Research Methodology

Literature surveys are used to discuss and evaluate articles and papers that researchers have written on a specific field of research. To this end a broad range of papers is identified from the research field of interest and an overview is given of what has been investigated. In addition, possible future directions of work are given that are the result of the new insights. A key aspect of a literature survey is that it covers the entire scope of a (sub)field of interest. If important papers are missed this may influence the usefulness of the literature survey. There are several other ways of setting up a literature survey or review, such as a systematic literature review (SLR) or a mapping study.

A SLR is considered to be of higher quality by following stricter guidelines, and aims to eliminate bias [Kysh, 2013]. SLRs are often conducted according to the procedures proposed by Kitchenham [2004]. Her approach aims to present a fair evaluation of a research topic using a credible and traceable methodology. Kitchenham [2004] describes an SLR to be more elaborate and time consuming time than a literature survey, while it introduces several advantages such as being able to provide evidence that the observed is transferable and robust across literature.

Another overview of a research field can be given by means of a mapping study. Mapping studies are conducted to provide a summary and wide overview of a field of research, in contrast to the in depth analysis of an SLR [Kitchenham and Charters, 2007]. Additionally, it is often used to establish whether there exists research evidence on a topic, and to give an indication of the quantity of evidence on that topic. The wide overview of the research area given by a mapping study may suggest more in depth analysis based on a more specific research question, that is to be answered with, for example, an SLR.

Our intention is to carry out a more in depth study than a mapping study. However, the objective is not to conduct a strict SLR, but rather a more lightweight literature survey that gives an overview of a field of research, while including in depth analysis. The thorough procedures by Kitchenham [2004], and the accompanying time required, are therefore not applicable to this literature survey. Inspired by the characteristics of mapping studies and SLRs we followed a more lightweight research methodology for our literature survey which we use for our paper selection phase explained in Section 8. In addition, the methodology captures the selection procedure that enables other researchers to replicate our study.

7.1 Methodology Overview

We describe the research methodology used to select papers related to the topic of interest, while attaining an understanding of the field and allowing a filtering step to obtain a set of high quality work. The methodology is designed to simultaneously

capture the various steps in the paper selection process that enables other researchers to reproduce the search. Identifying relevant and high quality work for a literature survey is an iterative process, especially when one is not an expert on the topic. Iteration takes place to identify more research and to subsequently learn from the findings. Iterating stops when we are satisfied with the obtained set of papers. Once the search phase is over it is time to filter out papers that do not satisfy the intended quality standard of the literature survey. This is an effective step to reduce the number of papers to consider, while maintaining the quality level. The three actions, *search*, *understand* and *filter* are described in more detail in the following sections.

7.2 Search

We describe the workings of several effective search procedures and explain when to use them. Each of these procedures may be applied multiple times in different iterations to get the desired result. For instance, it may well be that after applying all approaches an additional conference related to the topic is found that requires another search. Note that the different search approaches are not limited to what we describe in this methodology.

7.2.1 Database queries

Database queries are especially useful in the initial phase of paper selection. Simple queries based on limited domain knowledge can result in a decent number of papers to learn about the topic. Commonly used databases for search queries in the field of computer science are:

- Google Scholar[2]
- Scopus[3]
- ACM Digital Library[4]
- IEEE Xplore[5]

At a later stage in the search, once more domain knowledge is obtained, more advanced queries can be constructed. It may turn out that different terminology is commonly used or that more detail is required to limit the results to only the topic of interest. For instance, in a topic related to logging in software applications the simple query “logging” tends to yield results related to cutting down trees.

It must be noted that search queries in databases alone are likely not sufficient to

2https://scholar.google.com/

3https://www.scopus.com/

4http://dl.acm.org/

5http://ieeexplore.ieee.org/

capture all relevant work in a specific field. This is because some work may not have been indexed by the databases. Another reason is that important work may not show up in the top results, which increases the chance of it being missed. Finally, the use of synonyms of common terms by authors will make it difficult for some papers to be found using just search queries.

7.2.2 Search by hand

In the previous section we describe that the use of synonyms complicates the search process. This problem translates from searching databases, like Google Scholar, to searching individual conference proceedings or the table of contents of journals. Humans today are still more effective at language processing than computers and for that reason we believe that a human hand can not yet be eliminated in the search phase. We found that the painstaking task of hand-searching the proceedings of conferences and tables of content of journals can result in additional papers that using only a query search would have been missed. A handsearch is conducted by going through the papers one by one and scanning the title and optionally the abstract. To limit the scope of the search one can determine to only consider publications from a certain publication year onwards.

7.2.3 Snowballing

Snowballing refers to using the references or citations of a paper to identify additional papers [Wohlin, 2014]. The search approach can be explained more specifically by *forward snowballing* and *backward snowballing*. Forward snowballing is referred to as using the citations of a paper and considering these as possible related work. Backward snowballing is defined as using the references of a paper to identify previous work in the same field. To limit the amount of work one can decide to only subject top conference publications to snowballing. In addition short papers, like research proposals, can optionally be ignored as they may not present completed research. This does not apply to forward snowballing, as work that cites these papers may be the further work proposed.

7.2.4 Contacts

Although not a search procedure, colleagues, peers and contacts on social media can be a useful source of information. The contacts that have an interest in the topic of the literature survey can often point to valuable research. This can yield an initial set of papers that can be used to obtain an initial understanding of the field.

7.3 Understand

After every search step it is important to determine the value of the identified papers. This knowledge can be used to prepare another search iteration or one can conclude that the search process is complete. For instance, reading the abstracts of the collected papers will give a better idea of the work done on that topic. This

information can be translated into categories to separate the papers into groups. These groups can be used to pinpoint search efforts and can later translate to a textual structure for the survey. Furthermore, one may learn about terminology used and decide to change or extend database search queries to obtain more pinpointed results. Lastly, analyzing the source of a publication, e.g., conference or journal, of each paper can result in the identification of conferences and journals specialized on the topic of interest. This suggests a new round of search focused on these publishers.

We found it to be useful to keep a spreadsheet with metadata about each paper to create a quantitative overview of the identified papers. In addition, it can be used to accurately describe the paper selection phase. The list below contains a list of possible values to record for each paper.

id an incremental value to store the order in which the papers are found

date the date on which the paper is found

full paper to enable filtering on whether the paper is a full paper or not

category one or more categories that fit the paper

citations the number of citations

author the first author

type of publication whether published in a journal, conference, magazine, etc.

publisher where the paper was published

year the year of publication

search procedure which search procedure was used (e.g., forward snowballing or database search)

search query which search query was used to find the paper (optional)

7.4 Filter

The search and understand phase is used to identify a broad range of publications. Because no filtering has taken place, the resulting set of papers can be quite large. It may include content from conferences, workshops, newsletters and journals, all of different levels of quality. This is not necessarily a disadvantage for the search phase as lower quality work can point to previously unknown high quality work. The metadata gathered in the previous phase can effectively be used to facilitate the filtering step.

For instance, the categories that were identified during the search phase can be used to filter out papers that do not match the scope of the topic. For example, one may find several papers related to the visualization of log data, while the topic of interest is log

analysis techniques. Papers in the category “Visualization” may be filtered out. Furthermore, the metadata collected can be used to easily filter on the quality standard of choice. One can decide to only consider conferences and journals, and leave newsletters and workshop papers out of scope. Also, a selection of publishers presumed to publish high quality content can be considered, while filtering out other work. Another possibility is to remove papers published before a certain year as these may contain outdated insights or techniques. Lastly, the number of citations can be taken as criterion. However the year of publication is to be taken into account in this case as new work may not have been cited that often.

There is no set of rules that determines how to filter. This holds for the search and understand iterations as well. It is largely dependent on the topic and field, and is therefore not set in stone. What is important is that paper selection phase is transparent. This methodology enables other researchers to look into and reproduce the selection process allowing them to judge the quality of this purpose.

[1] Project reference: Grant Agreement-731529-STAMP.pdf

and the corresponding proposal, same content but document organization and presentation differ: stamp_ec_Proposal-SEP-210342849.pdf

[2] STAMP quality plan: d71_stamp_quality_plan.pdf

[3] D51 – Industrial requirements and metrics: d51_industrial_requirements_and_metrics.pdf

A link to the most recent version of this document.

8. Paper Selection and Scope

In this chapter we describe the paper selection phase using the literature survey research methodology proposed in Section 7. To obtain the selected set of papers we performed five search–understand iterations explained in Section 8.1 followed by a filter step elaborated on in Section 8.2. Next, we make a categorization among the papers to delineate the different subfields in log analysis in Section 8.3. Furthermore, in Section 8.4 the scope of this survey is given and we point out further research areas of interest. Finally, in Section 8.5 we outline the limitations of the literature survey we conduct.

8.1 Search and Understand

The first iteration started of with an initial set of 21 papers. This initial set contained papers from a variety of conferences and journals published between 1993 and 2015. We complemented this set with 18 additional papers found via Google Scholar query search. Because of the limited knowledge on the topic, three generic queries were used:

1. software logging
2. software AND logging
3. software AND logs

The terms ’logging’ and ’log’ are not only common in the field of computer science. It

may refer to the cutting and on-site processing of trees, the mathematical inverse operation of exponentiation, and any application of recorded activities. To deal with this ambiguity we decided to complement the term with 'software' as this pinpoints the field we are interested in. Interestingly half of the identified papers were published in the year 2000 or earlier. We expected to obtain important recent work via this database search, which was not the case.

A second iteration was performed to capture recent work in the field as well. We went through the publications of nine peer-reviewed conferences and journals by hand. The considered conferences and journals are:

- International Conference on Automated Software Engineering (ASE)
- International Conference on Mining Software Repositories (MSR)
- International Conference on Software Engineering (ICSE)
- International Conference on Software Maintenance and Evolution (ICSM(E))
- International Symposium on Empirical Software Engineering and Measurement (ESEM)
- International Symposium on the Foundations of Software Engineering (FSE)
- Journal of Systems and Software (JSS)
- Symposium on Operating Systems Principles (SOSP)
- Transactions on Software Engineering and Methodology (TOSEM)

We looked at the publications between January 2010 and September 2016 for each of these. The search phase of this literature survey took place in September 2016. For this reason any conferences held or journals published later in 2016 were not taken into account. The hand-search was done by going through the proceedings of a conference and table of contents of a journal. If the title suggested the paper to be relevant we further determined whether the paper qualified as applicable by reading the abstract. The search resulted in 23 papers from the following six conferences and journals: ASE, FSE, ICSME, ICSE, MSR and SOSP. Hand-searching ESEM, JSS and TOSEM did not yield any additional research. By this time we learned that there are several types of logs that are described by the same set of words often used interchangeably. Chapter 5 explains these terms and the ambiguity among them. With a third iteration we wanted to learn whether a previous survey on logging was conducted. To identify papers that are using synonyms for logs we extended the previously used search queries. We proceeded by doing additional Google Scholar searches using the following queries:

1. (audit OR event OR events OR execution OR system OR software) AND (log OR logs OR logging)
2. (SLR OR "systematic literature review" OR "literature review" OR "literature survey" OR survey) AND (log OR logs OR logging)

The search resulted in 12 new papers.

Among these, two papers were the result of the second query, which unfortunately turned out to be of poor quality. In addition, a colleague pointed us to another 12 log research papers. At this point we read the abstracts of the 86 papers so far to identify categories that would help to scope the survey. Each paper was labeled with one or more of these categories. In Section 4.3 we refine the categories and proceed to explain

them in detail.

To account for literature that we missed we proceeded to do backward and forward snowballing in the fourth iteration. We applied snowballing to papers published in the previously listed peer-reviewed conferences and journals minus those that did not yield any results. In addition, snowballing was applied to papers from the following venues as they were found to publish work related to our interest.

- International Symposium on Software Reliability Engineering (ISSRE)
- Large Installation System Administration Conference (LISA)
- Symposium on Operating Systems Design and Implementation (OSDI)
- Transactions on Software Engineering (TSE)

We applied backward snowballing by going through the references of the papers and determined from the title whether a paper was relevant to this survey. In case of doubt we read the abstract and proceeded by making a decision. Again, we only considered work from 2010 onwards. We reasoned that relevant older work would be captured by the newer papers. Backward snowballing resulted in 45 new papers. Forwards snowballing was done using the citations section on Google Scholar. The results for each paper were sorted on citation count and based on the title we determined the relevance of the papers. In case of doubt we read the abstract and proceeded by making a decision. To limit the amount of work we did not consider papers with fewer than 10 citations whilst doing forward snowballing. Forward snowballing resulted in 19 new papers.

By analyzing the publishers of the collected papers so far we found the *International Conference on Dependable Systems and Networks (DSN)* to be of interest as well. In the final fifth iteration we searched the DSN conference proceedings from 2010 onwards by hand. The additional search resulted in two extra papers to end up with a total of 152 papers, which we categorized as described previously.

8.2 Filter

The extensive search procedure resulted in a total of 152 papers. Among these are papers that are not relevant for the scope of this work and were therefore filtered out. Filtering was done according to three steps.

First we determined whether a paper was a full paper or not, filtering out papers consisting of less than 6 pages. The second step was to read the abstract of each paper and determine whether the topic was actually about logging. We found that some works were only using logs as means for other research, e.g., investigating the reliability of software as a service platforms using logs. Other papers were on log-based file systems, which do not fit the scope of this work. After this step the set was reduced to 76 papers. The third step was done in parallel with working on the survey itself. While reading the papers we filtered out those we deemed to be of insufficient quality. Three more papers were filtered out to yield a total of 73 papers.

8.3 Categorization

The collected papers were labeled with one or more categories to distinguish between sub-fields in log analysis. Using the abstracts of the papers we distilled groups of similar papers and came up with a category for each group that we believe describes it accurately. The categories as we defined them are as follows:

Abstraction describes how to deal with large numbers of logs

Detection describes how to identify errors and anomalies as well as to diagnose problems

Enhancing describes the practices to improve logging

Parsing describes how to extract message templates from logs

Modeling describes how to extract a model or finite state machine from log messages

Scalability describes how to apply log techniques on a large scale

Visualization describes how to visualize logs to gather new insights, e.g., anomalies

The labeling of the selected papers was not set in stone and was subjected to change if reading them in detail yielded new insights. Furthermore, the number of categories per paper was not limited by one, as a paper can fall into multiple categories. The categories were used to make a logical split between the different subfields in logging. A more in-depth look into the categories made us realize that papers in the category scalability were best used as enrichment of the other categories. Scalability namely deals with applying approaches and techniques on a larger scale, and may be related to any of the categories.

8.4 Scope

Log analysis comprises a large field of research and is therefore not easily explained. For this reason, we are not able to give a full overview of the entire log analysis field in this work. We scope around a subset of the categories defined in Section 4.3 and leave the remaining categories open for a future work.

In this work we aim to describe the full process of effective log analysis. Starting with *abstraction* techniques, including log *parsing*, to reduce the amount of data to process in further steps. We proceed to explain the techniques used for *detection* and diagnosis of failures and anomalies. Finally we discuss the work with the objective of *enhancing* the practice of logging itself. In addition we elaborate on the *scalability* aspect of log analysis by enriching the overview of the previously described categories.

We leave work related to *modeling* of logs for future work. For example, research that falls within this category is the work by Tan et al. [2008] who introduce SALSA, a tool that derives a state machine from system logs, which can be used for failure diagnosis. Also, *visualization* related work is left outside the scope of this survey. These papers are concerned with leveraging visualization techniques to provide insights into log data. Aharon et al. [2009], for instance, transform massive numbers of log messages into visualizations that provide meaningful insights that help operators detect problems.

8.5 Limitations

In this section we outline the possible limitations of the conducted literature survey. These limitations may have caused us to overlook relevant work in the field of log analysis.

The search queries were only run on one scientific database: Google Scholar. A possible threat to validity is that Google Scholar likely does not contain all the research on log analysis ever published. Other scientific databases like Scopus may be able to return the results missed by Google Scholar. To account for the possible misses we applied snowballing and performed hand-searches on the publications of several conferences and journals.

Both forward and backward snowballing were found to produce a large amount of additional work in the field. We applied a single iteration of snowballing where we considered the previously found papers as input. However, ideally, one would recursively apply snowballing on new findings until there are no new results. The downside of this is the amount of work that accompanies each recursion step.

In the selection phase we determine based on the title and optionally the abstract whether a paper was relevant or not. This involves human assessment which can involve a bias. One person may classify a paper to be related to logging, while the judgment of another person may result in the contrary. The assessment of the papers in this survey was done by one person. Ideally, two or more researchers would separately apply the paper selection methodology. After each iteration the results would be compared. Any mismatches in judgment can then be analyzed in more detail until consensus is reached. The bias of the opinion of one author can hereby be reduced. Similar reasoning can be applied to the filtering step in which we determine whether the main topic of a paper is logging. In this case there also is the bias of one assessor, which can be reduced by having multiple people assess the papers in parallel.

Lastly, due to time constraints, we were able to cover 36 of the 73 selected papers for the scope of this survey. This may be a threat to the completeness of this survey. These works may introduce techniques different to those that we describe and may influence the discussion. Future work may report on these papers to complete our overview.

9. Abstraction Techniques to Simplify Log Analysis

Nowadays, enterprise system easily produce millions of logs on a daily basis with log data reaching tens of terabytes [Lin et al., 2016, Xu et al., 2010]. One can imagine that line by line log analysis becomes unfeasible quickly with the growth of systems and the amount of information they log. Often log message specific information is not required until a later stage in log analysis, for instance during failure diagnosis. Until this time, abstraction techniques can be employed to reduce the number of log elements to consider during analysis. Abstraction is generally a necessary step.

Depending on the unstructuredness of the data the number of possible unique log messages can be enormous. Drawing conclusions from this vast number of possibilities may be impossible without a simplification step.
Parsing log messages is one way to reduce the number of elements to analyze. Instead of single log messages the corresponding message templates are considered as events, which greatly reduces the number of possibilities to consider. Another way is to filter out log messages redundant for analysis and thereby reducing the size of the input.
We describe several types of log parsing methods in Section 9.1. In Section 9.2 other abstraction methods are discussed. Finally, in Section 9.3 we elaborate on whether abstraction can benefit the quality of log analysis.

9.1 Log Parsing Methods

A typical log message contains a timestamp, a verbosity level (e.g., WARN) and raw free text content for convenience and flexibility. This raw, unstructured message content provides a way for developers to explain what happened in a specific execution path and for system operators this information can come in handy while diagnosing problems. The free text is clearly useful for human interpreters, but machines have no effective way of processing natural language. The lack of structure is a problem for automated log analysis using data mining techniques for tasks like failure detection, anomaly detection and failure prediction. Furthermore, free text in log messages implies a near endless number of possible messages. This makes finding patterns and discovering anomalies a difficult task. Log parsing can be used to simplify the number of possibilities to consider and therefore make log analysis more effective [He et al., 2016].

Log parsing is the practice of transforming raw unstructured logs into specific events that represent a logging point in an application. Typically the raw content consists of a *constant* part and a *variable* part. The constant part is the fixed plain text from the log statement which remains the same for every log message corresponding to that statement. The variable part contains the runtime information, such as the values of parameters. The objective of log parsing is to separate the constant and variable part of the log message. Its result is commonly referred to as a *message template* or *log event* representing the log messages printed by the same log statement in the source code [Xu et al., 2009, 2010, Lin et al., 2016]. The following example shows a log message and its corresponding message template. Variables are typically denoted by asterisks (*).

Authentication failed for user=alice on host 172.26.140.21 - attempt 3 Authentication failed for user=* on host * - attempt *

Just as applications evolve over time, so do log templates in source code [Kabinna et al., 2016]. Log templates change and new ones are added, which is considered an obstacle for log parsing methods. Reapplying log parsing after every release may be unfeasible and influence log analysis due to varying input. To solve this problem a log parser should be able to deal with changes in templates by detecting these.

We characterize four different groups of log parsing methods: heuristics methods, clustering methods, source code based methods and other methods. The groups are discussed in the following sections.

9.1.1 Heuristic Methods

A heuristic is a technique that is used to approximate a solution where an exact solution can not be found in reasonable time. The approximate solution may not be the best solution, but may be an adequate solution. Heuristics can be used in combination with other techniques to increase their efficiency. Often they are rules based on empirical data or theory. Heuristics used in log parsing consider the semantics of log messages. For instance, numeric values are less likely to belong to the constant part of a log message. They generally represent variables like process identifiers, parameter values and IP addresses. Salfner and Tschirpke [2008] make use of this heuristic and replace numeric values by placeholders to reduce the number of messages from a telecommunication dataset by 99% to a set of log templates. Gainaru et al. [2011] use the same heuristic in their clustering technique that splits logs up based on the maximum number of constant words in a certain position. To improve the effectiveness they give low priority to numeric values as these tokens have the most chance of being variable.
Besides numeric values, log messages also tend to contain special symbols used to present variables. These special symbols include colons, braces, and equals signs. In the log message “Invalid amount for currency: EUR with amount=100” the variable values found would be “EUR” and “100”. Fu et al. [2009] apply this more extended heuristic and are able to reach an accuracy of 95%. In other words, they are able to discover 95% of the message templates using just this heuristic. Jiang et al. [2008b] use a similar technique where they reason that the equal sign is used to separate a static description and a variable value. Furthermore they introduce a heuristic that recognizes phrases like “[is are was were] value” to capture more variables.

An obvious downside of using the numeric heuristic is that there may be constant values that are numbers. Important log templates may be missed by log parsers employing this heuristic. Furthermore, semantic heuristics vary per case and therefore require human input and application-specific knowledge to set up the log parser. On the contrary, the results are promising and the approach is computationally inexpensive. In addition, heuristics do not have to be used as the only mechanism to parse logs. They can be used as preprocessing step for a more computationally expensive parsing method or can be used in combination with a method such as Jiang et al. [2008b] demonstrate.
He et al. [2016] empirically evaluated the usefulness of preprocessing the logs using

heuristics. They found using domain knowledge to remove, for example, IP addresses, improves log parsing accuracy.

9.1.2 Clustering Methods

Data clustering is a technique in data mining and machine learning that groups entities into clusters. Each item in a group is similar to other items in that group, but dissimilar from items in other groups. Clustering techniques are usually employed for the classification and interpretation of large datasets.
In log analysis clustering can be a useful first step to reduce the number of elements to deal with. Further analysis may still be computationally expensive, but practically feasible due to the reduction of the input size. We found that clustering techniques tend to leverage the fact that words that occur frequently are more likely to belong together. Furthermore, the methods make use of a user defined threshold to determine the granularity of the clusters.
One of the first clustering tools used for log parsing was introduced by Vaarandi et al. [2003]. The tool, Simple Logfile Clustering Tool (SLCT), is based on the Apriori algorithm designed for frequent item set mining [Agrawal et al., 1994]. SLCT makes a few passes over the data. The first pass is used to mine frequent words. A word is frequent if it occurs at least N times in the same position, where N is a user defined threshold. Subsequently the messages are clustered based on whether a message contains one or more frequent words. The resulting clusters correspond to message templates as the frequent words and their corresponding position in the messages form the constant part. Based on an empirically determined threshold value, dependent on the nature of the log file, the algorithm can be tweaked to better discover frequent patterns and reduce the number of outliers. Its disadvantage is that it only finds clusters based on lines that appear frequently. Therefore it may miss important rules that are not so frequent. Furthermore, Apriori-based tools, like SLCT, are already costly in terms of computation for a small number of log messages. The ever increasing size of log files is a problem for these methods.
After the introduction of SLCT, other clustering algorithms for log parsing have been introduced to overcome its limitations. Makanju et al. [2009] introduce a clustering algorithm, IPLoM, that is not based on the Apriori algorithm. In three steps they apply iterative partitioning where logs are split based on individual word count, least variable token position and bijective relationships between tokens. The resulting clusters represent similar message patterns to those proposed by Vaarandi et al. [2003]. It was found that the iterative partitioning approach outperforms the Apriori based algorithm in mining the message patterns.

Hierarchical Event Log Organizer (HELO), a tool proposed by Gainaru et al. [2011], deals with the limitations of SLCT. HELO first considers the entire log file as a group and partitions it until a specified threshold is reached. Instead of mining algorithms that split based on information gain, HELO splits on the position that contains the maximum word frequency. These words are likely to be constants. They are able to do the clustering in $O(n \log n)$ runtime. To improve the accuracy, the authors consider the heuristics described in Section 5.1.1. HELO is designed in such a way that it can adapt to changes in incoming messages. In an online fashion it checks whether a message falls within a

cluster. If not, re-computation occurs and based on a threshold the message is added to a cluster anyway, or a new cluster is formed. With this approach HELO is able to deal with software updates and the accompanying change in log messages.

Many algorithms make use of word frequencies at certain positions in a message. A much simpler approach is utilized by Jiang et al. [2008b]. After using heuristics to identify variables they group together the abstracted messages based on the number of constant words and number of variables. For instance, all messages containing 5 constant words and 2 variables are grouped together. This is not the final step in their log parsing method, but allows further steps to work on a cluster level rather than the abstracted messages. Jiang et al. [2008b] evaluated their approach on four enterprise applications and compared it to the association rule learning technique by Vaarandi et al. [2003]. Their approach significantly outperforms Vaarandi's due to the disadvantage of finding only clusters based on lines that appear frequently.

In their evaluation study on log parsing He et al. [2016] found that cluster-based log parsing methods like SLCT and IPLoM obtain high levels of overall accuracy. This especially holds when domain knowledge based rules are used for preprocessing. The evaluated methods, however, do not scale well on large sets of log data. The authors suggest parallelization of clustering based parsing methods to reduce the runtime, but do not further investigate this. Furthermore, determining the threshold for clustering algorithms is a time consuming task. A task that seems to increase in complexity as the dataset increases in size and may become a bottleneck because the threshold will vary in different use cases.

9.1.3 Source code based methods

Instead of analyzing the log messages to obtain a set of message templates, one can parse to the source code to achieve the same goal. When using third-party software for example, the source code may not be available, so other methods have to be employed to obtain message templates.

The advantage of source code parsing is that it is computationally cheap. Furthermore, higher levels of accuracy can be reached [Xu et al., 2009]. For example, messages that rarely occur are still turned into a template, while log heuristics or clustering algorithms may miss them. However, source code parsing for log parsing is not as straightforward as it seems. Consider the following Java log statement.

```
log.info("Machine  " + m + " has state " +  STATE);
```

Several difficulties arise in parsing this statement. The parser needs to know which logging library is used and which methods it uses to log, for instance, *log.info()*. In this case *STATE* seems to be a variable, however *m* could be an instantiation of a machine class which overrides the default *toString()* method. This method itself may return a constant part intermixed with variables, which are more difficult to parse. The latter problem tends to occur in Object Oriented languages (e.g., Java).

Using the abstract syntax tree (AST) of Java source code Xu et al. [2009] address the challenge of log parsing using source code. Their log parser requires the logging class as optional input. They take the partial message templates from the logging statements and complement this with *toString()* definitions and type hierarchy information to overcome the described problem. In follow up research Xu et al. [2009] apply their log parsing technique to C/C++ programs and evaluate their parser to be successful by being able to match of 99.98% of the log messages to a template. Also in this case parsing is not as straightforward as C/C++ programmers often utilize macros that complicate analysis. In similar manner, Yuan et al. [2010] use ASTs to create message templates and extract message variables.

The parsers proposed are not easily applicable to a wide range of applications, in contrast to various clustering based algorithms. This is due to the differences in programming languages and log libraries. Customization is required to ready the parser for different applications.

9.1.4 Other log parsing methods

Next to heuristic, clustering, and source code based methods for log parsing we also consider methods that do not fall within these groups. This section describes these methods.

The edit distance metric, also called Levenshtein distance metric [Levenshtein, 1966], is a measure used to quantify the similarity between two strings of tokens. This distance is represented by the number of operations required to transform one string into the other. The operations *insertion*, *deletion* and *substitution* contribute to the edit distance.

In log parsing this metric is used to determine whether two log messages may or may not belong to the same message template. Words are in this case considered to be the tokens. Salfner and Tschirpke [2008] employ the edit distance metric, after using heuristics, to every pair of log messages to improve the effectiveness of their log parser. They were able to reduce the number of messages by 99% using heuristics and increase this percentage to 99.92% using the edit distance metric. Fu et al. [2009] calculate this metric for every two log messages as well, after applying heuristics to remove parameters described earlier, and group them based on a threshold value.

The edit distance metric has a theoretical runtime complexity of $O(n^2)$ and can therefore be unfeasible for large amounts of logs. A limited set of log messages, a training set, can be used to reduce the time required to find log events. However, messages that occur less frequently have a higher chance of being clustered incorrectly or may even not be detected. The technique can be suitable after a possible first clustering step that greatly reduces the number of elements to consider.

Jiang et al. [2008a] introduce a message template extracting technique that they use on clusters of abstracted messages. Each cluster contains messages with the same number of constants and variables. The first message in a cluster will become a message template. For subsequent messages it is checked whether they match an existing

message template. If not a new template is made. This approach can be quite expensive when applied to unclustered groups of abstracted messages. The clustering step however can make this approach feasible.

There exists a wide range of log parsing techniques, ranging from simple heuristics to complex clustering algorithms. Often, not a single technique is used, but a concatenation of multiple approaches. It seems to depend on the type of logs and the use case, which set of techniques is suited. The most effective way of log parsing seems to be based on source code analysis. However, source code is not available in all circumstances and other approaches are therefore necessary.

9.2 Other Abstraction Methods

Once an anomaly occurs within a system that produces warnings or failures, those anomalies tend to appear until the problem has been resolved. Events of the same type that occur within a short period of time from each other possibly do not introduce any new information. The redundant amount of information can be filtered out to reduce the load on further processing steps. In addition, in multi-node systems related warnings or errors can be generated in multiple locations. Also in this case logs may be filtered to reduce the number of log messages. Sahoo et al. [2003] employ a simple filtering mechanism where they remove sequential duplicate events caused by a check that occurs multiple times and by automatic retries. They were able to reduce the multi-node cluster logs by 90% using this simple filtering mechanism. A similar approach is applied by Fu et al. [2012] and Zheng et al. [2009] who, based on a threshold value, use the interval between the similar events to determine which logs to filter out. Additionally, Zheng et al. [2009] filter similar messages produced by multiple nodes and are able to compress by 99.97%. The authors verify that their filtering method can preserve useful failure patterns by comparing its failure precision and recall to methods proposed in earlier work. With this approach however, information about a stream of warnings is lost. To tackle this problem one can record the start and end time of such a stream including the number of occurrences and optionally the locations [Zheng et al., 2009].

Filtering mechanisms seem to be generally applied to logs from multi-node systems which have a tendency to produce a large number of similar log messages. A reason for this may be that the logs produced are more often on a component level (e.g., hardware connectivity logs) that are quite generic, and therefore allow for a potentially high compression rate. In contrast to, for example, software as a service systems that may record more variable data such as user interaction with the system. The variety of log data may cause filtering to not work well, making filtering mechanisms ineffective.

9.3 To Abstract or not to Abstract

Abstraction techniques are a great way to deal with extensive amounts of logs and reduce computation time in further analysis. In case of Apriori-based algorithms like Classification-Based Association [Ma, 1998] log analysis without log abstraction as a

preprocessing step is not even feasible for larger log datasets [Huang et al., 2010]. Abstraction in some cases is therefore a must. However, abstraction may also result in a lower precision or accuracy of further analysis.

Huang et al. [2010] apply machine learning based problem determination approaches to abstracted and non-abstracted log data. They verify that log abstraction can be successfully used to reduce the size of log data, thus improving the efficiency of analysis. Two associative classification methods, Classification based on Multiple Association Rules [Li et al., 2001] and Classification based on Predictive Association Rules [Machado, 2003], were found to outperform Naive Bayes and C4.5 [Quinlan, 1993], a decision tree learning algorithm, when non-abstracted data was considered. The precision of the associative classification methods however was not better when applied on abstracted logs.

Unfortunately, the precision of the methods and whether the logs were abstracted or not was not compared to the runtime of the methods. This could have given valuable insights into the trade-off between abstracting and not abstracting, and which type of method to use.

10. Log Analysis: Detection and Diagnosis

In the previous chapter we elaborated on abstraction techniques that reduce the number of log messages to consider to enable detection and diagnosis. In this chapter we discuss the various use cases that log analysis has and the different techniques that can be applied to that end. We consider failure detection, anomaly detection and failure diagnosis, and describe them as follows. Failure detection is the task of discovering that a system fails to fulfill its required function. Detecting that a system deviates from its normal or expected behaviour is referred to as anomaly detection. An anomaly does not necessarily indicate a failure, however, it may imply an extraordinary event that may lead to a failure. Over time, common failures may be easily recognized and fixed by operators. However, unknown or infrequent failures can be hard to resolve without knowing the origin of the problem. Failure diagnosis is the practice of overcoming this and aims to identify the root cause of a failure. The techniques used for detection and diagnosis are not specific to one particular use case, but can be used for all kinds of log analysis. For this reason we give an overview of the log analysis techniques in Section 10.1 and refer to them in the subsequent sections. In Section 10.2 we compare the failure detection techniques. Section 10.3 describes techniques used in anomaly detection and finally we elaborate on failure diagnosis in Section 10.4.

10.1 Log Analysis Techniques and Methods

The techniques and methods used in log analysis are not solely applicable to one of the analysis categories. For this reason we introduce the techniques and methods first and explain how they can be used in log analysis.

10.1.1 Associative rule learning

Associative rule learning is a supervised learning technique that discovers strong relations among variables [Agrawal et al., 1993]. The key idea behind association rules is that if a certain set of events has occurred then another event is also very likely to occur. The technique was introduced to mine relationships between items in a database, e.g., grocery items. A use case Agrawal et al. [1993] give is finding all the rules that have “diet coke” as consequence, which helps plan what a store should do to increase the sales of diet coke.

In log analysis associative rule learning can be used to translate sequences of events into rules [Lou et al., 2010]. For instance, three events that occur within a close time interval may indicate there is a high probability that a fourth event will occur. By mining associative rules from log events one can capture the execution behaviour of an application. In a similar manner, associative rules can characterize recurring failures that manifest themselves via a repetitive event signature.

10.1.2 Chi-squared test

The chi-squared test [Pearson, 1900], a simple statistical hypothesis test, can be used to compute the probability that two data vectors originate from different distributions. An application in log analysis of the chi-squared test is anomaly detection. By testing whether a sample of new log events originates from a different distribution of events that represent normal system behaviour it can be determined whether that sample is an anomaly.

10.1.3 Decision tree learning

Decision tree learning [Quinlan, 1986] is a supervised learning method used to create a model, a decision tree, that can be used to classify samples based on a number of input variables. Each node in the tree denotes one of the input variables, and based on the value of a variable a different edge is traversed. After descending the tree a sample is classified as the class linked to the leaf node.
In log analysis, decision trees are used to detect recurrent failures [Bose and van der Aalst, 2013, Reidemeister et al., 2011]. The tree encapsulates the characteristics of known failures. New log message events can be mapped to the tree to classify them as a failure or not.

10.1.4 Hierarchical clustering

Hierarchical clustering [Maimon and Rokach, 2005] is an unsupervised learning technique based on the idea that objects near each other are more related than objects farther away based on one or more attributes. What these attributes are and how they are computed differs from use case to use case. A hierarchical clustering technique is used when one does not want to make assumptions about the number of clusters, in contrast to techniques like k-means [MacQueen et al., 1967]. There are two strategies when it comes to hierarchical clustering: agglomerative clustering and divisive clustering [Maimon and Rokach, 2005]. Agglomerative clustering is a bottom up

approach where each object starts as a cluster and the number of clusters is reduced by combining two clusters. Divisive clustering is a top down approach where all objects start as one cluster, which is split repeatedly into smaller clusters. Clustering is not only used in log parsing as described in Section 5.1.2, but has an application in log analysis as well [Lin et al., 2016, Kc and Gu, 2011]. On a node or program level it can be used to detect anomalous instances based on characterizing features. Instances that experience erroneous behavior in this case do not end up in the expected cluster and therefore may indicate an anomaly.

10.1.5 Naive Bayes

Naive Bayes [Russell et al., 2003], a family of simple probabilistic classifiers, is used to classify based on features that are considered independent of each other. The applications of the supervised learning approach range from spam detection to categorization. A Naive Bayes classifier considers each of a set of features to contribute to the probability that an item falls within a classification. The known shortcoming of Naive Bayes is that even though features may be dependent on each other it will consider them as independent. Its main advantage is that it can greatly outperform more sophisticated algorithms in terms of runtime and that, in spite of its disadvantage, it works well in practice.
Its application in spam detection intuitively explains its relevance in anomaly detection [Bodic et al., 2005]. In anomaly detection the exact probability whether something is an anomaly or not is not crucial; a gross estimate is sufficient. This makes Naive Bayes robust enough for the classification of anomalous events.

10.1.6 Support vector machine

A Support Vector Machine (SVM) [Vapnik, 2013], used for classification, is a supervised learning model that, given a training set, builds a prediction model. Based on a set of labeled examples it attempts to find a way to separate the data into two groups. This model is used on new data to determine whether to label it as one of two categories, often success and failure. Its goal to classify data in one of two categories makes SVMs useful for failure detection [Bose and van der Aalst, 2013, Fronza et al., 2013].

10.2 Failure Detection

When a system is unable to conform to its specifications and therefore cannot carry out its intended function we speak of a failure [Radatz et al., 1990]. Failures can only be found by executing a program, so analysis is often limited to log messages. Even then, finding a failure can be troublesome due to the vast number of log messages produced. Failure detection in log analysis aims to ease this process and aims to catch failures. To this end we discuss the variety of techniques and methods used in literature.

The failures easiest to detect are failures that have occurred before. Decision trees are useful for detecting these recurrent failures. Usually such a tree is built as follows. The root node contains all information and based on the most distinctive attribute the tree is split. A threshold is often set to prevent overfitting at the leaves. Reidemeister et al.

[2011] use decision tree learning to detect recurrent failures. Their tree is constructed in such a way that it captures the symptoms of known failures. The tree is used to represent the log files in such a way that it can be used to quickly classify new logs. A disadvantage of this approach is that it does not capture previously unknown failures. However, if a system has limited functionality and is unlikely to change, such an approach may well fit the requirements.

Another way to detect recurrent failures is to capture them in a knowledge base. Creating a knowledge base of all previously detected failures and adding new failures when they occur is a simple incremental approach to detect recurrent failures. Although requiring an initial effort to tag failures in the beginning, over time the effort will be less. The advantage of such techniques, as opposed to machine learning techniques, is that failures are manually classified by an experienced human operator. Lin et al. [2016] use such a knowledge base, containing log sequences that represent clusters of logs, to automatically detect recurrent failures. Their knowledge base contains mitigation actions associated with known failures which are returned to the operators once a failure occurs. Unknown failures and their corresponding solution are added to the knowledge base to ease future failure recovery.

Instead of capturing failures in a knowledge base, associative rules can be used to create a signature of traces of events. Using log features combined with strongly correlated class labels (normal or faulty), Bose and van der Aalst [2013] exploit associative rule mining to discover these patterns in event logs. The signatures are used to classify known and unknown traces of events. In addition, the generated rules are easily understood by domain experts for faster diagnosis.

Another approach for failure detection is the use of SVMs, explained in Section 10.1.6. What makes SVMs hard to apply in failure detection is that in logs non-failures are over-represented compared to failures. In data science terms this means that the dataset is highly skewed. Fronza et al. [2013] use SVMs to classify operation sequences from session log files. They used a weighted SVM to deal with the problem of the skewed data. Bose and van der Aalst [2013] use SVMs for the same reasons to classify events, but use this as an intermediate step in mining signature patterns.

An advantage of decision trees and associative rule mining over the SVM approach is that the former give contextual information to the human operators when diagnosing a failure. In other words, after detection it is easier to find the root cause of a failure. In contrast to SVMs, associative rule mining is able to handle imbalanced datasets where class instances are not equally represented, although the weighted variant of SVMs attempts to deal with this shortcoming.

10.3 Anomaly Detection

Anomaly detection is the practice of identifying abnormal behaviour in the normal flow of operation. There are two directions to approach anomaly detection. One way is to identify events or sequences of events that do not regularly occur in a system's execution

and consider them as an anomaly. The other way is to determine the normal execution behaviour of a system and consider all other behaviour as anomalous.

Kc and Gu [2011] employ a divisive hierarchical clustering technique as a coarse-grained step in their two step approach to detect anomalous program instances. Clustering is done based on the Message Appearance Vector log feature, which captures the appearance of message templates in each log file. This clustering step alone is insufficient to detect anomalous instances effectively, but does allow a more fine-grained second step to do this using the clusters. The approach focuses on capturing normal execution behaviour to classify other behaviour as anomalous. Algorithms for anomaly detection do not have to be overly complicated, and the nearest neighbor algorithm is an example of this. Kc and Gu [2011] use a nearest neighbour approach within clusters of log messages with similar message types to detect anomalies. By first clustering the messages they limit the processing overhead. As they extract message templates from the application source code they are also able to ease the diagnosis process.

Lin et al. [2016] propose LogCluster that makes use of agglomerative hierarchical clustering as a first step to detect anomalous sequences. They use a vector based on the event types that occur in a log message and based on the differences between the vectors apply hierarchical clustering. From the clusters they extract a representative log sequence which they compare to previously extracted sequences to see if the sequence is new and a possible anomaly. Similar to Kc and Gu [2011] they do not make assumptions about the number of clusters, but rather use an empirically defined distance threshold to stop the clustering. It is interesting to see both the agglomerative and divisive approach being employed. Unclear is however what the advantages of choosing one over the other are.

Instead of using system events as means to anomaly detection one can also look at application user behaviour. If this behaviour suddenly changes this can indicate an anomaly, e.g., a page is down or a database does not reply. Bodic et al. [2005] use the chi-squared test to determine, from website access logs, anomalous page visit behaviour. Using this information they detect database outages and updates that introduce bugs to webpages. Another technique to detect sudden changes in behaviour is analysis using a Naive Bayes classifier. In addition, Bodic et al. [2005] applied this technique to website access logs and found Naive Bayes to be useful for different anomalies than the chi-squared test. Naive Bayes was found to be sensitive to increases in *infrequent* events, while the chi-squared test turned out to be more useful in case of increases and decreases in *frequent* events. The first technique may pick up the anomaly first, while the second takes a few hours. Which technique picked up the anomaly first gives valuable insights about the nature of the anomaly to the operator. The authors show that combining multiple approaches can result in better anomaly detection. The viewpoint of user behaviour instead of that of system events is an interesting angle that can be interesting for web-server applications and Software as a Service (SaaS) applications.

Similar to associative rule mining in failure detection, relationships that capture normal behaviour can also be used in anomaly detection. A statistical approach is to deduce a set of linear relationships among logs of the same type. More specifically, to obtain the

relationships between log messages and the ratio of relative occurrence that they have. Such an approach can quickly become an NP-Hard problem to solve due to the vast amount of variance between log messages. Lou et al. [2010] introduce a linear relationship mining technique that mines invariant-based groups of logs containing the same program variable. Determining the different logs that use this variable leads to a program execution path, which is in turn used to obtain an invariant. According to the authors the computational complexity can be controlled in practice, however they do not explicitly show this.

10.4 Failure Diagnosis

Failure diagnosis is essential to ensure failures do not occur again in the future. Knowing a failure occurred is one thing, but finding the root cause of the problem is the next challenge. Although logging the right information helps, operators still have to go through large numbers of related log messages to figure out the origin of a problem. Automated failure diagnosing is therefore a must for the successful operation of everyday systems.

For effective root cause analysis an operator would like to know the runtime behaviour of an application. Finding out what must have happened post mortem is difficult when one only has access to runtime logs. Efforts have been made to infer execution paths close to points of failure. Diagnosing problems using these execution paths can save valuable time. Yuan et al. [2010] use a combination of source code analysis and runtime logs to do just that, and combine it into the tool SherLog. Using the variables from log statements and combining them with conditions in the source code they create a set of regular expressions to match all possible runtime logs. Unfortunately, their approach does not work across threads, and processing the failures they used for evaluation takes up to 40 minutes. However, their approach does open an interesting perspective on failure diagnosis using a combination of runtime logs and source code.

Decision tree approaches are, besides failure detection, also used for failure diagnosis. At each node the tree is split based on a log attribute or feature. The path followed in the tree while classifying new data can give insights in the origin of a failure. Chen et al. [2004] use this approach to diagnose failures in network applications. They split at a node based on several attributes, including thread id, host, process id and request type. Similar to the failure detection technique by Reidemeister et al. [2011], Chen et al. [2004] split on the attribute that gives the biggest gain in information and use a threshold to avoid overfitting. An operator can use the decisions made during the classification of an event to narrow down the scope for root cause analysis and hence decrease the time required for diagnosis. The authors show with this technique that logging operational attributes in addition to a string message can be useful.

While a data reduction is of key importance to make detection possible, it is evenly important to be able to access the more detailed information once detection has taken place. Without clear pointers to what may have happened the task of root cause analysis is greatly complicated.

11. Log Quality Enhancements

Many methods have been developed to deal with the ever growing amount of log data. So far we have discussed how to abstract over logs to make analysis more comprehensible and in some cases even more accurate. Furthermore, we have given an overview of techniques to detect failures and anomalies, and how to improve failure diagnosis. We however, have not discussed how to improve the practice of logging itself. It was found that, in open source projects, developers often do not get a log message right the first time, and also, log statements tend to be changed more often than other pieces of code [Yuan et al., 2012b]. In a hindsight, verbosity levels are changed or new variables are added. This may indicate that the practice of logging is ad hoc.

An approach to benefit all of the mentioned aspects of logging is to enhance the log quality itself such that the task of log analysis is simplified. Potentially fewer log lines have to be written and analysis effort can be reduced. We distinguish three problems in log quality enhancement and elaborate on them case by case. First of all *what to log* is discussed in Section 11.1. Second, the problem of *where to log* in elaborated on in Section 11.2. Finally, Section 11.3 describes *how to log*.

11.1 What to Log

Writing a log statement is one thing, but logging the right information useful for debugging and problem diagnosis by operators is another. Log statements are often written by different developers that typically do not write these messages with an operator perspective in mind. Statements are not written to diagnose complex problems or structurally designed to use in failure diagnosis. This can be a problem when one wants to do exactly that.

Logging the right variables can significantly improve the time required for failure diagnosis. Yuan et al. [2012c] propose a system, LogEnhancer, that enriches log statements with extra information to achieve this. With their approach, based on source code analysis, they add on average 14 variables to every log statement to ease diagnosis. They evaluated LogEnhancer by checking how many variables, added by developers to log statements over the years, were inserted by the tool, and found this to be 95% of the variables. By adding the variables they are able to reduce the number of potential paths that need to be checked while diagnosing a failure as this precludes the occurrence of certain paths. The overhead that is the result of this approach is evaluated to be between 1.5% and 8.2% on the tested systems. In addition, the authors suggest that log inference tools [Yuan et al., 2010] can benefit from this work, however, they did not verify this.

After introducing their log diagnosis tool, Yuan et al. [2010] observe that recording the *thread id* in concurrent programs is a good guideline. This makes diagnosis easier as log messages can be separated out of each thread. In addition to logging the thread id Lin et al. [2016] and He et al. [2016] mention in their discussion that adding an *event id* to

each log statement is good practice. This way valuable time is saved as there is no need to parse log messages to events anymore; each log message already contains a corresponding event id. The authors suggest that tools can be built to automatically add these event identifiers before source code is submitted to a central repository. Yuan et al. [2010] propose a similar idea where they would like to log the filename and line number of the log statement.

Overall, it seems that by adding variables, e.g., an event id, log analysis can be effectively improved. Unfortunately, this approach has not be investigated in detail yet and it is therefore unclear what the implications are.

11.2 Where to Log

Although intuitively logging in more places seems to be better, this is not the case. Strategic logging, log placement to cover necessary runtime information, is difficult to practice without introducing performance overhead and trivial logs.
While complementing existing logs with valuable information simplifies problem diagnosis, it does not solve the problem of missing log messages. Yuan et al. [2012a] show that in 250 failures, in 5 widely used systems, in 57% of the cases detectable errors were not logged. If these errors were logged they could have eased diagnosis by a factor of 2. To improve diagnosis the authors present Errlog, which identifies potential unlogged exceptions and automatically inserts log statements in C/C++ source code. Reportedly the tool can speed up failure diagnosis by 60% while introducing only 1.4% of runtime overhead. Unmentioned is that log statements which are computer generated can also be inaccurate. A human developer could add more context and meaning to the log message making it more effective. Therefore an interesting angle of research would be to use the approach of Yuan et al. [2012a] to make log statement suggestions in the IDE. This allows developers to decide whether the statement is useful and to optionally add more information to it if required.

Further improving the effectiveness of logging would be easy if one knows where to add logging messages. Cinque et al. [2010] applied fault injection to investigate log coverage. Enabling exactly one fault at a time and assessing whether the logs capture the resulting failure or not allows the quantitative determination of this. Injected faults that do not result in a trace in the collected logs point to potentially missed logging locations in the source code. The authors found evidence of the injected faults in only 40% of the cases in the systems under test. Using a ranking mechanism the authors are able to prioritize the suggested logging locations to find a balance between the overhead of logging everything and detecting the most common faults.

If log statements are added in the wrong place in the code they will not be as useful. Cinque et al. [2013] describe a simplistic coding pattern that fits 70% of the log statements and involves inserting log statements at the end of an instruction block (e.g. *if* statements and *catch* blocks). Similarly, Pecchia et al. [2015] report that the simplistic pattern of adding log statements after *if* statements is used in 60% of the cases. This pattern leads to ineffective log statements resulting in 60% of failures to go unnoticed

[Cinque et al., 2013].

Cinque et al. introduce a rule based approach, that instead of the instruction block focused approach takes the entire design of an application into account. Using conceptual models, UML diagrams and the system's architecture they deduce a set of formalized rules for writing log statements. By means of software fault injection the authors verified their approach to log 92% of the failures while reducing the necessary number of log messages. The authors state the results are promising, although the approach requires an initial investment to translate design artifacts into input for the code instrumentation tool.

Knowing what to log is not the only aspect in good practice. Simplistic logging patterns result in failures to go unnoticed. For this reason more thought is required to determine where to log, not only to catch more failures, but also to reduce unnecessary logging.

11.3 How to Log

How to log deals with the development practices of writing log statements. It is concerned with the logging strategies and the observations that can be made about it from an application perspective.

Log messages are likely to change over time [Kabinna et al., 2016]. Information may be missing, the surrounding code may be changed or the statement may be removed. This imposes an issue for log processing tools like Sherlog, Log Enhancer and Salsa designed for log analysis. These tools heavily rely on log statements and changes may cause the tools to decrease in performance, which therefore requires updates. Kabinna et al. [2016] propose that identifying log statements that are likely to change in the future can help developers of log processing tools. Knowing which log statements are likely to change provides a way to prioritize on which log statements to base the analysis on. This can reduce the effort required to maintain logging tools. In their work the authors model whether logging statements will change using a random forest classifier. In combination with the commit history they find that developer experience, file ownership, log density and source lines of code (SLOC) have a significant influence on the likelihood that a log statement will change. Unfortunately, the authors do not verify whether this knowledge can actually benefit developers of log processing tools. Also the influence of changing log statements on the performance of these tools is only assumed.

Specifications that help developers log properly are not readily available. Zhu et al. [2015] propose a tool that helps developers make informed logging decisions. Based on the common logging practices in a system they extract, for that system, logging features (e.g. exception type) from the source code. Using these features they train a model using decision tree learning. As a result, for new blocks of code, the tool can propose to log in that place and can suggest a logging statement. In their evaluation the authors found that developers were able to make better logging decisions and overall spent less time on writing log statements.

In earlier days when log files were still small enough to process by hand it was important that log messages contained human readable text. Now that automatic analysis is becoming more and more important due to the increase in log volume it is not just about human readable text anymore. While manual analysis is still necessary for failure diagnosis, free text imposes restrictions on automatic processing. Salfner et al. [2004] propose several guidelines to ready logs for automatic analysis. For instance, splitting logs into an event type and the source of an event introduces a distinction between *what* has happened and *where* something happened. A hierarchical numbering scheme is a structural way to classify types of events based on categories. More specific errors are found further down this classification tree. For example, a data validation error can be written as *event.type=1.2.1*. This approach lends itself well for different levels of granularity. During failure diagnosis an operator can look into the full detail of the error type. In monitoring tasks the more generic type (e.g., *event.type=1.2*) may suffice. In addition, a hierarchical numbering scheme benefits clustering algorithms due to the added structure and its ordinal representation. An approach like this requires initial effort at design time as well as a commitment to update the error types as new types are implemented. Salfner et al. [2004] did not evaluate their proposal, however, taking the entire design of a system into account when introducing log statements intuitively seems more effective than the simplistic approach of writing single log statements when deemed valuable. For this reason this angle is interesting to investigate further.

Developers often do not write messages specially designed for operators and the rationale behind their logging decisions is not represented in the logs. This development knowledge however can be very valuable for understanding the log lines and to diagnose a failure. Examples of development knowledge: the surrounding source code of the corresponding log statement, source code comments, commit messages and issue reports. Shang et al. [2014] developed an approach which gathers this information on demand. They evaluated their approach qualitatively by examining log inquiries found in mailing lists and by web search, and found that their approach eases the resolution of the inquiries over 50% of the time. Furthermore, they were successfully able to use their approach to add missing information to randomly sampled log lines. Issue reports were found to be the most useful source of developer knowledge.

Knowing what to log and where to log does not guarantee effective log analysis. One has to consider logging from an application perspective. Log messages change and influence the performance of analysis. Log statements should not be an isolated part of an exception, but have meaning on a system level combined with other log statements. Lastly, log messages may not be enough for failure diagnosis, but adding the rationale behind the messages can improve this.

12. Discussion

We have made several observations while conducting this literature survey, which we discuss in this chapter. In addition we suggest possible research paths for future work. Section 12.1 describes our findings related to the applicability of techniques in different

scenarios and settings. The consequences of the ever increasing amount of log data for log analysis techniques in terms of feasibility are elaborated on in Section 12.2. In Section 12.3 we describe that techniques are rarely used in isolation, but rather are used in composition with other techniques. Finally, in Section 12.4 we treat commercial tools and their relation to log research.

12.1 Applicability

We defined event logs to refer to records of events taking place in a system. This remains a very general description and unfortunately in log research this terminology is not refined. For this reason it remains unclear to what sort of log messages one refers when speaking of event logs. They could be logs from, for instance, a distributed file system, a web-server, a game server, SaaS platform or a large multi-node supercomputer. All these applications serve a different goal and so their logs will be different. Also, the variety in the messages across systems is different. In a large multi-node system filtering may reduce the number of messages by 99% Zheng et al. [2009] , which is a promising and a useful result for these types of systems. However, filtering is likely not helpful in a web-server serving a large number of web pages and interacting with a webshop database, because of the variety in the logs. Some application logs may be information rich and may contain different variables and system states, while others may only contain an IP address and a constant part.

The applications are different, the variety in the messages is different, the goal is different and the number of log messages produced is different. In log research this essential information is not always given. First of all, this makes it difficult to see in which situation a proposed technique is applicable. Second, it makes it hard to make a proper comparison between different techniques. For instance, given these differences it is unclear when to use a specific parsing technique.

Future work may be directed at investigating whether the differences in event logs from different applications can be successfully characterized. This distinction can act as reference for work on log analysis to explain the applicability of the proposed techniques. In addition, several varying log datasets may be introduced to represent these different types of event logs. Having these datasets enables the side by side comparison of various techniques proposed in the field. These comparisons will give better insights into the advantages and disadvantages of certain techniques and methods and will make it easier for practitioners to see the applicability of these to their system.

12.2 Feasibility

In addition to the lack of clarity of the applicability of log research, its feasibility is also not always clear. The number of log messages that today's systems produce increases and so does the need for scalable solutions. Although simple techniques like heuristics are computationally cheap, the contrary is true for machine learning based clustering techniques for instance. While these techniques, despite their computational intensity, may perform well on small scale applications, in enterprise applications the number of

log messages can easily increase to millions a day. Most methods proposed are tested using datasets that contain at most a million log lines. Also, little research that we encountered takes this scalability aspect into consideration or reports on possible shortcomings on larger datasets. This lack of clarity makes it difficult for practitioners to determine whether a proposed technique is useful. In addition, comparing the effectiveness of multiple techniques is complicated by the lack of insights into their feasibility.

Besides the runtime complexity, the amount of manual work to set techniques up effectively and sustainably is not optimal. While some techniques, e.g., heuristic parsing, require a small amount of manual work up front, some clustering techniques do not. Most clustering techniques work with a user defined threshold, which has to be empirically determined for optimal results. Together with the runtime complexity of the clustering techniques the determination of this value becomes a time consuming task. Moreover, systems evolve and so do their log statements. To maintain the effectiveness of employed techniques one has to update the thresholds or heuristics, or re-apply the clustering technique. Little work considers the fact that systems are updated, which has an influence on log analysis.

Future work may focus on comparing existing techniques based on their effectiveness on a large scale. This may involve benchmarks on different sets of logs as well as comparing their runtime complexity. New research in the field that introduce techniques can benefit from adding a section on the runtime complexity. In addition, a performance estimate on a large log dataset can greatly clarify the feasibility of the proposed technique. Other future work may be directed at investigating the implications of updating a system in terms of the effectiveness of log analysis techniques. This will further contribute to insights into the feasibility of these techniques.

12.3 Technique Composition

The practice of logging, from writing log statements, through log abstraction, to log analysis, has not been clearly defined. There is no set of rules that determines which techniques to use and in what order to apply them to obtain the best results, nor is it described when parsing should be used and when it should not. Overall it seems that techniques can be used interchangeably. Also, multiple techniques can be concatenated to construct an optimal composition deemed effective. Again, there is no set of guidelines that helps to determine which techniques to use in which order. What we observed is that log parsing is an optional first step of which the result can be used in further log analysis. Furthermore, the more simple techniques like heuristics or filtering are often applied first, while machine learning steps often come last. This is however not universally true as simple heuristics can also be used to further refine the clusters constructed by an algorithm.

Future work may aim to propose a framework that can be used to compose multiple techniques together, ranging from simple log parsing techniques to advanced machine learning analysis techniques. Moreover, insights are needed in which situations lend themselves for combining techniques.

12.4 Commercial Tools

There are countless commercial tools available for log analysis. Among them the ELK Stack[6], Loggly[7], OverOps[8], Sentry[9] and Splunk[10]. Although these tools mostly focus on obtaining an overview, easy search, and providing visualizations, they also allow failure and anomaly detection. Every aspect of log analysis that we discussed in this survey, from log parsing to analysis, is also available in these commercial tools. What is interesting is that some of these tools are mentioned in research, but that no comparisons have been made between techniques proposed in research and their commercial counterpart. In other words, there is no evidence that commercial tools outperform what is proposed in research and vice versa.

Future work may target to compare commercial tools to research techniques in terms of precision and performance. Additionally, these tools may be evaluated on their usefulness in failure detection for instance. Although challenging due to the fact that most of these tools are closed source, the results will help to determine the current state of the art in log analysis. This knowledge can provide new insights into the shortcomings of present-day techniques and may lead to a basis for future research. Finally, a qualitative or quantitative study on the business goals of logging in industry can be conducted to outline what logs are used for.

12.5 Summary

Logs and log analysis are of vital importance in today's software systems. By means of analysis system failures and anomalies can be detected from event logs and the problem diagnosis process can be improved. However, the ever increasing number of log messages that present-day systems produce renders exclusive human analysis unfeasible. Log abstraction techniques are introduced to reduce the amount of data to consider in log analysis or detection. By obtaining log message templates using log parsing one can effectively group log messages to ease analysis. The techniques used range from source code parsing to simple heuristics to complex clustering algorithms. Furthermore, filtering is also used to reduce the amount of data to consider. The abstracted data can be used as input for failure detection, failure diagnosis or anomaly detection. Again, a wide range of techniques is available for this purpose ranging from simple hypothesis tests to machine learning approaches. Proper abstraction and detection techniques do not guarantee effectiveness. Where, what and how to log are of key importance to enable effective further analysis.

6https://www.elastic.co/

7https://www.loggly.com/

8https://www.overops.com/

9https://sentry.io/

10https://www.splunk.com/

The given overview of log research provides insights into the current state of the art and provides research angles for future work. We found the applicability of different techniques to different systems and types of logs to be nontransparent. To this end we recommend future work to be directed at removing this unclarity and introducing log datasets for research purposes. Furthermore, little insights are given into the runtime feasibility of the different techniques applied on a larger scale. With the number of logs produced by systems reaching millions a day, scalability is an essential angle to explore. Log analysis techniques are sporadically used in isolation. Often times a composition of different techniques is used and turned into a tool. There are no guidelines or best practices for this and it is not clear when to use one technique over another. Lastly, commercial log analysis tools are readily available. Log research, however, hardly considers these tools. For this reason the effectiveness of research techniques compared to commercial tools remains unclear.

In conclusion, the research field of log analysis is broad and nontransparent. The lack of clarity calls for structure in log research and interesting research angles to be explored.

Bibliography

Rakesh Agrawal, Tomasz Imielin´ski, and Arun Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD: International Conference on Management of Data*, 1993.

Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules.
In *International Conference on Very Large Data Bases*, 1994.

Michal Aharon, Gilad Barash, Ira Cohen, and Eli Mordechai. One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs. In

Conference on Machine Learning and Knowledge Discovery in Databases. Springer, 2009.

Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *ACM SIGSOFT: European Conference on Foundations of Software Engineering*, 2011.

Peter Bodic, Greg Friedman, Lukas Biewald, Helen Levine, George Candea, Kayur Patel, Gilman Tolle, Jonathan Hui, Armando Fox, Michael I Jordan, et al. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *IEEE International Conference on Autonomic Computing*, 2005.

RP Jagadeesh Chandra Bose and Wil MP van der Aalst. Discovering signature patterns from event logs. In *IEEE Symposium on Computational Intelligence and Data Mining*, 2013.

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

Marcello Cinque, Domenico Cotroneo, Roberto Natella, and Antonio Pecchia. Assessing and improving the effectiveness of logs for the analysis of software faults. In *IEEE International Conference on Dependable Systems & Networks*, 2010.

Marcello Cinque, Domenico Cotroneo, and Antonio Pecchia. Event logs for the analysis of software failures: A rule-based approach. *IEEE Transactions on Software Engineering*, 39(6), 2013.

Ilenia Fronza, Alberto Sillitti, Giancarlo Succi, Mikko Terho, and Jelena Vlasenko. Failure prediction based on log files using random indexing and support vector machines. *Journal of Systems and Software*, 86(1), 2013.

Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *IEEE International Conference on Data Mining*, 2009.

Xiaoyu Fu, Rui Ren, Jianfeng Zhan, Wei Zhou, Zhen Jia, and Gang Lu. Logmaster: mining event correlations in logs of large-scale cluster systems. In *IEEE Symposium on Reliable Distributed Systems*, 2012.

Ana Gainaru, Franck Cappello, Stefan Trausan-Matu, and Bill Kramer. Event log mining tool for large scale hpc systems. In *European Conference on Parallel Processing*. Springer, 2011.

Ceki Gu¨lcu¨ and Scott Stark. *The complete log4j manual*. QOS.ch, 2003.

Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. An evaluation study on log parsing and its use in log mining. In *IEEE International Conference on Dependable Systems and Networks*, 2016.

Liang Huang, Xiaodi Ke, Kenny Wong, and Serge Mankovskii. Symptom-based problem determination using log data abstraction. In *Conference of the Center for Advanced Studies on Collaborative Research*. IBM Corp., 2010.

Bernard J Jansen. Search log analysis: What it is, what's been done, how to do it. *Library & information science research*, 28(3), 2006.

Zhen Ming Jiang, Ahmed E Hassan, Parminder Flora, and Gilbert Hamann. Abstracting execution logs to execution events for enterprise applications (short paper). In *IEEE International Conference on Quality Software*, 2008a.

Zhen Ming Jiang, Ahmed E Hassan, Gilbert Hamann, and Parminder Flora. An automated approach for abstracting execution logs to execution events. *Journal of Software*
Maintenance and Evolution: Research and Practice, 20(4), 2008b.

Suhas Kabinna, Weiyi Shang, Cor-Paul Bezemer, and Ahmed E Hassan. Examining the stability of logging statements. In *IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2016.

Kamal Kc and Xiaohui Gu. Elt: Efficient log-based troubleshooting system for cloud computing infrastructures. In *IEEE Symposium on Reliable Distributed Systems*,

2011.

B. Kitchenham and S Charters. Guidelines for performing systematic literature reviews in software engineering, 2007.

Barbara Kitchenham. Procedures for performing systematic reviews. *Keele University, UK*, 2004.

Lynn Kysh. Difference between a systematic review and a literature review. https://figshare.com/articles/Difference_between_a_systematic_review_and_a_literature_review/766364/1, 2013. Accessed: 2016-12-22.

Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, volume 10, 1966.

Wenmin Li, Jiawei Han, and Jian Pei. Cmar: Accurate and efficient classification based on multiple class-association rules. In *IEEE International Conference on Data Mining*, 2001.

Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. Log clustering based problem identification for online service systems. In *ACM International Conference on Software Engineering Companion*, 2016.

Haibin Liu and Vlado Kesˇelj. Combined mining of web server logs and web contents for classifying user navigation patterns and predicting users future requests. *Data & Knowledge Engineering*, 61(2), 2007.

Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. Mining invariants from console logs for system problem detection. In *USENIX Annual Technical Conference*, 2010.

Bing Liu Wynne Hsu Yiming Ma. Integrating classification and association rule mining. In
International Conference on Knowledge Discovery and Data Mining, 1998.

FP Machado. Cpar: Classification based on predictive association rules.

2003.

James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, 1967.

Oded Maimon and Lior Rokach. *Data mining and knowledge discovery handbook*, volume 2. Springer, 2005.

Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. Clustering event logs using iterative partitioning. In *ACM SIGKDD: International Conference on Knowledge Discovery and Data Mining*, 2009.

Leonardo Mariani and Fabrizio Pastore. Automated identification of failure causes in system logs. In *IEEE International Symposium on Software Reliability Engineering,* 2008.

Sergio Moreta and Alexandru Telea. Multiscale visualization of dynamic software logs. In *Joint Eurographics/IEEE VGTC Conference on Visualization*, 2007.

Karl Pearson. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302), 1900.

Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. Industry practices and event logging: Assessment of a critical software development process. In *IEEE International Conference on Software Engineering*, 2015.

John Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1), 1986. John Ross Quinlan. *C4. 5: programs for machine learning*. 1993.

Jane Radatz, Anne Geraci, and Freny Katki. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, 1990.

Thomas Reidemeister, Miao Jiang, and Paul AS Ward. Mining unstructured log files for recurrent fault diagnosis. In *IEEE International Symposium on Integrated Network*
Management and Workshops, 2011.

Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Upper Saddle River, 2003.

Ramendra K Sahoo, Adam J Oliner, Irina Rish, Manish Gupta, Jose´ E Moreira, Sheng Ma, Ricardo Vilalta, and Anand Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *ACM SIGKDD: International Conference on Knowledge Discovery and Data Mining*, 2003.Felix Salfner and Steffen Tschirpke. Error log processing for accurate failure prediction. In *USENIX Conference on Analysis of system logs*, 2008.

Felix Salfner, Steffen Tschirpke, and Miroslaw Malek. Comprehensive logfiles for autonomic systems. In *IEEE International Parallel and Distributed Processing Symposium*, 2004.

Santonu Sarkar, Rajeshwari Ganesan, Marcello Cinque, Flavio Frattini, Stefano Russo, and Agostino Savignano. Mining invariants from saas application logs (practical experience report). In *IEEE Dependable Computing Conference*, 2014.

Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, 2(2), 1999.

Weiyi Shang, Meiyappan Nagappan, Ahmed E Hassan, and Zhen Ming Jiang. Understanding log lines using development knowledge. 2014.

Tetsuji Takada and Hideki Koike. Tudumi: Information visualization system for monitoring and auditing computer logs. In *IEEE International Conference on Information Visualisation*, 2002.

Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Salsa: Analyzing logs as state machines. *USENIX Conference on Analysis of System Logs*, 8, 2008.

Microsoft TechNet. Transaction log management. https://technet.microsoft.com/ en-us/library/ms345583(v=sql.105).aspx, 2016. Accessed: 2016-12-21.

Risto Vaarandi et al. A data clustering algorithm for mining patterns from event logs. In *IEEE Workshop on IP Operations and Management*, 2003.

Vladimir Vapnik. *The nature of statistical learning theory*. Springer Science & Business Media, 2013.

Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *ACM International Conference on Evaluation and Assessment in Software Engineering*, 2014.

Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In *ACM SIGOPS: Symposium on Operating systems principles*, 2009.

Wei Xu, Ling Huang, and Michael I Jordan. Experience mining google's production console logs. In *USENIX Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*, 2010.

Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *ACM SIGARCH: Computer Architecture News*, volume 38, 2010.

Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: enhancing failure diagnosis with proactive logging. In *USENIX Symposium on Operating Systems Design and Implementation*, 2012a.

Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *IEEE International Conference on Software Engineering*, 2012b.

Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems*, 30(1), 2012c.

Ziming Zheng, Zhiling Lan, Byung H Park, and Al Geist. System log pre-processing to improve failure prediction. In *IEEE International Conference on Dependable Systems & Networks*, 2009.