STAMP

Deliverable D3.1

Survey on logging practices and tools

Project Number	:	731529
Project Title	:	STAMP
Deliverable Type	:	Report

Deliverable Number	:	D3.1
Title of Deliverable	:	Survey on logging practices and tools
Dissemination Level	:	Public
Version	:	2.20
Latest version	:	https://github.com/STAMP-project/docs-forum/blob/master/docs/d31_survey_on_logging_practices_and_tools.pdf
Contractual Delivery Date	:	M6
Contributing WPs	:	WP 3
Editor(s)	:	Xavier Devroey, TU Delft
Author(s)	:	Mauricio Aniche, TU Delft Joop Aue, TU Delft Jeanderson Candido, TU Delft Xavier Devroey, TU Delft Arie van Deursen, TU Delft Daan Schipper, TU Delft Andy Zaidman, TU Delft
Reviewer(s)	:	Benoit Baudry, KTH Lars thomas Boye, TellU Caroline Landry, INRIA

Abstract

Using logs to detect and diagnose problems in software systems is no longer a possible 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, which makes it difficult for practitioners to identify what is relevant to them, and for researchers to determine relevant angles to explore. The goal of task 3.1 is to establish how developers use log data collected from running software for debugging or resilience purposes. To this end, we present a literature survey on the state-of-the-art of log analysis and insights of the state-of-the-practice of log file usage from interviews with the STAMP partners.

Revision History

Version	Type of Change	Author(s)
1.0	First version	Joop Aue, TU Delft Mauricio Aniche, TU Delft Arie van Deursen, TU Delft Andy Zaidman, TU Delft
1.10	Final draft Mauricio Aniche, TU Delft	Joop Aue, TU Delft Arie van Deursen, TU Delft Andy Zaidman, TU Delft
2.0	Revised draft	Mauricio Aniche, TU Delft Joop Aue, TU Delft Jeanderson Candido, TU Delft Xavier Devroey, TU Delft Daan Schipper, TU Delft Arie van Deursen, TU Delft Andy Zaidman, TU Delft
2.10	Revised release after review	Mauricio Aniche, TU Delft Joop Aue, TU Delft Jeanderson Candido, TU Delft Xavier Devroey, TU Delft Daan Schipper, TU Delft Arie van Deursen, TU Delft Andy Zaidman, TU Delft
2.20	Revised release after additional review	Mauricio Aniche, TU Delft Joop Aue, TU Delft Jeanderson Candido, TU Delft Xavier Devroey, TU Delft Daan Schipper, TU Delft Arie van Deursen, TU Delft Andy Zaidman, TU Delft

Contents

Acronyms

EC	European Commission
WP	Work Package
TUD	TU Delft
AEon	Activeeon
Eng	Engineering

Glossary

This glossary presents the terminology used across the different deliverable of work package 3.

Botsing: Meaning *crash* in Dutch, Botsing is a complete re-implementation and extension of the crash replication tool EvoCrash. Whereas EvoCrash was a full clone of EvoSuite (making it hard to update EvoCrash as EvoSuite evolves), Botsing relies on EvoSuite as a (maven) dependency only and provides a framework for various tasks for crash reproduction and, more generally, test case generation. Furthermore, it comes with an extensive test suite, making it easier to extend. The license adopted is Apache, in order to facilitate adoption in industry and academia. Botsing is the name of the framework for online test amplification developed in WP3.

Code instrumentation: Code instrumentation in Botsing consists in the injection of probes into the bytecode of a Java application to monitor and log the runtime behavior of specific classes.

JCrashPack: JCrashPack is a benchmark containing 200 crashes to assess crash replication tools capabilities.

Model seeding: in search-based software testing, seeding consist in providing external information to the search algorithm to help the exploration of the search space. Model seeding, developed within STAMP, is the seeding of transition systems (a formalism similar to state machines describing a dynamic behavior) to a search based test case generation algorithm. The models represent the common usages of classes and allow the generate objects with sequences of method calls, representing a common behavior in the application. In Botsing, models are learned from the source code (static analysis) and from the logs produced by the execution of the system (dynamic analysis using instrumentation) using n-gram inference.

Stack trace: a (crash) stack trace is a piece of log data usually denoting a crash failure. Stack traces provides information on the exception thrown and on the propagation of that exception trough the stack of method calls.

Stack trace preprocessing: stack traces can contain redundant and useless information preventing to automatically reproduce the associated crash. The preprocessing allows to filter stack traces to keep only relevant information.

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 [12, 72].

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 [70]. It has been stated that human interpretation is no longer feasible at this scale and automated approaches have become a necessity [35, 44]. 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 introduces 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, together with the results of interviews of the industrial members of STAMP to identify their current state-of-the-practice.

Chapter 1 - Literature Review on Logging presents the state-of-the-art of logging.[1]

Chapter 2 - State-of-the-practice on Logging presents the state-of-the-practice of logging of the STAMP partners.

[1]The work described in Chapter 1 is based on two literature surveys done by MSc students at TU Delft. The first literature survey was carried out by Joop Aue; this work had started before the official start of the STAMP project and was concluded at the end of 2016, coinciding with the start of the STAMP project. See also: https://repository.tudelft.nl/islandora/object/uuid%3A90afad8d-7010-4407-a502-fb5d73c0f291?collection=education The second literature study was carried out by Daan Schipper.

Chapter 1

Literature Review on Logging

In this literature review, we 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.

1.1 Methodology

Inspired by the characteristics of mapping studies and SLRs we followed a more lightweight research methodology for our literature survey. In this section, 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.

Our 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.

1.1.1 Search and Understand

The first iteration started of with an initial set of 21 papers. This initial set was devised by the authors of this paper through manual search. 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. It is also important to notice that we only considered full papers (i.e., papers with more than 6 pages).

A second iteration was performed to capture recent work in the field. 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 2010 and 2016 for each of these. 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.

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, researchers in our network pointed us to another 12 log research papers.

To account for literature that we may have 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.

We also noticed that papers on "logging visualization" deserved its own search. To that aim, we repeated the search on Google Scholar with the terms:

- software logging analysis,
- (software OR system OR execution) AND (logs OR logging OR log OR traces), and
- visualization AND (software OR system OR event) AND (logs OR logging).

After applying snowballing, we ended up adding 26 new papers.

1.1.2 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 handsearches 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 is also the bias of one assessor, which can be reduced by having multiple people assess the papers in parallel.

Lastly, due to time constraints, in this version, we are able to cover 62 of the selected papers. This may be a threat to the completeness of this survey since the uncovered papers may introduce techniques that will not be part of our discussions. Future work will report on these papers to complete our overview.

1.2 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 [38, 70]. 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 1.2.1. In Section 1.2.5 other abstraction methods are discussed. Finally, in Section 1.2.6 we elaborate on whether abstraction can benefit the quality of log analysis.

1.2.1 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 [29].

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 [69, 70, 38]. 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 [34]. 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: heuristic methods, clustering methods, source code based methods and other methods. The groups are discussed in the following sections.

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 [59] 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. [27] 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. [25] 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. [33] use a similar technique where they reason that the equals 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. However, 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 computational expensive parsing method or can be used in combination with a method such as Jiang et al. [33] demonstrate.

He et al. [29] empirically evaluated the usefulness of preprocessing the logs using heuristics. They found that using domain knowledge to remove, for example, IP addresses, improves log parsing accuracy.

1.2.2 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. [66]. The tool, Simple Logfile Clustering Tool (SLCT), is based on the Apriori algorithm designed for frequent item set mining [3]. 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 some 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. [44] 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 (i) individual word count, (ii) token position with the least number of unique values, and (iii) bijective relationships between tokens. The resulting clusters represent similar message patterns to those proposed by Vaarandi et al. [66]. 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. [27], 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 1.2.1. 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. [33]. 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. [33] evaluated their approach on four enterprise applications and compared it to the association rule learning technique by Vaarandi et al. [66]. 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. [29] 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.

1.2.3 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 [69]. 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 [69] addresses 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. [69] apply their log parsing technique to C/C++ programs and evaluate their parser to be successful by being able to match 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. [72] 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.

1.2.4 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 [36], 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 [59] 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. [25] 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 frequent 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. [32] 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, not in all circumstances source code is available and other approaches are therefore necessary.

1.2.5 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. [58] 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. [26] and Zheng et al. [77], who, based on a threshold value, use the interval between the similar events to determine which logs to filter out. Additionally, Zheng et al. [77] 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 [77].

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.

1.2.6 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 [40] log analysis without log abstraction as a preprocessing step is not even feasible for larger log datasets [31]. 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. [31] 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 [37] and Classification based on Predictive Association Rules [41], were found to outperform Naive Bayes and C4.5 [53], 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.

1.3 Log Analysis: Detection and Diagnosis

In this section, 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 1.3.1 and refer to them in the subsequent sections. In Section 1.3.2 we compare the failure detection techniques. Section 1.3.3 describes techniques used in anomaly detection and finally we elaborate on failure diagnosis in Section 1.3.4.

1.3.1 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.

Associative rule learning

Associative rule learning is a learning technique that discovers strong relations among variables [2]. The key idea behind association rules is that if a certain set of events has occurred that 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 [2] 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 [39]. For instance, three events that occur within a close time interval may indicate then 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.

Chi-squared test

The chi-squared test [50], 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.

Decision tree learning

Decision tree learning [52] 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 leave node.

In log analysis decision trees are used to detect recurrent failures [11, 56]. 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.

Hierarchical clustering

Hierarchical clustering [43] is an unsupervised learning technique based on the idea that objects near each other are more related then 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 [42]. There are two strategies when it comes to hierarchical clustering: agglomerative clustering and divisive clustering [43]. 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 1.2.2, but has an application in log analysis as well [38, 35]. 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.

Naive Bayes

Naive Bayes [57], a family of simple probabilistic classifiers, is used to classify based on features that are considered independent of each other. The applications of this 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 [9]. 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.

Support vector machine

A Support Vector Machine (SVM) [67], 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 [11, 24].

1.3.2 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 [55]. 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. [56] 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. [38] 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 [11] 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 1.3.1. 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. [24] use SVMs to classify operation sequences from session log files. They used a weighted SVM to deal with the problem of skewed data. Bose and van der Aalst [11] 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 gives 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 or not equally represented, although the weighted variant of SVMs attempts to deal with this shortcoming.

1.3.3 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 [35] 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 [35] 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. [38] 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 [35], 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. [9] 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. [9] 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 webserver 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. [39] 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.

1.3.4 Failure Diagnosis

Failure diagnosis is essential to ensure failures do not occur again in the future. Knowing that 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. [72] 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. [12] 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. [56], Chen et al. [12] 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 equaly 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.

1.4 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 [74]. 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 1.4.1. Second, the problem of *where to log* is elaborated on in Section 1.4.2. Finally, Section 1.4.3 describes *how to log*.

1.4.1 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. [75] 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 [72] can benefit from this work, however, they did not verify this.

After introducing their log diagnosis tool, Yuan et al. [72] 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. [38] and He at al. [29] 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. [72] propose a similar idea where they would like to log the file name 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.

1.4.2 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. [73] 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 factor 2. To improve diagnosis the authors present Errlog, which identifies potentially not logged 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% 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. [73] 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. [14] applied fault injection to investigate log coverage. Enabling exactly one fault at a time and accessing 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. [15] 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. [51] 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 [15].

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 is able 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.

1.4.3 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 [34]. 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. [34] 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. [78] 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. [60] 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. [60] 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. [61] developed an approach which gathers this information on demand. They evaluated their approach qualitatively by examining log inquiries found in mailings list 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.

1.5 Log Visualization

1.5.1 System Behavior

Log analysis can help optimise or debug system behaviour. Today's infrastructure is based on cloud systems, which are required to meet certain performance and dependency goals. Managing and analysing the distributed logs produced by nodes of the system is required to understand the system's behaviour, which is often related to understanding how the resources in that system are used [48]. A visualisation can provide an additional edge for a developer in failure-diagnosis and other uses.

1.5.2 Resources

An approach to gain insight into the system behaviour is to follow the interactions of resources. A clear overview as how the resources are affected and when nodes are communicating with each other can provide the clue of a bug.

The system's structural decomposition and the nature of interactions during a trace can be shown with a circular view, as presented by Cornelissen et al. [18]. A massive sequence view of the execution traces is given next to the circular view to give a navigable overview of the consecutive calls in chronological order. For each trace a detailed visualisation of the system's structural entities and their interrelationships can be viewed, which can be seen in Figure 1.1.

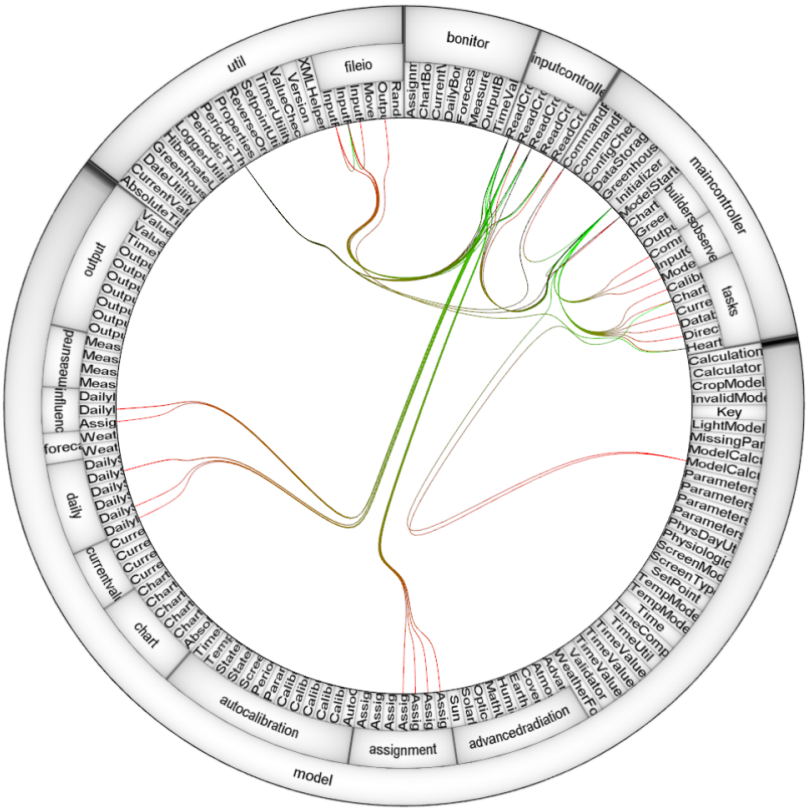

Figure 1.1: Circular view of the nature of interactions of a single trace. Extracted from [18]

Artemis is a modular application designed for analysing and troubleshooting the performance of

large clusters running datacenter services, presented by Crectu et al. [19]. The log data is collected, filtered and summarised before storing it in a database. A vertical (columns) and horizontal (lines) tagging allows to visualise the data with histograms and distributions, and to identify if misbehaved instances (lines) are also outliers with respect to different metrics.

Oliner et al. [49] deduce the influences among components in a system by looking for pairs of components with time-correlated anomalous behaviour. The term influence infers to the relationship between components or groups of components by looking for surprising behaviour that is correlated in time, these components share an influence. Nodes execute simultaneously in a distributed setting and during the execution they generate asynchronous logs that are hard to reason about. By synthesising the log events, the order of execution can become clear to the developers, allowing them to more easily debug the program's execution. The extent to which a component is behaving anomalously is defined as the distance from the mean of some appropriate model at a certain time, for instance the deviation from average log message rate. Then the divergence of how much these values differ from the previous shown expected behaviour is calculated for each component and stored in a matrix, for which correlation coefficients can be calculated. If such a correlation exceeds a certain threshold, it means the components share influence. All the influences of the components are then captured in a Structure-of-Influence Graph, a graph with one vertex per component and edges that represent influences.

Rabkin et al. [54] track identifiers of entities present in log statements and visualises how much and by which components these identifiers are used. The so-called identity graph captures the influence of the components, since the components using the same entities are identified from the visualisation. This allows developers to spot dependencies or lack thereof. The authors identify the static and dynamic parts of a log statement. Every dynamic message of the same type has the same set of variable fields, which results in a fixed relationship between message type and the set of identifier classes referenced by messages of that type. These relationships are then visualised in the identity graph. An example of this visualisation can be seen in Figure 1.3.

```
1:  JobTracker: Adding task 'attempt_200911091331_0010_m_000002_0' to
 tip task_200911091331_0010_m_000002, for tracker tracker_r25
2:  JobInProgress: Choosing data-local task task_200911091331_0010_m_000002
3:  JobInProgress: Task 'attempt_200911091331_0010_m_000002_0' has
 completed task_200911091331_0010_m_000002 successfully.
```

Figure 1.2: Sample of a log. Extracted from [54]

Abrahamson et al. [1] create structure in the logs by adding partial ordering with the happened-before relationship. This ordering is then utilised to create an overview of the system execution. A space-time diagram of a single system execution is created, which relates events on different hosts based on the partial ordering information, a process that is normally hard to understand.

Zhao et al. [76] identify request flows of distributed systems, which are useful for profiling and understanding system behaviour. This is achieved by performing static analysis of the system's byte-code. The log printing statements are extracted in order to parse each output message, identifying the variable values. Furthermore the data-flow of these variables can give identifiers which remain unchanged during a request. Finally the temporal relationships are identified. For the request flows the latency over time, counts and trends over time and the average latency per node are automatically visualised.

Fittkau et al. [21] present a different kind of trace visualisation for large software landscapes, which builds on their previous work [23, 22]. It combines a landscape and a system level perspective by introducing three hierarchical abstractions: (1) systems which consist of one or more server nodes, (2) node groups which cluster nodes that are running the same application configuration, and (3) the amount of communication between the application. The resulting visualisations in landscape and

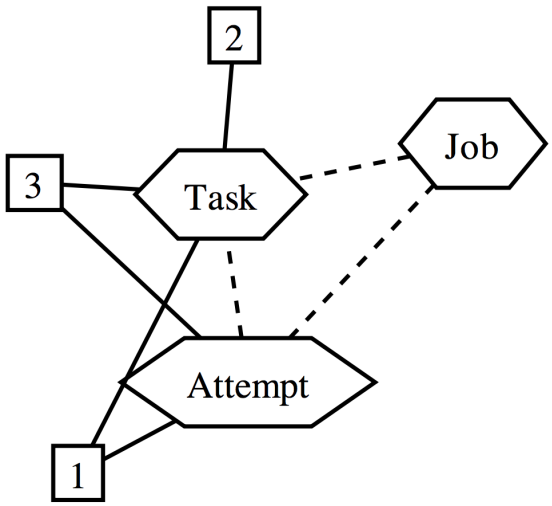

Figure 1.3: Identify graph of the log in Figure 1.2. Extracted from [54]

application level can be seen in Figures 1.4 and 1.5 respectively. The landscape can even be explored with virtual reality [20].

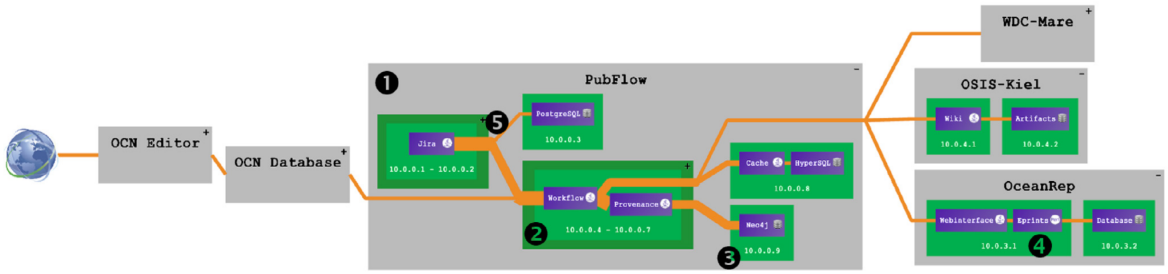

Figure 1.4: Landscape-level visualisation of ExplorerViz. Extracted from [21]

1.5.3 Model inference

The goal of a model-inference algorithm is to produce a model, typically a Finite State Machine (FSM), that accurately and concisely represents the system that produced the log [5]. Such models have been used to improve developers' understanding of system implementations and to find, diagnose, and remove bugs [7]. This section highlights the different visualisation techniques that have exactly that purpose.

One of the first log analysis techniques in the form of a FSM is presented by Tan et al. [64]. The first view is the control flow on each node, which can reveal hotspots of communication, highlighting particular key nodes, and captures the equity of distribution of tasks. The data flow view summarises the bulk transfers of data between each pair of nodes, which would reveal any imbalances of data accesses and shows the equity of distribution of workload. The final view correlates the state occurrences casually, which provides a time based view of the execution on each node, and also shows the

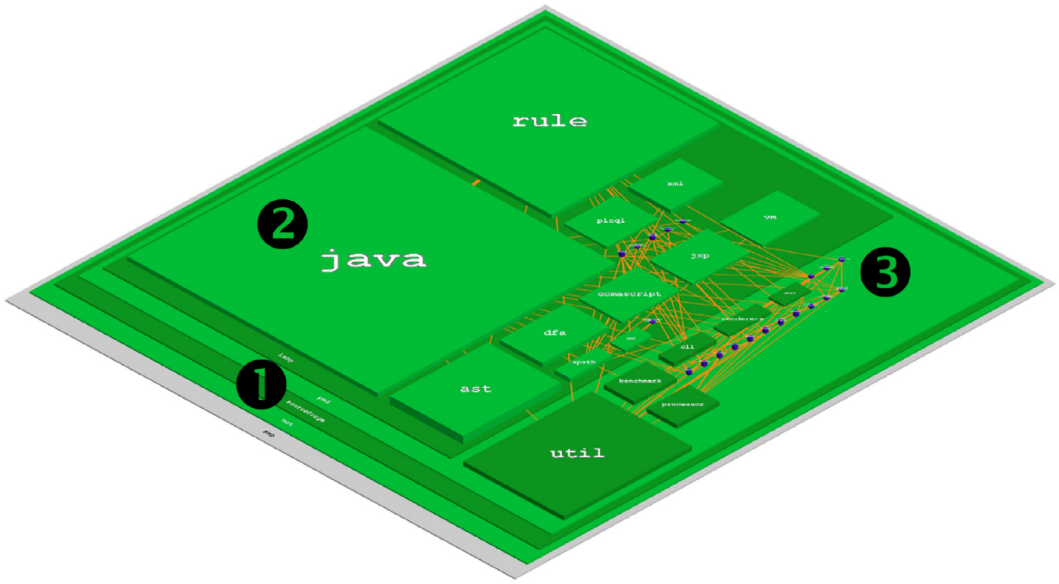

Figure 1.5: Application-level visualisation of ExplorerViz. Extracted from [21]

control-flow dependencies amongst nodes.

Beschastnikh et al. [7] present Synoptic, a tool that helps developers by inferring a concise and accurate system model, as seen in Figure 1.6. First event instances are extracted, which consist of a triplet containing a trace identifier, a timestamp and an event type are extracted with a set of user-defined regular expressions. This does not restrict developers to a particular log format. All event instances of the same trace are combined in a trace-graph. Next a model is formed of all trace-graphs, guided by the mined temporal invariants: a always followed by b, a never followed by b and a always precedes b.

The work in Synoptic is used as foundation and extended by CSight [6] and Perfume [47]. The focus of CSight is to infer a model of the system's behaviour in the form of a Communicating Finite State Machine (CFSM), whereas Perfume differentiates behaviourally similar executions that differ in resource consumption.

CSight follows the same stages as Synoptic: first mine temporal properties, construct a model and then refine the model which satisfies all invariants mined in the first stage. Instead the model is now in the form of a CFSM, which models multiple processes, or threads of execution, each of which is described by a FSM. CSight mirrors Synoptic's inference procedure but uses a different modelling formalism and algorithms, and works for concurrent systems that log concurrency as a partial order.

Perfume builds on Synoptic, but extends its property mining, model checking, and refinement algorithms to account for and enforce the resource-constrained temporal properties, to take into account the resource usage information in system logs, such as timing, CPU and memory utilisation, and other resource measures. Perfume infers models by generalising observed executions by enforcing a rich set of temporal, resource-constrained properties that hold in the observed executions. These models are then refined to eliminate counter-example generalisation that violate the final model, and augment with the aid of the kTails algorithm [8] to account for suboptimal refinement. Finally a model is presented that reveals what types of executions imply a problem.

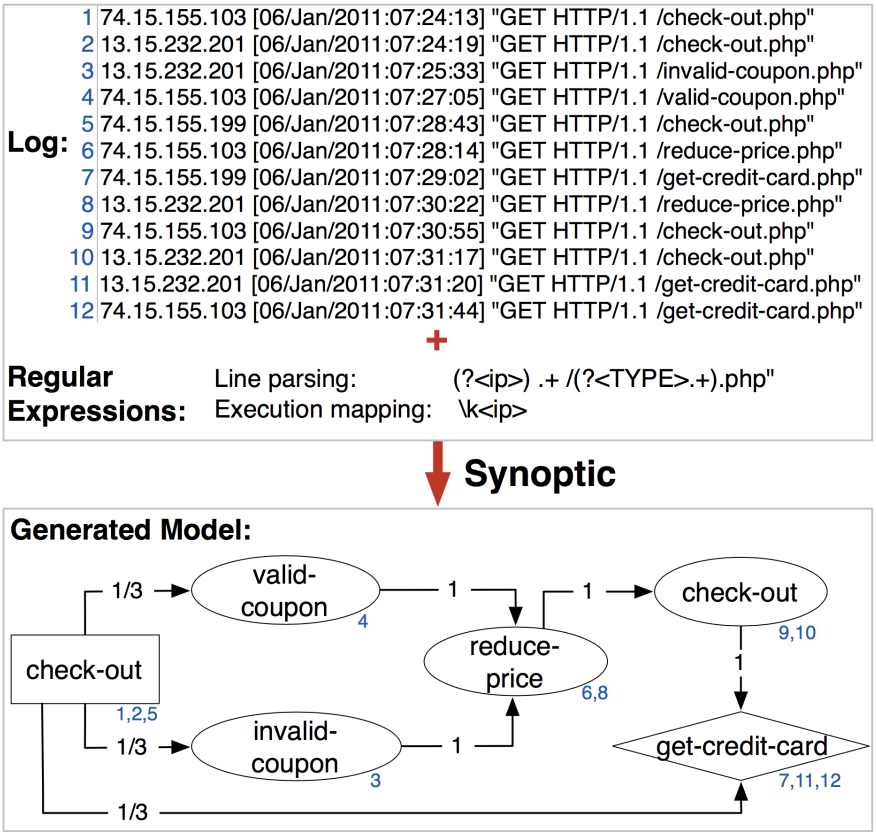

Figure 1.6: Model inferred by Synoptic, with edge labels indicating transition probabilities and subscripts to the right of each node lists the log line numbers. Extracted from [7]

1.6 Anomaly Visualization

Today's systems produce unprecedented amounts of logs, causing monitoring of logs to become increasingly challenging. Log data can be used to detect if the behaviour of the system is still normal, or anomalies are starting to surface. Either one can observe when the current behaviour deviates from the already known behaviour captured in reference models, or examine if instances of familiar problems captured as signatures start to appear. This section discusses the articles that perform anomaly detection based on logs, and then visualise them in an interesting way.

1.6.1 Reference Models

One can compare the current state of the system to the known and normal state to predict anomaly behaviour of a system. The normal system behaviour is then represented by reference models. The divergence of incoming logs are analysed against the reference models to find the origin of the anomaly. Many papers exist on this topic [65, 13], but this section focusses on the papers which include some type of visualisation.

Tak et al. [63] introduce LOGAN, which is able to identify the root cause of the anomaly by comparing the current behaviour against reference models. The references are identified by the log comparison algorithm, which consists of correlating relevant logs, parsing the logs to templates and aligning the logs to a known set of logs. The correlating of relevant logs is achieved by comparing

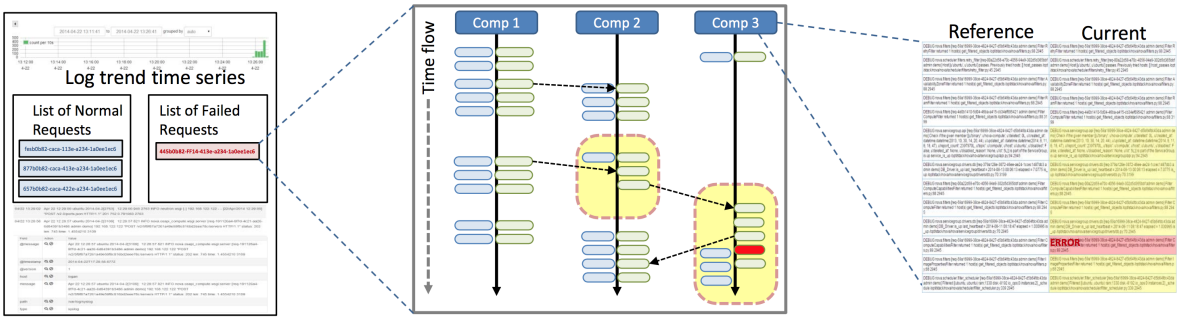

Figure 1.7: Presenting of anomaly behaviour by LOGAN. Extracted from [63]

identifiers present in log messages. The logs with the same identifiers are aggregated into a set. Within a set other identifiers and value pairs might be present, all the logs containing those identifiers are also added to the current set only if they do not belong to a larger scope. Next, the log templates are extracted by combining source-code parsing and clustering techniques. Finally, the sets are aligned with each other, identifying the unique and common logs which results in a reference model. The results are showed in a Kibana[1]-based dashboard, displaying the trend of log volumes, list of requests issued by the users, request types and list of log entries collected in real time. The difference between the reference models and current behaviour are highlighted and can be easily spotted, as seen in Figure 1.7. The exact details of this anomaly can then be further inspected by reviewing the difference between the normal and the current anomalous behaviour.

Another approach by Mariani and Pastori[45] is to identify failure causes by deriving Finite State Machines (FSAs) of expected behaviour from logs recorded during legal executions. The FSA are obtained by the previously developed kBehaviour inference engine [46], which is applied to the log files with rewritten attributes. Finally, the logs of failing executions are compared to these models and the likely illegal events are then presented to the user.

Yu et al. [71] present a similar technique called Cloudseer to build an automaton for task workflows generated by multiple correct executions. Next, during the execution of the system, the stream of log messages are matched against the set of automata to find individual log sequences out of the interleaved stream. The tool then searches for divergences, either the log sequence contains an error message or the expected log messages do not appear within a time interval. The latter is to account for silent failures. The necessary context information and the log sequence are presented to the user to aid further diagnosis.

1.6.2 Signatures

The other way around of modelling logs, looking for known anomaly behaviour in system behaviour, can be achieved by using signatures. Signatures are sequences of undesired system behaviour, for instance performance issues. The signatures are formulated based on specific metrics that relate to the performance problem, such as CPU and memory utilisation. Recurring faulty behaviour of a system is then detected by the the similarity to the signatures. Operators can more easily identify already known problems and measure the occurrences of the problem appearing, or know when a new complication has emerged.

A method to extract the essential characteristics from a system state into signatures is presented by Cohen et al. [17]. The capturing of the signatures is not evident, as the naive approach of using only the values of the measurements leads to poor clustering and recognition of anomalies. Therefore the authors use their previous work [16] to capture relationships between low-level system metrics and

[1] https://www.elastic.co/products/kibana

high-level behaviours based on a Bayesian network classifier and a heuristic search over the space of features.

Bodik et al. [10] also use this methodology to create signatures in their tool HighLighter, but based on logistic regression with L1 regularisation. The added benefit of using logistic regression is it still performs well when the number of features are significantly larger than the number of captured faulty executions. Moreover, this approach is more suitable to scale due to the very efficient implementations available. The collected features are enhanced with the gradient and standard deviation with respect to time periods in the past and normalised to zero mean and variance of one. Finally the signatures are created for each machine at each time interval with the metrics and label them according to their contribution of abnormal behaviour. An anomaly can be captured by the different signatures, as shown in Figure 1.8

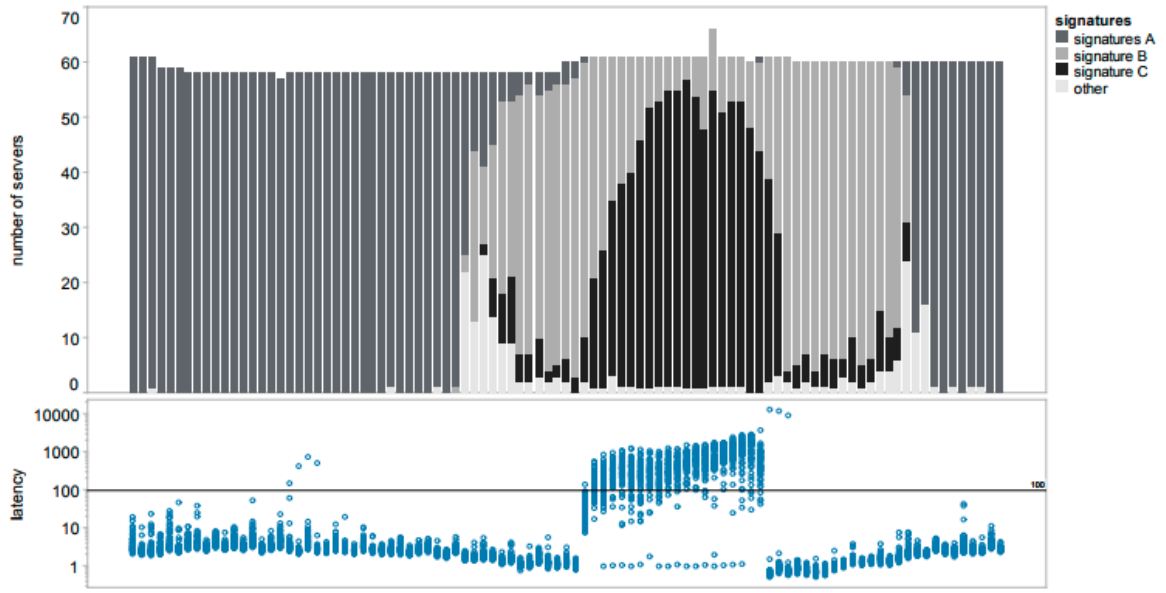

Figure 1.8: The captured signatures of system behaviour by HighLighter, with the blue dots representing latency during a single interval. Extracted from [10]

1.7 User Interaction

Logs can also be used to profile the behaviour of how a system is utilised by the end users. Applications are designed for a specific user demographic. However, the user demographic might evolve and require new interactions. Logs characterise visitors, which can provide insight into customer demographics. This chapter focusses on techniques that capture this evolvement of the user by investigating the navigation patterns of the users.

1.7.1 User behaviour

Ghezzi et al. [28] introduce the framework BEAR, a combination of an inference engine to cluster users based on their behaviour and an analysis engine that uses designer specified properties to infer insights about the users' behaviours and the relation among users. The BEAR inference engine uses log files and user defined propositions to infer Discrete Time Markov Chains [4] of the behaviour. The

user declares which URLs correspond to a state in the model, so the engine can associate the incoming logs to a state. Next, when a log line is associated with a certain state, the amount of times this node is visited and the transition of the previous state to this state is increased. The engine determines the previous state according to the IP address, if the IP address is already encountered in a previous state in the past time interval then that state is considered as the source state. Based on these statistics, a model can be inferred which shows the users behaviour.

Keeping track of each users' individual mouse clicks can lead to massive amounts of incomprehensible data, for example at eBay one-day web session contains over 900 million sessions and is about 6 terabytes [62]. The sheer size makes it increasingly challenging to visualise. The solution at eBay is TrailExplorer [62], where knowledge can be discovered from the users' browsing behaviours by tracking and recording the individual mouse clicks. To cope with the massive amounts of web session data they utilise the idea of MapReduce. First the developers are asked to select a set of events of their interests. The system splits the data and retrieves all the sessions which contain any of these events. Next the retrieved data is visualised in the visual interface consisting of four components, as seen in Figure 1.9. The main view shows the user sessions, of which the details can be viewed next to it. The distribution of events are shown below, and finally a legend is presented to show the colours of the events.

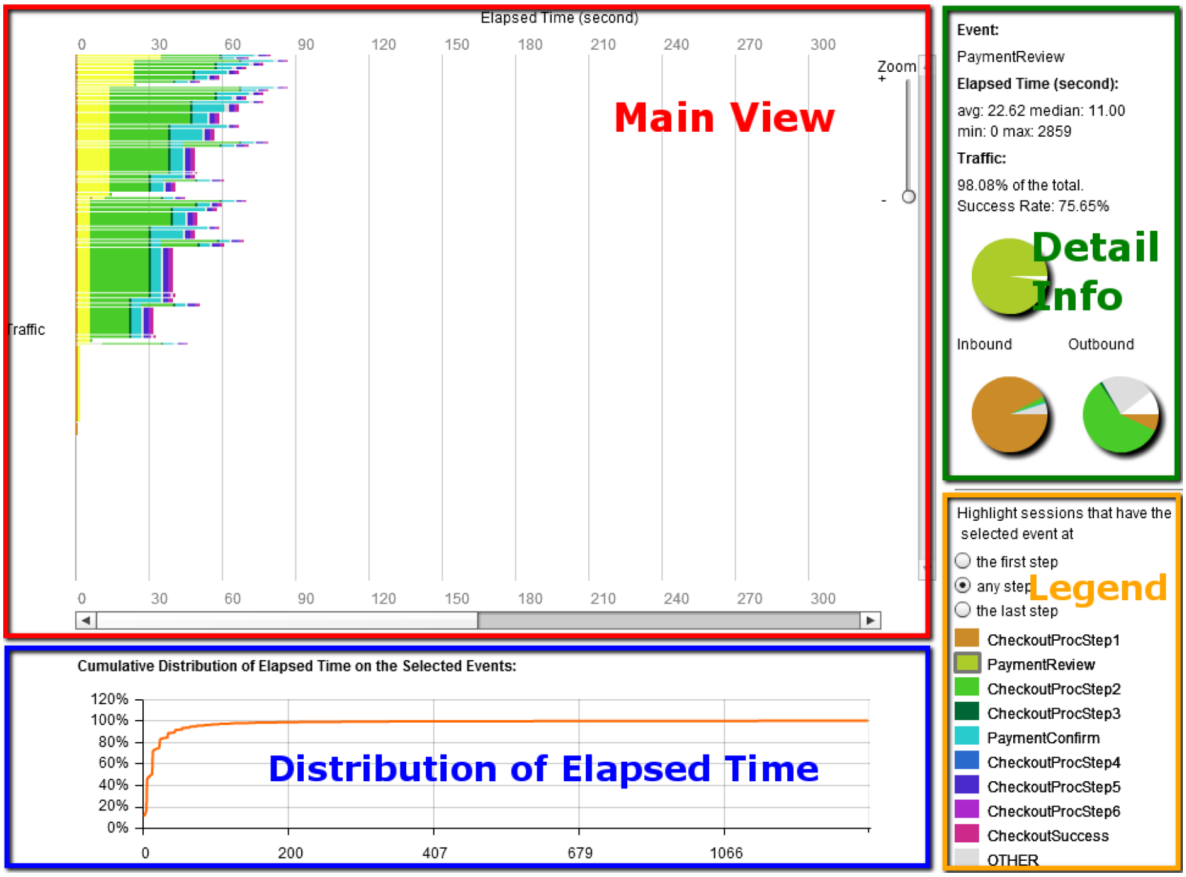

Figure 1.9: TrailExplorers user interface. Extracted from [62]

The visualisation of the large scale logging at Twitter is detailed by Wongsuphasawat and Lin [68]. Multiple views of the client events collection can be rendered dynamically with the aid of the JavaScript

Figure 1.10: Icicle tree view. This interactive visualization displays a collection of log events from a hypothetical product that operates on three platforms: strawberry, coconut, and banana. Extracted from [68]

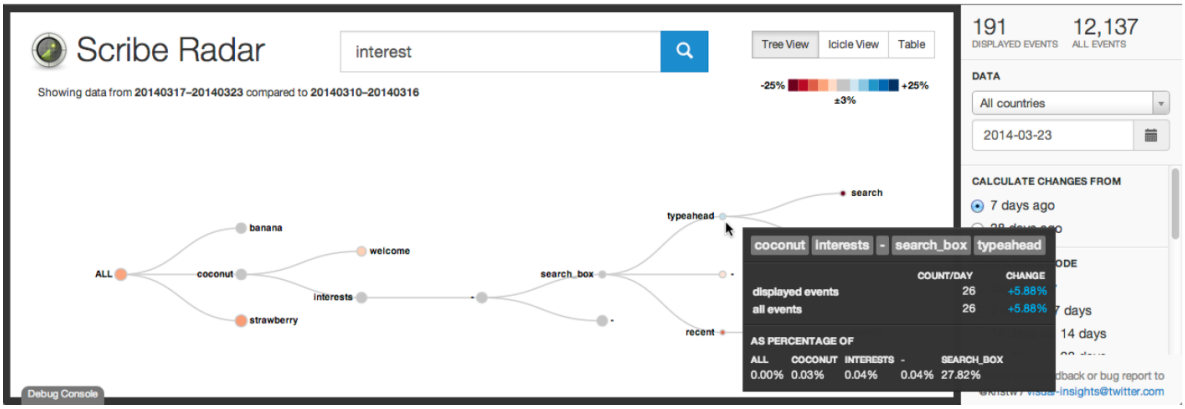

Figure 1.11: Tree view. A node-link diagram view of the event hierarchy. Extracted from [68]

library D3[2], for example an icicle tree and a node link diagram as seen in Figures 1.10 and 1.11 respectively. Furthermore, a visualisation tool is introduced specifically for funnel analysis that captures how many users exactly complete all the events leading towards a goal, for instance buying a product.

[2] https://d3js.org/

Chapter 2

State-of-the-practice on Logging

This chapter presents the results of the preliminary survey on logging practices and the semi-structured interviews on logging practices. The preliminary survey is part of a more extensive study conducted in WP5. We focus here on the aspects questions to logging practices. We conducted the semi-structured interviews with the STAMP use case providers and Eng (since they have a lot of experience as a consulting company). Semi-structured interviews enable interviewees to freely share their thoughts and the interviewer to follow up on and explore interesting topics that might emerge [30].

2.1 Preliminary Survey

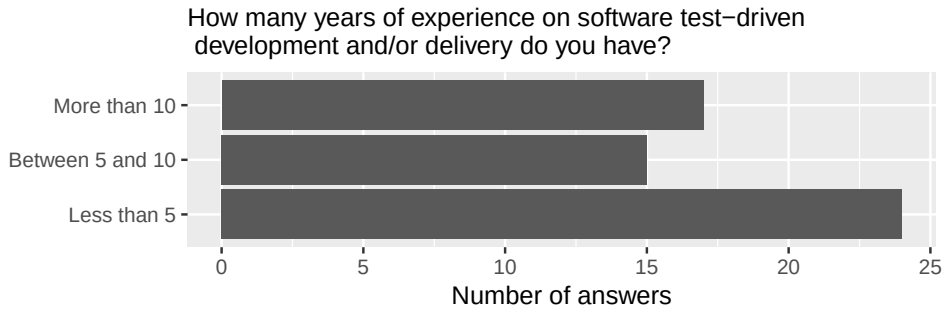

Figure 2.1: Years of experience

The results of the preliminary survey presented here are part of a more extensive open study conducted in WP5. Up to now, we received *56* answers from developers coming from various companies. Figure 2.1 presents the experience in test-driven development or delivery of the respondents. 24 respondents indicated to have less than five years experience, 15 reported to have between five and ten years of experience, and 17 respondents indicated to have more than ten years.

Figure 2.2 shows the presence and usage of logs. 44 respondents indicated that their application uses logs and that they actively use them. Only eight respondents reported that they do not use the logs produced by their application. Two respondents indicated that their implementation does not produce logs, and two respondents indicated that they do not know if logs are generated.

The next question asked if developers can access the logs. Results are presented in Figure 2.3. 24 respondents indicated that developers have access to the complete log. 25 indicated that developers have only access to some parts of the logs. Three indicated that developers have no access to logs, and three indicated not to know if developers can access the logs.

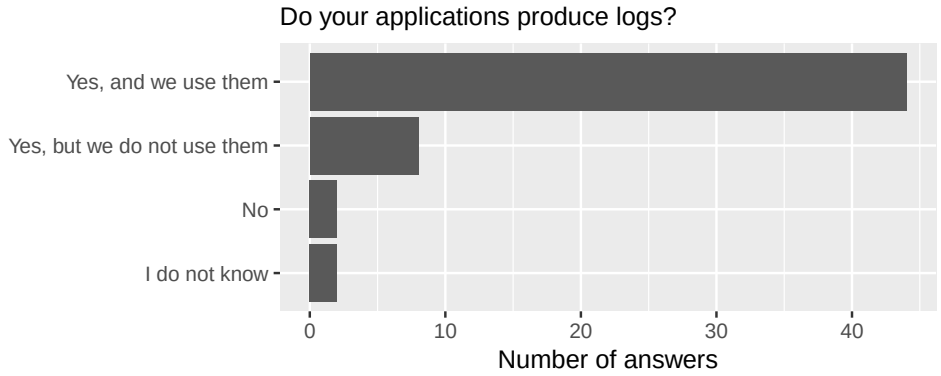

Figure 2.2: Logging usage

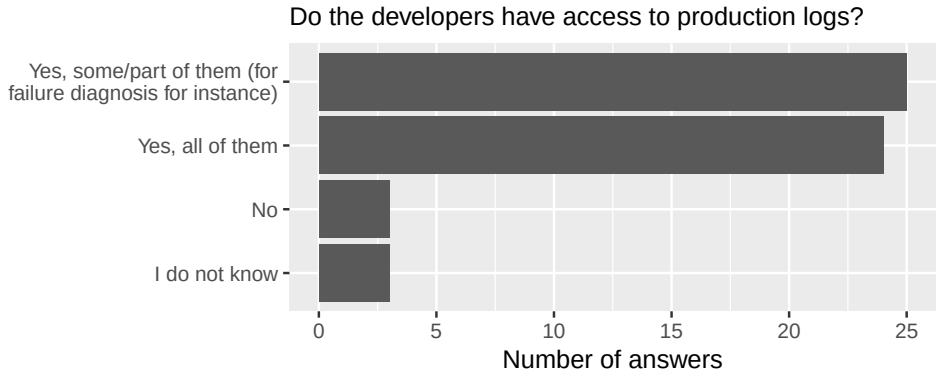

Figure 2.3: Developers access to the logs

Figure 2.4 presents the purposes for which logs are used. More than one answer per participant is possible for that question. 43 respondents indicated that they used logs for application failure diagnosis and that the developers use failure reports in the logs. 37 respondents indicated that logs are used for application failure detection and that failures are detected and reported to developers. 28 respondents indicated that logs are used for anomaly detection and that abnormal behaviors are identified and reported. 27 respondents indicated that logs are used for application monitoring purposes, like liveliness and throughput. And one respondent noted that they also sometimes create logs for specific functionality.

Figure 2.5 shows the distribution of the answers to the question related to logging policy for developers. 29 respondents indicated that there is no common logging policy defined for the developers. 13 stated that the logging policy is common knowledge shared among the developers. Seven indicated that the logging policy is described in a document and enforced by checks during the quality assessment of the application. And four indicated that the logging policy is described in a document but never checked.

For anonymization concerns of the logs, presented in Figure 2.6, 33 respondents indicated that they do not have anonymization concern regarding their logs, 13 reported that they do have anonymization concerns, and seven indicated to not knowing if they have anonymization concerns.

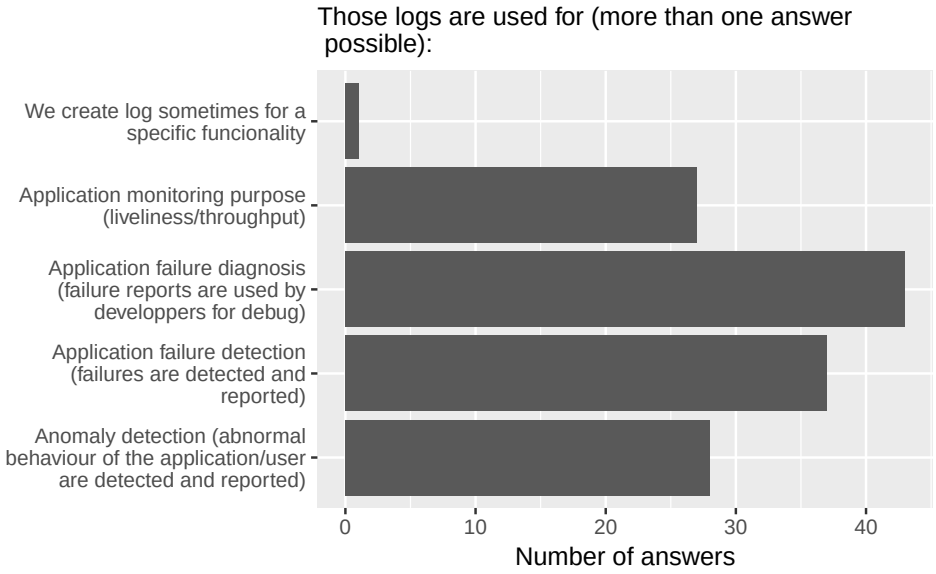

Figure 2.4: Logging purposes

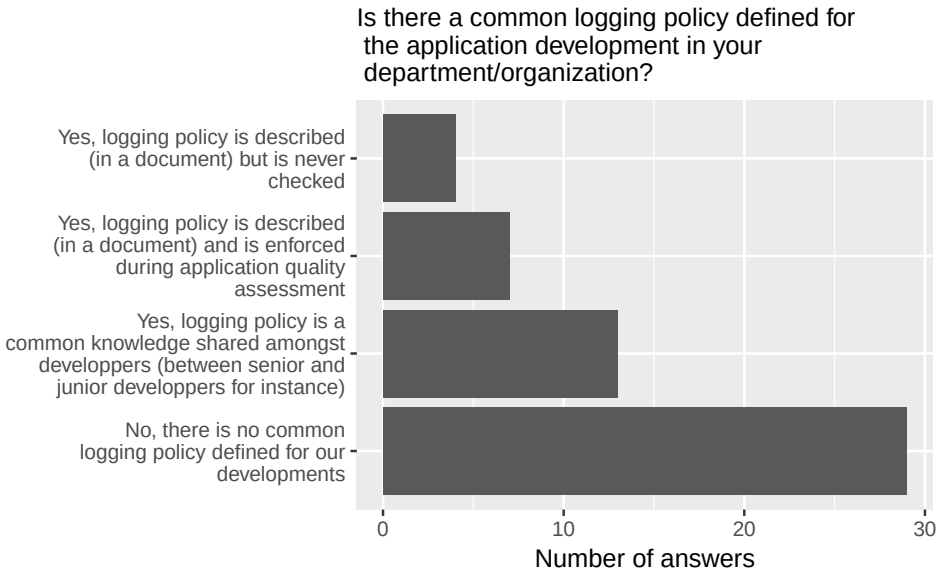

Figure 2.5: Logging policy

2.2 Interview Plan

The interview plan serves as a basis to cover different aspects of logging. We list hereafter the various aspects and the related questions. Since we performed semi-structured interviews, not all questions have been asked to all participants. However, the categories allow to check that relevant aspects have been covered during the discussions.

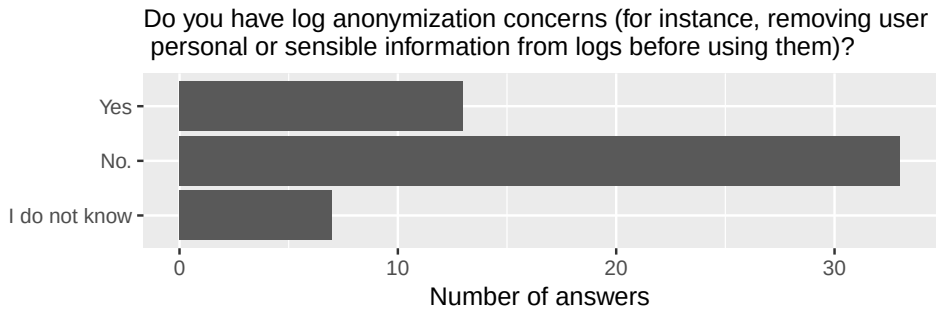

Figure 2.6: Anonymization concerns

2.2.1 Purposes of Logging

This first category of questions is focused on the purpose of logging. Our first general question is:

Question 1 *What is the main goal of logging, in your opinion? Why do you do it?*

Depending on the answer of the interviewees, the question may be refined to address *monitoring* and *debugging* specifically.

Monitoring

The two first questions are related to monitoring tools and their usage. The third and fourth questions are related to failure detection and stack trace collection from the logs.

Question 2 *Logs are often connected with monitoring tools. What kind of tools do you use for monitoring?*

Question 3 *Why did you choose them? What would you want your tool to have, but it currently doesn't?*

Question 4 *Do you automatically detect failures, like stack traces, in your logs?*

Question 5 *Are those stack traces reported to developers? In an issue tracker? If not, how are failures reported to developers? Does this report usually include a stack trace?*

Debugging

Related to the last questions on monitoring, we follow up with questions on debugging techniques involving the use of logs.

Question 6 *Do you see your logs as a debugging technique?*

Question 7 *How do you use logs for this purpose?*

Question 8 *What challenges do you face when using logs for debugging purposes?*

2.2.2 Logging in Third Parties

Since applications may also run off-site, on a client's server, for instance, logs are not always directly accessible or require special treatment (like anonymization) before being used by the developers.

Question 9 *I suppose your customers deploy your software on their premises. How do you deal with such cases, where you don't have access to logs?*

Question 10 *Do you (or they) need to anonymize the logs before sending it to you? How do you do it?*

2.2.3 Logging Policy for Developers

The following questions are related to the *decision* of a developer to add a log statement to the code and the *reviewing* of those log statements by other developers.

Decision Making

Question 11 *How do you decide to add a log statement somewhere?*

Question 12 *How do you decide what to message and what to log, in there?*

Question 13 *How do you decide the log level?*

Question 14 *Do you decide what to log during the requirements phase? Or is it something that happens only at the development phase?*

Code Review

Question 15 *Do you review log lines?*

Question 16 *What kind of feedback do you usually give to people? What kind of feedback do you usually receive back?*

2.2.4 Maintenance and Evolution

Finally, the last questions are related to the maintenance and evolution of log statements in the code.

Question 17 *What makes a 'good log statement', in your opinion?*

Question 18 *What makes a 'bad log statement', in your opinion?*

Question 19 *What kind of maintenance do you often do in log statements? Do you change their message? Why? Do you add variables? Why didn't you log them in the first version? Do you change the log level? Why didn't you get it right in the first attempt?*

Question 20 *What's your heuristic to decide when to delete a log statement?*

Question 21 *If you were to re-write all log statements in your source code base, what would you do? What would the new ones look like?*

2.3 Interviews Highlights

We performed the interviews with the different STAMP partners and provide hereafter highlights for each of them.

2.3.1 Activeeon

The interviewee is a software developer.

Purposes of Logging

Logging is mainly used for debugging. However, developers have to rely on log data provided by the clients of Activeeon to diagnose failures.

Logging in Third Parties

There is a concern about privacy and sharing sensitive data. In case of error reporting, clients have to send log data to Activeeon developers to help them during their debugging tasks.

Logging Policy for Developers

There is no logging for the front-end application, only the back-end services use logging. There are no defined guidelines about logging practices. Log statements may be discussed during code reviews, based on the personal experience of the reviewers.

Maintenance and Evolution

Previously, all log data were saved into the same file, which made it hard to understand the behavior of the system due to its asynchronous nature. Now, each microservice has a dedicated log.

2.3.2 Atos

The interviewee is a software architect and research engineer.

Purposes of Logging

Logging is mainly used for debugging in pre-production environments. When a problem arises, the logs are manually explored using standard command line tools to understand the root cause of the problem. If the problem concerns a crash of the system, or if a stack trace is reported to the developers. The developers will manually add a log statement to the source code, based on the content of the stack trace, to debug the code. This process can be complex and time-consuming because: (i) the developers usually do not have the required information about the context in which the crash happened, and (ii) systems tend to be more and more complex.

Logging in Third Parties

Not applicable.

Logging Policy for Developers

There is no common logging policy. Various logging libraries are used, and there is no standard template for the log messages and logging levels. The interviewee mentioned that there was an attempt to put some logging best practices in place. The idea was to log method calls and parameter values to ease debugging. It did not succeed, mainly due to the low adoption by the developers who do not see the benefits of a consistent logging policy. An idea to solve this issue would be to integrate the logging policy in the build system by checking pre-defined rules in the continuous integration server.

Maintenance and Evolution

The logging code is usually improved after a problem appeared only. It sometimes resulted in a large number of log messages produced by the application. The number of messages was reduced by increasing the logging level.

The interviewee also mentioned that having good logs across systems would be useful because when there is more contextual information upfront, it is easier and faster to analyze failures. There is motivation on raising awareness about logging practices, but it is hard to change the mindset. It is why the company culture plays an important role. Developers preferably invest time implementing use cases than planning how to log the code because they don't perceive the benefits of planning upfront.

Finally, the interviewee pointed out that there is a resistance to introduce new technology on the stack. This resistance comes from concerns about the time to configure, run, maintain, and familiarize with the latest technology. The interviewee relies on command line tools to analyze log data. However, managing and manually analyzing log data is difficult. Some experience of monitoring tools would be beneficial, according to the interviewee.

2.3.3 Engineering

The three interviewees are a research engineer and two software developers. Engineering group is a typical system integrator and has to deal with the different logging practices of their providers.

Purposes of Logging

Logging is mainly used for debugging and monitoring. Logs help to understand the behavior of the system and to diagnose the root cause of a problem. Developers may use the Atlassian tools to compare a given piece of log to a database of previous issues to recommend fixes for the source code. However, a lot of reported issues lack context, especially for the front-end applications that may run on a large variety of platforms (operating systems, web browsers, etc.).

Logs are also used to monitor the system, using dedicated tools. Those tools help to understand what is happening and collect statistics about the usages of the system. In case of detected misbehavior, the tools also trigger alarms.

Logging in Third Parties

To address privacy issues in logging, Engineering has a GDPR compliant solution called *privacy manager*. This framework uses the log entries to audit privacy and security concerns for their clients.

Logging Policy for Developers

Different developers have a different schema and way of logging. Engineering has to deal with those different practices and integrate different frameworks. One of the interviewees mentioned that it is not uncommon to see `System.out` in the source code. Those statements are usually left after a debugging session by developers of other companies.

So far, there are no best practice guidelines defined for Engineering developers. Since teams are small, communication is simple, and practices are shared informally.

Maintenance and Evolution

Since Engineering group is a typical system integrator, there is no product maintenance *per se*. However, one of the interviewees mentioned that usually, developers log statements use the *info* level, which results in performance issues, detected during performance testing.

2.3.4 TellU

The two interviewees are developers at TellU. One of them has more experience with DevOps practices.

Purposes of Logging

The system is built using a microservices architecture. The logs are used primarily to monitor the global behavior of the system, using dashboards and a third-party service. The information level is applied to acknowledge communication between services and allow traceability of the messages, and

debug level is only used for more specific details. Log data are also an essential source of information to investigate reported problems.

Logging in Third Parties

The interviewees mentioned privacy concerns about the log messages sent to the third-party monitoring service and the policy in place to mitigate the risk of leaking information.

Logging Policy for Developers

The TellU system went into a massive refactoring. The interviewees mentioned that there is (not yet) a common logging policy defined. The logging statements are discussed during the code reviews, and reviews are based on the experience of the reviewers.

Maintenance and Evolution

The interviewees explained that during a refactoring process, developers work to adjust log levels and make log entries consistent. For instance, developers remove all error levels that were not logging actual errors. A consistent level of logging improves the analysis of filtered messages.

One of the interviewees also mentioned that logging upfront is challenging. The relevance of the logged information varies with time. For instance, when implementing new functionality, a lot of data are recorded. The logging level will be latter decreased as those are not used by the developers anymore.

2.3.5 XWiki

The interviewee is an experienced developer and knows the product well‘.

Purposes of Logging

XWiki uses different logging for different environments (*e.g.*, server-side logging, or build logging). It helps to understand the different contexts of each service. The interviewee points out that the logs are used for debugging purposes (in the development environment, using breakpoints), but that production logs are sometimes useless since they lack contextual information. For instance, stack traces (generated when a Java exception is thrown in the code) are usually not reported.

There is also no active tool in place to monitor the logs. The current workflow is to look at the logs only when a client reports an incident.

Logging in Third Parties

XWiki may be hosted in a cloud. Most clients choose this option and running XWiki applications are hosted at the XWiki farm. In this case, logs are accessible by the XWiki developers.

The interviewee also pointed out that they have logs translation concerns. For instance, for custom plugins installation and background tasks started by users.

Logging Policy for Developers

XWiki developers follow general rules, enforced by the build system (using *checkstyle*[1]). For instance, the build system fails if the source code prints something to the standard output (`System.out`), or if another logging framework is introduced. Those rules also provide general guidelines on the content and level of the log statements, but developers are relatively free on that regard and rely on common knowledge and code review.

[1] http://checkstyle.sourceforge.net

Code review plays an essential role in the quality of the logs. For instance, the logging level is discussed during the review process. Most logging statements are at the debugging level, and only critical parts use the information (or higher) level.

Maintenance and Evolution

In specific and critical cases, developers need to record more information by adding log statements. For instance, when a new extension is added to the platform. However, this is limited in time. The interviewee also pointed out that some running tasks in XWiki take time and produce lots of log entries. Those tasks should be identified to reduce the number of entries.

2.4 Discussion

In general, results of the preliminary survey and interviews with the STAMP use case providers show that there is a significant disparity in logging practices. For most of the respondents and interviewees, there is no common logging practice, enforced by checks in the build system of the application. Instead, logging practices rely on the experience of the developers and code reviewers. Logs are usually used after a problem happened only, to debug the source code. Several interviewees also pointed out that if logging plays an essential role for debugging, the logs are sometimes useless, due to inadequate or insufficient reporting of the context in which the problem happened.

Conclusion

In this deliverable, we presented a literature survey on the field of log analysis. This literature survey started as a tour d'horizon of the scientific literature on the matter.

In this survey (Chapter 1), we outlined the different techniques and practices introduced by research in the area. We found that the results of the various works do not lend themselves well to a comparison, and suggest future steps of research to overcome this lack of clarity. Furthermore, we suggest areas of improvement regarding the applicability and feasibility of analysis techniques.

Subsequently, in Chapter 2 and with the knowledge that we had obtained from the literature survey in Chapter 1, we interviewed the industrial members of the STAMP consortium to identify their current state-of-the-practice in terms of logging. Our investigation showed that there is a significant disparity in logging practices. Logging practices usually come from the experience of the developers and code reviewers and are rarely documented. Logs are mainly used and enhanced during the debugging of the source code after a problem occurred and has been reported.

Consequences for future work in WP3

The findings as reported in Chapter 2 have some repercussions for WP3 of the STAMP project, namely:

- There is no uniform way of logging that we found among the STAMP project partners, and even different systems of the partners might have different logging approaches.
- Some of the STAMP consortium partners do not have access to logging data from deployed instances of their software (as their customers have privacy concerns).
- Some STAMP partners do log details of exceptional situations in their default log.

The above observations and the knowledge that stack traces, i.e., exceptional situations that occur in deployed software, are reported in the issue tracker, have instilled a slight course shift in WP3 to analyze stack traces, which are small pieces of log information that contain information on exceptional behaviour in the software.

Bibliography

[1] J. Abrahamson, I. Beschastnikh, Y. Brun, and M. D. Ernst. Shedding light on distributed system executions. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 598–599. ACM, 2014.

[2] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD: International Conference on Management of Data*, 1993.

[3] R. Agrawal, R. Srikant, et al. Fast algorithms for mining association rules. In *International Conference on Very Large Data Bases*, 1994.

[4] C. Baier, J.-P. Katoen, and K. G. Larsen. *Principles of model checking*. MIT press, 2008.

[5] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Transactions on Software Engineering*, 41(4):408–428, 2015.

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

[7] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 267–277. ACM, 2011.

[8] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE transactions on Computers*, 100(6):592–597, 1972.

[9] P. Bodic, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. 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.

[10] P. Bodík, M. Goldszmidt, and A. Fox. Hilighter: Automatically building robust signatures of performance behavior for small-and large-scale systems. In *SysML*, 2008.

[11] R. J. C. Bose and W. M. van der Aalst. Discovering signature patterns from event logs. In *IEEE Symposium on Computational Intelligence and Data Mining*, 2013.

[12] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *IEEE International Conference on Autonomic Computing*, 2004.

[13] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 36–43. IEEE, 2004.

[14] M. Cinque, D. Cotroneo, R. Natella, and A. Pecchia. Assessing and improving the effectiveness of logs for the analysis of software faults. In *IEEE International Conference on Dependable Systems & Networks*, 2010.

[15] M. Cinque, D. Cotroneo, and A. Pecchia. Event logs for the analysis of software failures: A rule-based approach. *IEEE Transactions on Software Engineering*, 39(6), 2013.

[16] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *OSDI*, volume 4, pages 16–16, 2004.

[17] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 105–118. ACM, 2005.

[18] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. Van Wijk, and A. Van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pages 49–58. IEEE, 2007.

[19] G. F. Creţu-Ciocârlie, M. Budiu, and M. Goldszmidt. Hunting for problems with artemis. In *Proceedings of the First USENIX conference on Analysis of system logs*, pages 2–2. USENIX Association, 2008.

[20] F. Fittkau, A. Krause, and W. Hasselbring. Exploring software cities in virtual reality. In *Software Visualization (VISSOFT), 2015 IEEE 3rd Working Conference on*, pages 130–134. IEEE, 2015.

[21] F. Fittkau, A. Krause, and W. Hasselbring. Software landscape and application visualization for system comprehension with explorviz. *Information and software technology*, 87:259–277, 2017.

[22] F. Fittkau, P. Stelzer, and W. Hasselbring. Live visualization of large software landscapes for ensuring architecture conformance. In *Proceedings of the 2014 European Conference on Software Architecture Workshops*, page 28. ACM, 2014.

[23] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live trace visualization for comprehending large software landscapes: The explorviz approach. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, pages 1–4. IEEE, 2013.

[24] I. Fronza, A. Sillitti, G. Succi, M. Terho, and J. Vlasenko. Failure prediction based on log files using random indexing and support vector machines. *Journal of Systems and Software*, 86(1), 2013.

[25] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *IEEE International Conference on Data Mining*, 2009.

[26] X. Fu, R. Ren, J. Zhan, W. Zhou, Z. Jia, and G. Lu. Logmaster: mining event correlations in logs of large-scale cluster systems. In *IEEE Symposium on Reliable Distributed Systems*, 2012.

[27] A. Gainaru, F. Cappello, S. Trausan-Matu, and B. Kramer. Event log mining tool for large scale hpc systems. In *European Conference on Parallel Processing*. Springer, 2011.

[28] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli. Mining behavior models from user-intensive web applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 277–287. ACM, 2014.

[29] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu. An evaluation study on log parsing and its use in log mining. In *IEEE International Conference on Dependable Systems and Networks*, 2016.

[30] S. Hove and B. Anda. Experiences from Conducting Semi-structured Interviews in Empirical Software Engineering Research. In *11th IEEE International Software Metrics Symposium (METRICS'05)*, number Metrics, pages 23–23. IEEE, 2005.

[31] L. Huang, X. Ke, K. Wong, and S. Mankovskii. Symptom-based problem determination using log data abstraction. In *Conference of the Center for Advanced Studies on Collaborative Research*. IBM Corp., 2010.

[32] Z. M. Jiang, A. E. Hassan, P. Flora, and G. Hamann. Abstracting execution logs to execution events for enterprise applications (short paper). In *IEEE International Conference on Quality Software*, 2008.

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

[34] S. Kabinna, W. Shang, C.-P. Bezemer, and A. E. Hassan. Examining the stability of logging statements. In *IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2016.

[35] K. Kc and X. Gu. Elt: Efficient log-based troubleshooting system for cloud computing infrastructures. In *IEEE Symposium on Reliable Distributed Systems*, 2011.

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

[37] W. Li, J. Han, and J. Pei. Cmar: Accurate and efficient classification based on multiple class-association rules. In *IEEE International Conference on Data Mining*, 2001.

[38] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen. Log clustering based problem identification for online service systems. In *ACM International Conference on Software Engineering Companion*, 2016.

[39] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining invariants from console logs for system problem detection. In *USENIX Annual Technical Conference*, 2010.

[40] B. L. W. H. Y. Ma. Integrating classification and association rule mining. In *International Conference on Knowledge Discovery and Data Mining*, 1998.

[41] F. Machado. Cpar: Classification based on predictive association rules. 2003.

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

[43] O. Maimon and L. Rokach. *Data mining and knowledge discovery handbook*, volume 2. Springer, 2005.

[44] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios. Clustering event logs using iterative partitioning. In *ACM SIGKDD: International Conference on Knowledge Discovery and Data Mining*, 2009.

[45] L. Mariani and F. Pastore. Automated identification of failure causes in system logs. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 117–126. IEEE, 2008.

[46] L. Mariani and M. Pezzè. Dynamic detection of cots component incompatibility. *IEEE software*, 24(5), 2007.

[47] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, and Y. Brun. Behavioral resource-aware model inference. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 19–30. ACM, 2014.

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

[49] A. J. Oliner, A. V. Kulkarni, and A. Aiken. Using correlated surprise to infer shared influence. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 191–200. IEEE, 2010.

[50] K. 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.

[51] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo. Industry practices and event logging: Assessment of a critical software development process. In *IEEE International Conference on Software Engineering*, 2015.

[52] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1), 1986.

[53] J. R. Quinlan. *C4. 5: programs for machine learning*. 1993.

[54] A. Rabkin, W. Xu, A. Wildani, A. Fox, D. A. Patterson, and R. H. Katz. A graphical representation for identifier structure in logs. In *SLAML*, 2010.

[55] J. Radatz, A. Geraci, and F. Katki. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, 1990.

[56] T. Reidemeister, M. Jiang, and P. A. Ward. Mining unstructured log files for recurrent fault diagnosis. In *IEEE International Symposium on Integrated Network Management and Workshops*, 2011.

[57] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Upper Saddle River, 2003.

[58] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *ACM SIGKDD: International Conference on Knowledge Discovery and Data Mining*, 2003.

[59] F. Salfner and S. Tschirpke. Error log processing for accurate failure prediction. In *USENIX Conference on Analysis of system logs*, 2008.

[60] F. Salfner, S. Tschirpke, and M. Malek. Comprehensive logfiles for autonomic systems. In *IEEE International Parallel and Distributed Processing Symposium*, 2004.

[61] W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang. Understanding log lines using development knowledge. 2014.

[62] Z. Shen, J. Wei, N. Sundaresan, and K.-L. Ma. Visual analysis of massive web session data. In *Large Data Analysis and Visualization (LDAV), 2012 IEEE Symposium on*, pages 65–72. IEEE, 2012.

[63] B. C. Tak, S. Tao, L. Yang, C. Zhu, and Y. Ruan. Logan: Problem diagnosis in the cloud using log-based reference models. In *Cloud Engineering (IC2E), 2016 IEEE International Conference on*, pages 62–67. IEEE, 2016.

[64] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. Salsa: Analyzing logs as state machines. *WASL*, 8:6–6, 2008.

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

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

[67] V. Vapnik. *The nature of statistical learning theory*. Springer Science & Business Media, 2013.

[68] K. Wongsuphasawat and J. Lin. Using visualizations to monitor changes and harvest insights from a global-scale logging infrastructure at twitter. In *Visual Analytics Science and Technology (VAST), 2014 IEEE Conference on*, pages 113–122. IEEE, 2014.

[69] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *ACM SIGOPS: Symposium on Operating systems principles*, 2009.

[70] W. Xu, L. Huang, and M. I. Jordan. Experience mining google's production console logs. In *USENIX Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*, 2010.

[71] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. *ACM SIGOPS Operating Systems Review*, 50(2):489–502, 2016.

[72] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *ACM SIGARCH: Computer Architecture News*, volume 38, 2010.

[73] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage. Be conservative: enhancing failure diagnosis with proactive logging. In *USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[74] D. Yuan, S. Park, and Y. Zhou. Characterizing logging practices in open-source software. In *IEEE International Conference on Software Engineering*, 2012.

[75] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems*, 30(1), 2012.

[76] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *OSDI*, volume 14, pages 629–644, 2014.

[77] Z. Zheng, Z. Lan, B. H. Park, and A. Geist. System log pre-processing to improve failure prediction. In *IEEE International Conference on Dependable Systems & Networks*, 2009.

[78] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang. Learning to log: Helping developers make informed logging decisions. In *IEEE International Conference on Software Engineering*, 2015.