Finding Relevant Errors in Massive Payment Log Data

Version of January 10, 2017

Peter Evers

Finding Relevant Errors in Massive Payment Log Data

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Peter Evers
born in Lelystad, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Adyen Simon Carmiggeltstraat 6 - 50,
1011DJ
Amsterdam, the Netherlands
www.adyen.com

© 2017 Peter Evers.

Finding Relevant Errors in Massive Payment Log Data

Author: Peter Evers
Student id: 4031964
Email: P.H.Evers@student.tudelft.nl

Abstract

Logs play an important role in debugging and maintaining large applications. When a system fails, developers investigate the log records to gain insight to identify the problem. Traditionally developers used tools like *grep* and *tail* to identify irregular behavior. This approach is time consuming for large companies that can generate over 600GB of log data each day. We propose Logness, an approach that clusters raw log data and uses developer feedback to prioritize important log messages. Through experiments on live log data at Adyen, we show that our approach is able to capture subtle, but serious mistakes from the log data. In this thesis, we also share some of our success stories and lessons learned while applying Logness in the wild.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
Committee Member:	Dr. Ir. J.A. Pouwelse, Faculty EEMCS, TU Delft
Committee Member:	Dr. S. E. Verwer, Faculty EEMCS, TU Delft
Committee Member:	Dr. M. Aniche, Faculty EEMCS, TU Delft
Company Supervisor:	MSc. M. Lobbezoo, Adyen

Preface

What started as an interest in electronics and computers at a young age can now finally be concluded with a master thesis in Computer Science at the Technical University of Delft.

Many thanks go to the people of Adyen, who generously provided me the opportunity to work in an interesting field of software technology, while being part of an exciting start-up. The valuable discussions with Maikel and his boundless energy helped me tremendously in finishing this project. The discussions with Mark and Bert shaped the direction of this research such that it is of additional value for Adyen as well. Together with the enthusiasm of Michel, I am proud that I have realized a project that is live and serving a company that is one of the best Payment Service Providers out there.

I would like to thank my supervisor, Prof. Arie van Deursen, for the excellent guidance and help throughout my research. His excitement and encouragement helped me to stay positive even when I struggled to see the relevance and significance of my project.

My girlfriend Eline, was also of great help in supporting me in the ups and downs of this project. She devoted her time even when she was buried in the work of her own master thesis.

Completing this work would have been all the more difficult without the help of Joop. I was amazed by the time he willingly spent, to read my paper and continuously discuss the development of my research.

Special thanks goes to Maurício, who helped me throughout the thesis and especially helped me the last 4 weeks fixing my thesis. Much work was needed to shape the thesis in such a way that it is of an academic level. I was happily surprised that he read, and gave his knowledgeable feedback about every sentence I wrote.

Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1
1.1 Our Industry Partner: Adyen . . . 3
1.2 Structure of the Thesis . . . 5

2 Background 7
2.1 Related Work . . . 7
2.2 Industry Solutions . . . 10

3 Logness 13
3.1 Why Building Our Own Solution? . . . 13
3.2 Design of the Tool . . . 14
3.3 Step 1: Filtering Mechanism . . . 14
3.4 Step 2: Extract Features . . . 15
3.5 Step 3: Aggregation Mechanism . . . 16
3.6 Step 4: Visualize . . . 18
3.7 Real-world Example . . . 21

4 Evaluation Approach 23
4.1 Research Questions . . . 23
4.2 Evaluation Method . . . 24

5 Results 27
5.1 RQ1: Accuracy of the Aggregation Algorithm . . . 27
5.2 RQ2: Logness in the Real-World . . . 30
5.3 RQ3: Logness as Monitor Instrument . . . 42

5.4 RQ4: Training Effort 46

6 Conclusion **49**
6.1 Implications 50
6.2 Limitations 50
6.3 Beyond Adyen 51
6.4 Future Work 51

Bibliography **53**

List of Figures

1.1 Schematic view of stakeholders (and their relations) in a Payment Network . . 5

3.1 Workflow overview, the dashed box represents Logness. 15
3.2 Flowchart describing the classification algorithm 17
3.3 Dependencies of a log message. The arrows indicate the flow of an exception through different classes.
3.4 Screenshot of Logness showing 0 severe issues, 6 known issues and 6 unknown issues 20
3.5 Details of a log message in Logness. 21
3.6 No trendline visible from the error/warning logs in the current stack 22
3.7 Trendline discovered by Logness (same time-frame as in Figure 3.6 22

5.1 Number of issues and annotation type every day recorded by Logness. 46
5.2 Type of exceptions and severity collected by Logness as a percentage of the total. 47

Chapter 1

Introduction

Understanding the behavior of a system in its production environment can be challenging. Commonly, when a system fails, developers investigate the log records to gain insight and identify the problem. Thus, it is clear that logs play an important role in debugging and maintaining large applications, as they capture the events providing information about the execution of the program.

In many cases, this is done manually by means of, for example, searching for simple keywords such as “error” or “exception” in the log data. However, analyzing logs manually does not scale, as it is time consuming and error-prone. Our industry partner Adyen[1], a Payment Service Provider (PSP) and unicorn start-up, deploys code changes at least every two weeks containing thousands of changes serving millions of customers indirectly. As a consequence, over 600GB of log data are generated every day. This log data is a rich source for analyzing the performance of different operations and troubleshooting problems. However, with millions of log messages generated every day, log management tools become crucial.

Regarding log management, Adyen is currently using the following practices:

1. **ELK stack.** The Elasticsearch-Logstash-Kibana (ELK[2]) stack is a popular choice by many companies such as LinkedIn, StackOverflow and Netflix. Elasticsearch is a powerful database and search engine, Logstash structures and normalizes inconsistent log data, and Kibana is a dashboard that provides visualizations. The stack is optimized to run on a cluster of machines and process log data at high speed. Answers to questions such as *1) How many errors/warnings do we produce each second?* or *2) Which queries are running for a long period of time?* can be found quickly using the ELK stack.
2. **Custom monitoring.** Adyen has several custom monitors in place that alerts the developers in certain scenarios. Custom alerts trigger under set conditions and thresholds. For example, when the number of “error” log messages (e.g. a connectivity issue with an external party) is above a certain threshold in the last 15 minutes or when a certain part of the application starts producing errors the system will alert the

[1] `https://www.adyen.com`.

[2] `https://www.elastic.co/webinars/introduction-elk-stack`.

developers. Other monitors keep an eye on the transaction level and alert when there is a sudden drop in the transaction level. When these alerts are triggered, the duty team of Adyen is responsible for investigating and fixing the problem.

Even with these tools in place, it is not uncommon to miss out on issues. The size of the product and the hundreds of different parties involved in the platform, makes it difficult to verify the stability of application after a release. Much too often issues are unfold only when a customer starts complaining. We noticed the following shortcomings in the current practice:

1. **Noise in the error/warning graph**. It is not uncommon that the log files contain more than 1 million warning logs every day[3]. Note that many of them are not actually real issues, as aforementioned. Thus, due to the large amount of noise that is generated from the application it is hard to see infrequent issues. This amount of warnings are generated by the hundreds of external parties involved in the business of Adyen. Therefore, it becomes hard to detect new issues because they are buried under the existing noise. Changing the verbosity of the logs does not have the desired effect as most of the time, the log statement is dependent on the context. For example, an incidental timeout to an external party is not problematic most of the times. However, continuously exceptions with no successive retries is a problem that needs to be reported.
2. **False positives generated by the customized alerts.** The current monitoring practice generates too many false positives keeping developers awake during the night. The alerts are triggered whenever a burst of errors occur. These errors are often generated by external parties and the developer on duty, can most of the time do nothing more than wait until the issue is resolved by itself.
3. **Creating custom triggers is hard**. Creating custom triggers is a powerful tool in system monitoring, but it is often difficult to produce all possible conditions. One must know what to expect when the system fails, however in practice this is much harder to predict, as also pointed out by Mariani et al. [13].
4. **Poor insight in the logging itself.** There are important questions that Adyen still can not answer with their current log stack, such as *1) Which log statements are producing most of the messages?* or *2) What is the importance of this error?* because the log data is dynamic and hard to analyze automatically. The ELK provides a powerful way to search and visualize specific issues but it is not able to detect these anomalies in raw log data.

As a motivational example, on September 5, 2016 Adyen generated 480 million logs of which 1 million are warning and error logs. During the day an issue rose involving a credit card with an unusual card length: most credit card numbers consist of 16 to 19 digits and start with a number between 1 to 6; however, in some countries, new credit card numbers have been released using a number that starts with “95” and that is 19 digits long. In the payment network, the number is converted to a long-type, causing an overflow resulting in

[3]Measured on November 27, 2016 from all applications and machines as an average over the preceding 7 days.

failed payments. This error was not caught because it was flooded by the large number of warnings generated every day. Although the issue only affected a few people every month, the issue is not negligible as every month more credit card numbers are released. Eventually, a merchant starts to notice these issues and can decide to leave Adyen because of bad user experience.

In this thesis, we propose a solution at Adyen that is able to uncover issues that are hidden in the log data: **Logness**. The tool clusters logs together based on a couple of heuristics. They are then presented to the developers and prioritized based on the frequency in which they appear and the feedback that is given by the developers. By examining a small, previously unseen log clusters, the accuracy is improved such that only relevant issues are shown to the developers. The tool runs on top of Adyen' current stack without any changes to the infrastructure, source code or deployment process.

Through a series of experiments on live log data from Adyen, we show that 1) our tool is effective and outperforms the current monitoring practice by detecting many issues that were otherwise gone unnoticed, and 2) during deployment, we show that our tool is effective in detecting issues that are caused by a new release.

1.1 Our Industry Partner: Adyen

Adyen has been founded in 2006 and is now one of the most successful unicorn start-ups in Europe. Adyen has created a formula, which makes the company culture more explicit. The formula consists of a list of points to create a long term value for their customers. One of these points is related to the work in our research: “We launch fast, iterate, and don’t stop until it works”. This can be difficult as a code churn will also mean more changes and a higher chance of bugs. A similar motto was introduced by Mark Zuckerberg in 2012: “Move fast and break things” [3], a motto that seems to be contradictory to the Adyen formula. However, in 2014 Facebook changed their motto to: “Move fast with stability”. Not surprisingly with an ad revenue of 17 billion dollars. This emphasizes that change is important but stability as well.

We see Adyen as a valuable partner when it comes to research log data because 1) more than 500 million logs are generated every day making it a valuable source for detecting problems automatically, 2) the scale of the company and the importance of uptime create an excellent opportunity for us to detect problems as soon as possible, and 3) thousands of code changes every two weeks indicate a company with a high code churn and thus higher chance of issues during a release.

1.1.1 Logging and Monitoring Practices at Adyen

The *de facto* standard for logging on Java application level in 2016 is Log4J[4], a library specialized in logging, which is also used by Adyen. The library supports custom log levels, different formats and custom appenders. All logs that are produced via the Log4J library

[4] `http://logging.apache.org/log4j`.

are normalized by Logstash and structured in a JSON format. The logs are then sent to the Elasticsearch server and stored for at least two months.

Logs can be stored in different log levels to indicate the severity of the message. For example, logs can be prefixed by a “INFO”, “WARN” or “ERROR” level. There is some debate on which log levels to use and what level of verbosity a specific error should have [21]. Adyen is using the log levels “INFO, “WARN”, “ERROR” and “DEBUG”. Developers are mostly free in using the log levels, but should be considerate in using the “ERROR” log level as it can literally wake up the duty team in the middle of the night. The source code contains about 12,000 “WARN” and 1,500 “ERROR” log statements.

These logs vary from data about transactions to error logs about system failures. As we discussed before, most of the million warning and errors log messages are not real issues. We found that the log messages originate from about 300 different log statements, which means that we can ignore a great deal of the log data (we discuss this in more detail in Chapter 5). In practice, we also notice that these log messages are not always meaningful as the log statements originate from a wide group of developers and often contain domain specific knowledge not shared by all who have access to the logs.

As said before, Adyen is using ELK for monitoring, debugging and analytics purposes as it provides a powerful mechanism for full text search, real-time analytics and high-availability. Logging is tightly coupled to system monitoring as logs are used to analyze the system performance and usage trends. Whenever a release is deployed on a machine, a developer is assigned to watch for changes in the log files with tools like *grep* and *tail*. In the meantime the overall system performance is watched using the Kibana dashboard. Different metrics such as response times to different parts of the system and an error rate are visible in the dashboard. Sudden rise in response times or error rates can be caused by the release and require investigation.

Another tool used for monitoring is called *SysMon* and is internally developed. The tool watches for specific events such as errors on acquirers, error spikes in the logs, and authorization rates. Customized alerts are created to notify developers in the case of certain events and possibly, wake them up during the night. The combination of the ELK stack and SysMon for monitoring is referred to as “current monitoring practice” throughout this research.

1.1.2 Payment Industry

To better understand the complexity of the Adyen case, a short introduction in the payment network is necessary. Nowadays, a customer of a webshop can shop online effortlessly by entering a couple of numbers, which settles the payment in a flinch. For merchants, this process is much more complex as they have to deal with a network of multiple (different) stakeholders, a changing environment and regulations. Adyen solves this problem by offering merchants an online and offline service for accepting payments all over the world.

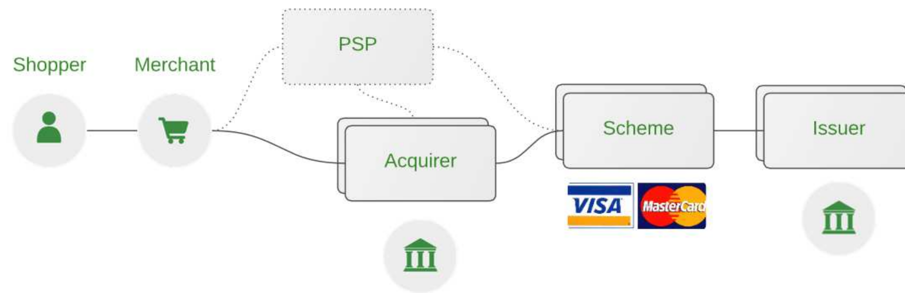

Figure 1.1: Schematic view of stakeholders (and their relations) in a Payment Network

The payment network can be rather complex as it involves many parties as shown in Figure 1.1. In this Figure, a shopper can initiate a payment from a merchant. The merchant is connected to an acquirer and the Payment Service Provider (PSP) Adyen. The acquirer is a bank that captures and processes payments on behalf of the merchant. The acquiring bank is connected to schemes such as VISA and MasterCard. The schemes handle the communication between the acquiring bank and the issuer (bank of the shopper). Large merchants often need different acquirers in order to maximize shopper reach and revenue.

A connection to an acquirer is accompanied by a range of complexities as each connection requires a different agreement, contract and implementation. All of these requirements need to be maintained. Most merchants want to focus on their core business instead of the payment related part. A PSP takes the complexity of the payment network out of the hands of the merchant by managing the full payment flow.

1.2 Structure of the Thesis

The thesis structure is as follows: in Chapter 2, we discuss the state of the art in the industry of logging and log analysis. In Chapter 3, we discuss the tool that we propose to the stated problems. In Chapter 4, we formulate our research questions and the evaluation of each question. In Chapter 5, we discuss our results and in Chapter 6, we discuss the implications our tool and future research directions as well as the conclusion to our approach.

Chapter 2

Background

This chapter is divided in two sections. The first section covers related research on the topic, such as log clustering and anomaly detection. The second part of this chapter presents existing solutions in industry; we also discuss why we built our own solution.

2.1 Related Work

Clustering logs is an essential data mining technique to group entities into clusters. Each group is distinct and the items in the group are similar to the other items in the group. As a data mining function, cluster analysis serves as a tool to gain insight into the distribution of data to observe characteristics of each cluster. In log analysis, clustering can be useful as a first step in reducing the number of elements to work with. However, clustering logs together based on the log statement they originate from is not trivial. Logging stack-traces is expensive and is therefore not used in all log statements. In Section 2.1.1 we provide an overview of the various techniques that are described in literature.

In data mining, anomaly detection is the identification of items, events or observations which do not conform to an expected pattern or other items in a dataset [2]. Typically this translates to some kind of problem such as a bug or fraud attempt. In log analysis, anomaly detection can be used to detect abnormal behavior in the logs. Since the logs contain a wide spectrum of data, interesting events can be extracted such as fraud attempts, bug fixes or hardware failures. We further discuss anomaly detection in Section 2.1.2.

2.1.1 Log Clustering

Various techniques have been used to pre-process log data and identify common patterns. Note that it is often difficult to mine patterns of event types from raw event logs. For example, the message *"Router router1 interface 10.10.10.10 is down"* and *"Router router2 interface 10.10.10.12 is down"* represent the event type "Router interface down" and correspondent to the pattern *"Router * interface * down"*. Log messages rarely contain explicit event type codes and thus need various pre-processing techniques. Vaarandi et al. [16] have used a variation of the FP-growth algorithm to identify frequent patterns in log data. The FP-growth algorithm is a technique, proposed by Han et al. [7] to efficiently mine frequent

patterns from a data set. The core of this method is the usage of a special data structure named Frequent-Pattern Tree (FP-Tree). Log pre-processing is done on individual words in the raw log entries. The experiment is not done on real-time log data and because the algorithm is expensive, Vaarandi only considers raw logs that exceed a certain threshold. This means that infrequent behavior is ignored, which is a typical case we specifically want to catch.

Another promising solution is the log mining algorithm by Xu et al. [19]. In this approach, a log map is built from the source by creating regular expressions from log statements. An example is the statement: "Log.warn("Invalid card number provided pspreference 123123123")". This is translated to the regular expression: "Invalid card number provided pspreference (.*)". A set of these expressions is extracted from the source code and matched against each log entry to find the best match. Because of the structure of the regular expressions, it is painless to extract the value of the variable parameters.

As has been pointed out by the research of Nagappan et al. [14], there are a couple of drawbacks in the work of Xu et al. First, access to the source code is required to extract regular expressions. This is sometimes problematic if the people who carry out the analysis are different from the people who develop the application. A second problem is that the research of Xu et al. was carried out on an application that only consists of 100 unique log statements. Every log statement requires a regular expression and produces overhead that can become problematic for large applications.

The research of Jiang et al. [8] is similar to our proposed log clustering approach. Four steps are proposed: anonymize, tokenize, categorize and reconcile. In the anonymize part they will pick words that are likely parameters. Most likely that are words followed by an '=' or the value following the words 'is–are–was–where'. These values are replaced by a template variable. In the tokenize step, logs with similar characteristics are packed together in bins. In the categorize part, all bins are traversed to match the log lines and group the message into a bin. The reconcile step is to match groups together that only differ one word. This approach is similar to ours, as we roughly take the same steps but our anonymize and tokenize steps differ in implementation. Details of our implementation can be found in Chapter 3.

The research of Fu et al. [5] is using a similar technique in which empirical rules have been set up to cluster logs together. These rules consist of parsing out numbers and grouping logs bases on the edit distance in words. Operated words at the beginning of a raw log message should be more significant than words at the end, because most log messages consist of a static first part and then parameters. We have proposed similar heuristics for matching raw log messages (more on this in Chapter 3).

Visualizing log clusters is done in different fashion. The work of Sabato et al. [15] proposes a new way of ranking the logs based on the probability that logs appear. A system is described in which the frequency is calculated for each log cluster and machine. If this cluster is suddenly appearing more often than 'normal' on a specific server, it is ranked higher.

Makanju et al. [12] base their approach upon the clustering algorithm of Vaarandi et al. [16]. The aim of the work is to develop a visualization tool that can be used to view log files based on the clusters produced. The visualization utilizes treemaps to visualize the

hierarchical structure of the clusters.

2.1.2 Anomaly Detection

Various techniques have been described applying anomaly detection to large log data. An unsupervised learning algorithm has been applied to a large Google production system by Xu et al. [18]. In this paper the authors have parsed the source code to identify print statements and extracted features from these messages. Then Principal Component Analysis (PCA) is used to identify the 'normal' state of operation on a sequence of logs to detect anomalies. The result is then visualized using a decision tree from the work of Witten et al. [17] wherein red nodes indicate an abnormal event.

Another approach is the work of Mariani et al. [13] in which a Finite State Machine (FSM) is built by extracting the log event sequences. Anomalies are detected by keeping track of the current event sequences. From successful event sequences a model is generated that will alert on event sequences that deviate and could potentially form an issue. Building a FSM from log data is expensive and that is why this research is only done on small data sets (less than 50 MB).

Fu et al. [5] built an FSM by recording the log event sequences from successful job executions. This requires however that after each source code change the FSM needs to be rebuilt. Another issue is that it can be difficult to extract a 'correct' sequence of log events as there could be many. If a log key sequence can be generated by the FSA, then they are not considered an anomaly. Otherwise, the first log key in the sequence that can not be generated is reported as a workflow error.

Similar to anomaly detection on a FSM is the approach of Lin et al. [10] in which they create a vector containing word information from a sequence of log messages. This sequence is extracted from a unique 'taskid' that is supplied to correlated log events. Most of the event sequences are extracted from a local test run and stored in a database. After this step, the system is deployed on production servers and sequences that deviate from the knowledge base are reported back to the developers. This approach is difficult in the case of Adyen as the system is so complex and heterogeneous that capturing a 'normal' state of operation is difficult.

Lou et al. [11] approach the problem of characterizing log message sequences in a similar fashion, by extracting the work flow of a program by mining the parameter values in log messages. They categorize log messages based on the same parameter values and detect invariants in the flow and report them to the developers.

These log event sequences can be extracted from log traces. There has been quite some effort in generating log traces efficiently by Fonseca et al. [4] and Gregg et al. [6]. However, progress has been slow mainly because there is no standard for structured tracing. It is technically also difficult to design a standard to accommodate all information contained in console logs in a structured format in an efficient way [18].

2.2 Industry Solutions

There are multiple tools available for logging, log analysis and, anomaly detection. In this section we compare these tools and discuss their strengths and weaknesses. We discuss the tools that are commonly known as log analysis solutions and are used by large tech companies such as LinkedIn, Facebook, and Dropbox.

Splunk Splunk is a big player in the Log industry and comes as a SaaS and on-premise solution[1]. It offers a wide variety of visualizations, such as tables, charts, and listings. There are around 400 plugins to customize the tool to the needs of the client. The charting and search tool is feature-rich and can be accessed through an API or UI. The security is extensive and it can process a large range of machine data. The limitations are that it is required to configure and setup a dedicated cluster. This means that there is no lightweight approach for log analysis and that companies probably have to switch from the current logging frameworks to a specific Splunk cluster. Splunk is a relatively expensive option for real-world applications that generate a vast amount of data.

Loggly Loggly[2] is a cloud-based solution and makes use of open source technologies such as Elasticsearch, Lucene and Kafka. The dashboard can be configured with custom widgets and the API can be used to query historical statistics (such as sum, average, variant, etc.) of the events. Loggly reports all the problems extracted from raw log data but also informs the user why the problem occurred and what the cause of the problem is. The tool can automatically identify normal log sequences and anomalies from the live stream of logs. However, Loggly is only offered as a cloud-based solution and is therefore not applicable for Adyen.

OverOps OverOps[3] takes another direction in log management and is focused on delivering stack traces for error messages in production. These stack traces are generated by injecting a JVM agent into the target machine that transfers state information like threads and variables to the cloud. They have also released an on-premise solution very recently. OverOps claims to have minimal impact on the performance on the target machine and comes as a SaaS and on-premise solution. Downside to this approach are the possible security risks and performance overhead. An enterprise JVM agent cannot be verified for correctness. A crash of the JVM can be catastrophic and cannot be easily fixed.

GrayLog GrayLog[4] offers an open-source solution that runs on top of MongoDB and Elasticsearch. Fast log aggregation is possible to search through terabytes of log data to discover and analyze important information. Alerts can be set that triggers on failed login attempts, exceptions or performance degradation. Downside is that the aggregation of

[1] `https://www.splunk.com`.
[2] `https://www.loggly.com/`.
[3] `https://www.overops.com`.
[4] `https://www.graylog.org`.

log data often fails because log descriptions contain variable parameter values. Secondly, knowledge is required in setting up both GrayLog and Elasticsearch.

Sentry Sentry[5] is focused on detecting issues especially after a production deployment. Sentry is able to associate errors with a specific release. Custom tags can be applied to exceptions and Sentry is able to distinguish between different production environments. There is a wide support for different programming languages and there is support for both SaaS solutions as well as an on-premises variant. Downside is that it requires a special Log4J appender which affects the performance of the application.

[5] https://sentry.io.

Chapter 3

Logness

Manually inspecting large amount of log files is impractical. To gather more insights in the issues that persist in the log data, we propose the tool **Logness**. The goal of the tool is to increase monitoring efficiency by clustering logs intelligently. Developer feedback is used to visualize and prioritize log clusters. Our hypothesis is that by using this semi-supervised approach, Logness is able to provide log insights that can solve potential hidden problems.

3.1 Why Building Our Own Solution?

In this section we describe the motivation for building our own solution (Logness). The objectives are:

1. **Runs on top of any existing stack.** Logness should be designed as a light-weight application that can do log analysis using the existing log framework. The algorithm can run in parallel in order to be able to respond faster to real-time events in the log data. Large companies are often hesitant in changing the existing logging infrastructure without the guarantee of major improvements. This is not surprisingly as logs often play a core role in the infrastructure of the application. Besides the infrastructural changes; developers are familiar with the tools of the current stack. Logness should require no change to the existing infrastructure and can run with a minimalistic setup.
2. **Better monitoring capabilities** As mentioned before, our industry partner Adyen is using the ELK stack for log management. The ELK stack is powerful in storing, searching and visualizing logs. However, aggregating log data based on the raw log description often fails because the log statements contain variable parameter values. A powerful visualization of the log statements that generate most of the exceptions or a list of unique, previously unknown log statements is also impossible to realize using ELK. Logness should be able to cluster raw log messages and prioritize them such that more insight is given in the current logging state.
3. **Security and performance considerations.** A Software as a Service (SaaS) solution is not suitable because our industry partner processes sensitive data. Anonymizing log data requires a powerful parser and cripples the performance of the solution. An on-premises solution is therefore required. Enterprise solutions such as OverOps

require an agent to be injected into the JVM. A closed-source agent that can possible crash the JVM or slow down the performance brings serious security risks in terms of stability and performance. Secondly, most of the existing solutions make use of Log4J appenders that log line numbers or stack traces for every warning statement. This is not suitable for our industry partner Adyen, because that can have a dramatic impact on the performance of the system due to the large amount of warnings and errors produced by the system.

4. **Room for experiment.** Experiments on clustering algorithms and visualization can be done without much effort. Altering existing solutions such as the ELK stack can be problematic because the design is fixed for a specific use. By building our own implementation we have the freedom of experimenting with various visualization and clustering methods.

3.2 Design of the Tool

An overview of how Logness works is depicted in Figure 3.1. In the top of the image, logs are streamed to Logness, in our case from Elasticsearch. The logs are then processed through 4 steps. The first step filters out most of the log data that is uninteresting for our application: catching issues. The second step extracts useful features from the raw log data. The third step aggregates log data based on a couple of heuristics. The last step visualizes the log clusters. Developers interact with the visualization to get notified about possible issues and to create annotations to learn Logness to prioritize possible issues. Further details are explained in the following sections.

3.3 Step 1: Filtering Mechanism

Logness filters log messages that are only tagged as warning or error due to the fact that Adyen is logging over 600 million logs every day. This decision was made under the assumption that most issues are logged with a “WARN” or “ERROR” severity level. This assumption is also recognized by Yuan et al. [20]. Applying this filter brings down the total amount of log messages down to approximate 1 million every day.

Logness applies a second filter that ignores warning log messages that do not contain an exception (note that we only apply this filter at the warning level, not at the error level). For example, the following log messages are processed by Logness:

Processed Log Lines
```
12/15 10:00:05 - ERROR: Cannot connect to acquirer
12/15 10:10:02 - WARN: NullPointerException ...
```

Whereas the following log messages are ignored by the filtering mechanism:

Ignored Log Lines

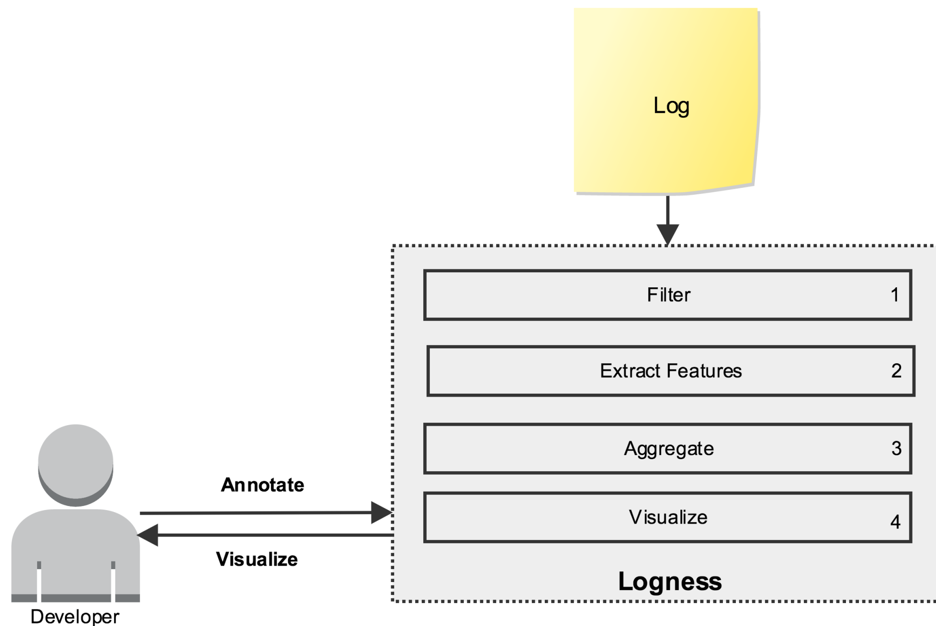

Figure 3.1: Workflow overview, the dashed box represents Logness.

```
12/15 10:07:05 - WARN: Invalid amount specified
12/15 10:09:02 - WARN: Could not load cookie properties
```

The experiment in Chapter 4 reveals that this filter is justified and that we do not miss out on issues because of a filtering mechanism that is too strict.

3.4 Step 2: Extract Features

Features are extracted from the log messages to better cluster and visualize log data. Possible *exceptions* are extracted from the raw log message by utilizing a regular expression. The type of exception is used to make a distinction between custom made exceptions such as a *ServiceException* and standard Java exceptions, such as *IllegalStateException* and *NullPointerException*. If a log message contains a *stack-trace* it is extracted as well such that we can match it against unknown incoming messages. The *classname* and the *name* of the applications are extracted features to faster match incoming log messages. Whenever a new log message enters the system, only a subset of clusters that match the classname and the application name are extracted to run the clustering algorithm on.

3.5 Step 3: Aggregation Mechanism

Log data should be clustered together based on the cause of the problem and if the log message itself does not provide extra information by being a distinct cluster. However, if the clustering is too strict, we loose information.

Log messages that originate from the same log statement should not always be clustered together. A log statement that originates from a top class that produces an abstract error, may describe different issues. The following simplified independent log messages illustrate a case where the same statement should not be clustered together:

```
Log
09/15 11:28:05 - Caught Throwable: java.lang.IllegalStateException
09/15 11:28:02 - Caught Throwable: java.lang.IndexOutOfBoundsException
```

Parameter values in log statements can cause log messages to differ much from each other. Log statements that describe the same issue but involve different merchants, acquirers or even shoppers, *should* be clustered together.

The following example illustrates a case in which the same log statement should be clustered together:

```
Log
09/15 11:29:51 - Could not book settlement report for MerchantA
09/15 11:29:05 - Could not book settlement report for MerchantB
```

The Logness clustering algorithm is able to distinguish between both cases. The clustering algorithm is not a one to one mapping between the logs and the log statements in the code. This is because we are primarily interested in anomalies and not the origin of the log statement per se. In Chapter 5 the algorithm is evaluated to assess the usability of Logness as a log analysis tool.

The algorithm can be described in the flow chart depicted in Figure 3.2. In the top left part a raw log file enters the system (No. 1). The first step is the filtering mechanism to extract only the log messages that are interesting for problem detection.

After this step all the digits are filtered out of the log message (No. 2). If there is no cluster that matches the same application, classname, and filtered log description, a next filtering step is applied to filter out the hashes (No. 3). Hashes contain credit card details or other sensitive information that should not be visible in raw log data. We apply a length check and remove blobs of text that are larger than 256 characters without any space delimiters as hash filtering mechanism.

If there is still no match, the Longest Common Sub-sequence (LCS) algorithm is utilized [1] (Label 4). This algorithm is the basis of data comparison programs such as the Unix diff utility. This problem is NP-hard but can be solved in polynomial time by using a dynamic programming solution. To cap resource cost, only the first 10,000 characters of the log message are compared. The algorithm outputs the longest common sub-sequence of two log messages. If the size of the result is larger than the threshold θ which is 65% of the original cluster size, the log message is said to be “equal”.

Varying the threshold θ can lead to interesting results. Setting the threshold to a low value results in matching almost every message that originates from the same app, classname and exception. So for example, every *ParseException* from all the methods in some *ClassA* will be clustered under the same cluster. This is usually not wanted because issues can be missed easily. Setting the threshold to a high value will result in a detailed clustering algorithm, but can lead to a flood of log clusters. A single parameter influences the algorithm such that a new cluster is created for every small change in parameter values. For example, the message "InvalidResponseException: Invalid response for acquirer ABCD and customer A" and the message "InvalidResponseException: Invalid response for acquirer ABCD and customer B" will not be clustered together for a high value of θ. We observed that most of the business logic is filtered out when using a threshold value of $\theta = 65\%$. We further elaborate on this choice in Chapter 5.

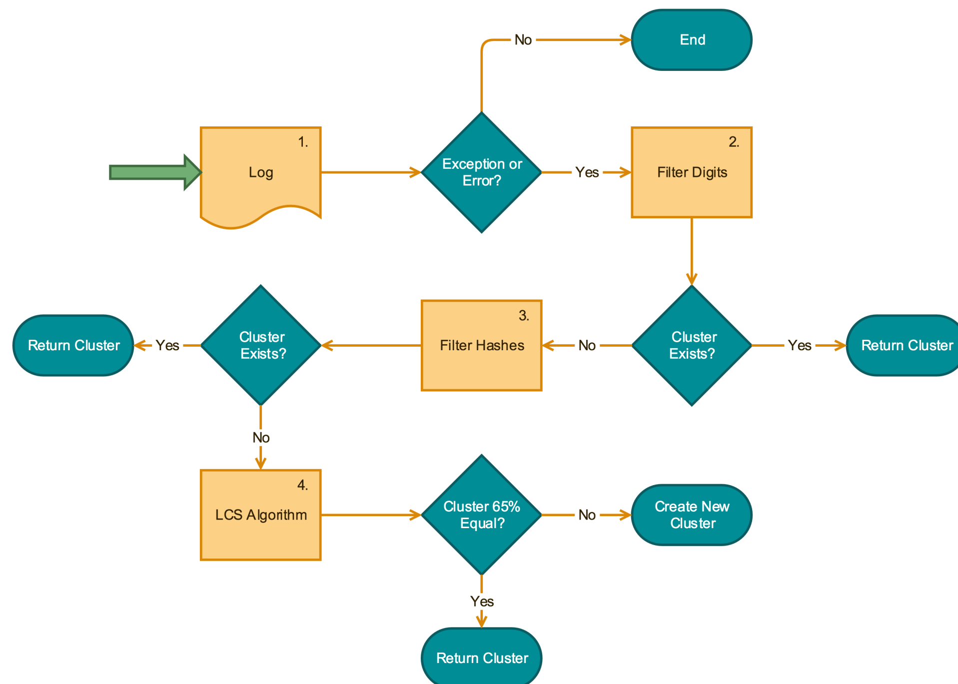

Figure 3.2: Flowchart describing the classification algorithm

We observed that around 300 different log clusters are producing logs every day at Adyen using the algorithm in Figure 3.2. To cut down the overhead cost of annotating and inspecting log clusters, a second clustering step is applied to combine different log clusters. An issue is often logged as a trace of log messages that are involved in the same issue. An example is illustrated in Figure 3.3. In this image, *ClassA* logs a message, then *ClassB* logs a message that is related to the same problem and then *ClassC* logs an abstract error such as: "500 Server Error". Most of the time, it is only interesting to display the message from *ClassA* and *ClassC*, as *ClassE* is only logging an abstract message.

To utilize this, a unique reference number of the execution is used to record the flow of the execution. In the case of Adyen, they use a *pspreference* to record the flow of the execution. Error logs that are tagged with the same pspreference are likely to be correlated and can be used to tie log messages together.

Logness creates a 15 minute cache storing all the processed logs. If Logness finds distinct log messages that belong to the same payment within this 15 minute cache, and thus have the same pspreference, they are combined. This means, that in the case of Figure 3.3 only *ClassA* and *ClassC* are displayed as a possible issue. Most companies store similar references, e.g., user or request id's that can be used in a similar fashion. We chose a 15 minute time window because most related issues appear in this interval. Keeping this window minimal preserves memory capacity.

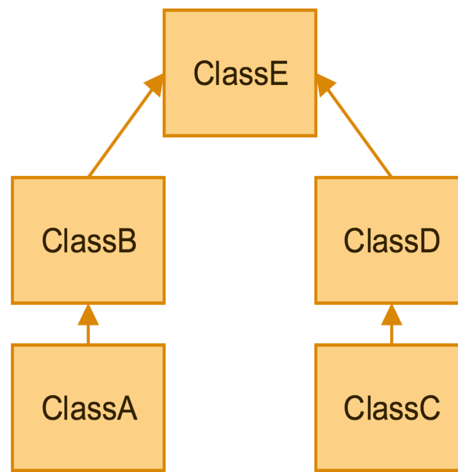

Figure 3.3: Dependencies of a log message. The arrows indicate the flow of an exception through different classes.

3.6 Step 4: Visualize

The created clusters, also called log categories, are visualized in a web application that is accessible for all developers. Initially the clusters are prioritized based on the frequency in which they appear. Prioritization is done by letting developers annotate log clusters as follows:

1. **Severe**. Tagging an exception as severe means that the issue is top priority and an immediate fix is required. The cluster is prioritized in favor of all other exceptions, except for severe exceptions with a higher frequency. In the case of an ongoing patch, a severe log statement on the patched machine should stop the deployment of the patch. Developers should also be notified in the case of a severe log message that is generated by the system.

2. **Bug**. Tagging an exception as bug means that a fix is needed, but no immediate action is required. Examples are simple bugs that cause no direct harm but do impact the performance of the system. Bugs have the second highest level of importance, meaning that they are prioritized below the severe exceptions but above the rest of the issues.
3. **Monitored**. The monitored tag is introduced to actively watch log clusters on infrequent behavior. Depending on the external party, timeouts can result in many logged exceptions. Logness alerts on monitored exceptions whenever the set threshold is exceeded. This threshold is set by the developer using the tool. If this scenario happens, the issue is treated as a "severe" issue and is therefore listed above all other issues.
4. **Non-Severe**. A non-severe issue is the lowest in rank and is useful for hiding unimportant log data. This tag can be used to hide duplicate log statements. For example, correlated log statements that are not combined in the aggregation step can cause multiple log clusters that are all involved in the same issue. Another example is hiding log statements that are involved in business logic such as "Invalid username provided" or external parties that send in bogus data. In the case of Adyen, some terminals in the field are producing unimportant error logs that cannot be fixed right away.

A trend displays the severe, known and unknown issues connected to the table displaying the details of the clusters as we can see in Figure 3.4. We chose to cluster the log messages annotated as "monitored" and "bug" together as a "known" issue in the graph. This is done to make a better distinction between a severe, unknown and known issue without cluttering the graph. A monitored issue that exceeds the threshold, and thus requires attention, is visualized as a "severe" issue in the graph and is prioritized the same as a severe issues.

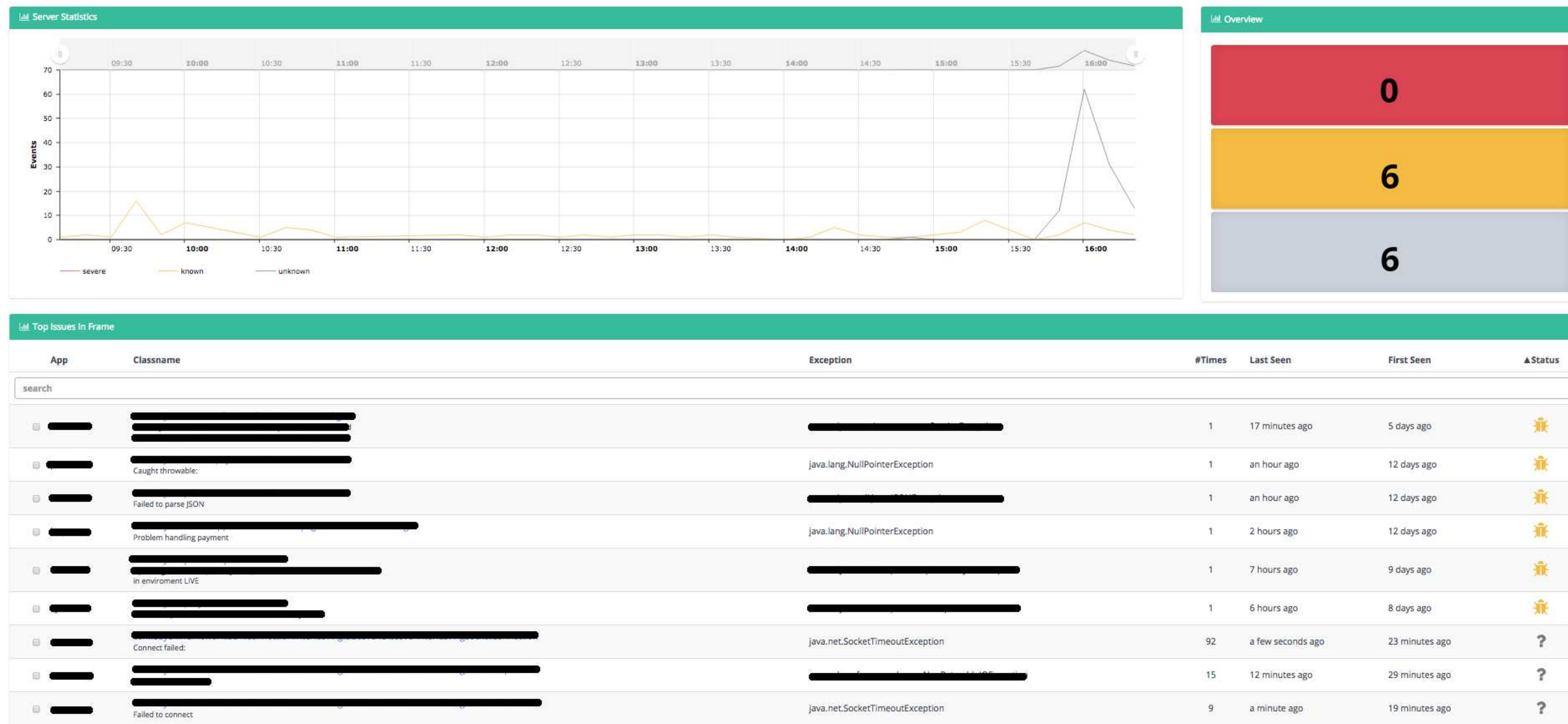

Figure 3.4: Screenshot of Logness showing 0 severe issues, 6 known issues and 6 unknown issues

When an issue in Figure 3.4 is clicked, a detailed view of the issue is opened in a pop-up (see Figure 3.5). Information is provided such as the host machines involved in the issue and the trendline over the last 24 hours. Developers can leave a note to explain the issue, such that other developers know what the cause is and how the issue can be resolved. The trendline in the Figure 3.5 shows that the issues was solved around 11.00AM the next day and quickly diminished. Further inspection of the problem can be done by using the current logging practice, in our case the ELK stack. Logness is able to generate an Elasticsearch query that is helpful in gathering all the individual log messages that are involved in the issue.

Figure 3.5: Details of a log message in Logness.

3.7 Real-world Example

We provide a real-world example to demonstrate how Logness is able to capture issues in the raw log data of Adyen. This example is the same issue described in Section 5.2.11 and happened because of a server that was unable to connect to an external party. The issue was gone unnoticed because of two reasons:

1. The issue was not visible in the error/warning trendline, see Figure 3.6 and Figure 3.7 because of the relative low frequency.
2. The issue was not part of the custom set alarms that trigger when too many errors are generated in a short time span.

Figure 3.4 is a screenshot of the Logness application shortly after the issue first appeared. Although most of the data is anonymized, it is clearly visible that a burst of un-

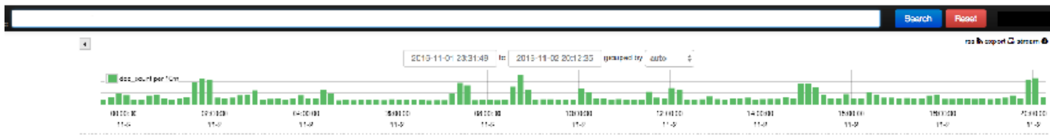

Figure 3.6: No trendline visible from the error/warning logs in the current stack

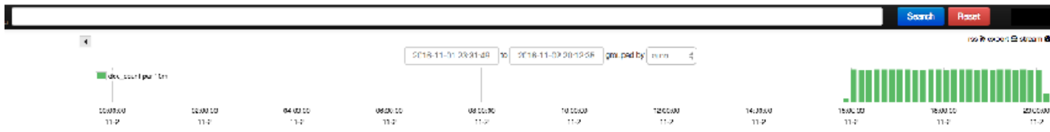

Figure 3.7: Trendline discovered by Logness (same time-frame as in Figure 3.6

.

known issues started to appear around 16:00. In the top-right cell the number of severe (in red), known (in yellow) and unknown (in grey) issues are shown. The table is intelligently sorted to first display the already known issues and after this, list the unknown issues over the last 12 hours. The last 3 issues in the table are involved in the problem described in this section and are dominant in frequency over the other issues. Logness is able to produce a query to investigate the issue in the current monitoring practice. This query is anonymized in Figure 3.7 but the trendline clearly unfolds the issue. Notice that the issue is not visible in the current monitoring dashboard depicted in Figure 3.6.

Chapter 4

Evaluation Approach

In the previous chapter, we have presented Logness, a tool to increase monitoring efficiency and detect possible issues as quickly as possible. In the present chapter, we present our approach for empirically establishing the usefulness of Logness in the concrete Adyen setting.

4.1 Research Questions

Logness makes use of a classification algorithm to filter and aggregate log data to create a visual representation for the developers. The accuracy of this aggregation is important as a poor classification algorithm results in flooding the user with log messages, whereas a too strict aggregation algorithm results in data loss. Therefore our first research question is:

RQ1: How accurate is Logness' classification algorithm?

We explore the use of Logness to identify relevant issues. A relevant issue is an issue that requires action, e.g., reboot a server or patch the system. We consider Logness as "relevant" when the tool is instrumental in finding the issue. Therefore our second research question is:

RQ2: Does Logness support developers in finding relevant issues?

Logs are also important in monitoring the deployment of a new release, as new issues may appear. Manually monitoring issues during a release is a challenge because humans can inspect at most a couple of servers at the same time. We investigate whether Logness also supports developers during release activities. Therefore our third research question is:

RQ3: How does Logness behave around a release cycle?

Log clusters can be tagged as "severe", "bug", "monitored" or "non-severe" (see Section 3.6 for more details on the prioritization). This means that manual work is required from the developers to annotate log data; the more issues the more work they have. In our

fourth research question, we investigate the effort that is required to create these annotations in practice:

RQ4: What is the effort of training Logness?

4.2 Evaluation Method

To answer RQ1, we collect three days of log data, sampled from random office days (Monday to Friday) from October to November. For each of these days, we collect the number of categories created by Logness as well as all the raw log data. We then manually verify for each log if the aggregation was done correctly.

Before starting the experiment, we made sure that Logness was already operational for atleast the last two weeks. We argue that this data is relevant to the evaluation as a category assigned to a log statement might appear to be correct when analyzed in isolation (i.e. analyzing only the logs for a single day), but might be incorrect when analyzed together with more data. As an example, suppose a log statement that contains the month of the year as a word (e.g. it prints "Apr" in April and "May" in May). These logs should always be clustered together regardless of the month that was printed in that log.

A distinct log cluster created by Logness can suffer from the following two issues:

1. Logness clustered logs together that belong to different categories. Possible reason is that the log originated from a different log statement or contains valuable information that identify a separate issue. The log messages are clustered too **strict**.
2. Logness created separate categories for logs originating from the same log statement describing the same problem. This can happen when, for example merchant names, are logged as parameter values causing the log messages to differentiate much. We say that the log messages are clustered too **loose**.

To verify the performance of Logness in aggregating log data, we calculate its precision and recall. To do this, we build a specific table called a confusion matrix. Each row represents the instances in the actual classes *distinct* and *not-distinct*, and each column represents the instances in the created classes *distinct* and *not-distinct*. For example, we parsed 100 logs during the day for which we manually identified 12 distinct log categories. Logness created 10 categories of which 7 are correctly categorized. The precision is $7/10$ while the recall is $7/12$. The first measurement tells us something about the correctness of the algorithm while the second one tells us something about the completeness.

Performance measurements like precision and recall can be calculated from the confusion matrix. In this case, high precision relates to *the quality in the prediction of an issue* as misclassification will mean that we alert the user too much. Maximizing the recall, on the other hand, can be seen as *maximizing the completeness* because misclassification will mean that we miss out on issues.

To answer RQ2, we deploy Logness in a beta-stage on our own system such that we can use the tool ourselves. We have asked one of the experienced developers to help us train the

tool, meaning that he will annotate unidentified log messages.

We carefully monitor the issues detected by Logness and investigate the problem if it has not been noticed within 2 hours of first occurrence. For example, Logness detects a problem at 2 PM and if the developers have not noticed the problem by 4 PM, we start to investigate whether this really is a problem, what could have caused the problem, and how other developers experience the problem. If the issue is actually considered a real problem, we further elaborate why Logness was able to find the issue and why it has not been noticed by the development team.

To answer RQ3, we deploy the tool on the production servers of Adyen. We monitor at least two release cycles using the tool with trained data from us and the developers. During a release, we carefully monitor Logness in order to detect issues during the patch. We report on the following two cases:

1. **Logness detects an issue that was not found by the developers.** We investigate if the found issue was fixed before continuing the patch or that the issue was ignored. If the issue was ignored we investigate why the issue was ignored.
2. **Developers detect an issue that was not discovered by Logness.** We investigate why the tool did not find the issue. There can be multiple causes, for example the classification of logs could be too aggressive. Another cause can be that that the problem is not logged at all. We report on both cases and suggest improvements that can be made to Logness.

Finally, to answer RQ4, we deployed Logness on the live Adyen servers and ran it for 25 consecutive days. Every day, we simulated the work of a developer by analyzing the new generated clusters and performed the annotation; if, at any moment, we could not figure out the best annotation for a specific cluster, we asked help from a developer.

We then collected data about each generated cluster such as creation date, annotation type and the type of exception. We consider "effort" the number of new log clusters created every day. We also analyze the type of exception and the most frequent occurring issues to gain insight in the complexity of annotating issues.

Chapter 5

Results

This chapter answers the research questions proposed in Chapter 4 aimed at assessing the usefulness of Logness. We reflect on the research questions and we discuss the issues we found and solved using Logness as an instrument.

5.1 RQ1: Accuracy of the Aggregation Algorithm

In the following, we present the precision and recall of the executions that we performed in three different days.

5.1.1 Run I

Date:	November 2, 2016 10:40 AM — November 3, 2016 10:40 AM
Total Warnings and Errors Logged:	466,957
Logs Processed:	11,515
Errors:	5,186
Warnings:	6,329
Real Categories:	219
Categories in Logness:	223
Correctly Clustered:	198
Precision:	$198 \div 223 = 88.8\%$
Recall:	$198 \div 219 = 90.0\%$
Logness Version:	1.9 ($\theta = 0.60$)

Logness clustered logs together with high precision (90%) and recall (88.8%). In the manual analysis, we grouped logs into 219 clusters. Logness created 223 clusters of which 25 are incorrect. Errors are caused by clustering logs too strictly (12 classes) or by matching logs too loose (13 classes). Notice that we only looked at 2,5% of the total number of errors and warnings logged during the day.

The most common case of failure when clustering too loose, was caused by a combination of Logstash erroneous extraction of stack traces and minimalistic log statements. Whenever a log message does not contain a description, Logness relies on the stack trace to cluster logs together. Logstash does not always succeed in extracting the complete stack trace which will cause the matching algorithm to fail. A possible fix is to identify if a stack trace is contained within another stack trace. However, large stack traces from different issues often contain similar traces because of the hierarchy of a program.

There are 12 classes that are clustered too strict in Logness. The cause is that messages look alike but resemble a different problem. For example:

```
Log
TransferException:
123123123 STATUS_REDIRECT-AcquirerA CONVERTER_NOT_SUPPORTED_REQUEST
TransferException:
123123123 STATUS_REDIRECT-AcquirerB NO_RESPONSE_WITHIN_TIMEOUT
```

Logness clustered these logs together while it should distinguish between the two cases: 1) a converter issue (response that could not be interpreted from the acquirer) and a 2) timeout issue.

5.1.2 Run II

Date:	November 7, 2016 09:39 AM — November 8, 2016 9:39 AM
Total Warnings and Errors Logged:	1,114,857
Logs Processed:	18,783
Errors:	4,185
Warnings:	14,598
Categories in Logness:	191
Correct Clustered:	172
Real Categories:	192
Precision:	$172 \div 191 = 89.0\%$
Recall:	$172 \div 192 = 89.6\%$
Logness Version:	1.11 ($\theta = 0.65$)

We changed the threshold from a value of $\theta = 0.60$ to $\theta = 0.65$ because after *Run I*, we noticed that the tool missed out on subtle issues because the clustering was too strict. There are more warnings than in the previous run but the amount of errors are less. Recall and precision are similar to previous run and the types of errors are also similar. In 3 classes the clustering algorithm was too loose, in which logs should have been clustered together. In 4 other classes the clustering algorithm was too strict in which logs should have been clustered separately. In 2 of these cases, problems were caused by a cut-off in the stack-trace, the same problem seen in the first run. In the other 2 cases, the variety of parameter values caused a mismatch. The next example illustrates one of the problems:

```
Log
DbException for eventId=123123123,
requestType: MergeShoppers request:
        UnpackedRequest{... JSON ... } Caused by X

DbException for eventId=321321321,
requestType: AddToShopper request:
        UnpackedRequest{... JSON ... } Caused by X
```

Logness created two categories but only one category is required with the cause *X*. The JSON in this example contained information about a shopper including name, address, city and detailed information. Logs from this statement are 50/50 matched against the first or the second cluster because of the various parameters. This edge case also demonstrates the reason why a threshold higher than $\theta = 0.65$ is not desirable to match log messages. Picking a higher value can potentially generate many duplicates.

5.1.3 Run III

Date:	November 23, 2016 09:37 AM — November 24, 2016 9:39 AM
Total Warnings and Errors Logged:	1,100,124
Logs Processed:	21,425
Errors:	5,812
Warnings:	15,613
Categories in Logness:	375
Correct Clustered:	343
Real Categories:	367
Precision:	$343 \div 375 = 91.4\%$
Recall:	$343 \div 367 = 93.5\%$
Logness Version:	1.12 ($\theta = 0.65$)

On November 23, 2016 a small patch was installed on all the machines that are heavily involved in the communication between acquirers and merchants. A short burst of connectivity issues between the merchants and Adyen rose when a machine was taken out of the load balancer. Any pending payments will be rerouted to another server. This caused substantial more log clusters than in *Run I* and *Run II*.

Logness clustered 33 categories incorrectly from 12 classes in this run. However, unlike the previous run, most errors were caused by a loose clustering algorithm (8 out of 12). In the 4/8 classes there was a mismatch caused by the same stack trace issue we described in the previous two runs. In the other 4 classes the clustering failed because Logness could not identify similarities in the log messages. This was caused by the variety in additional data such as shopper names and merchant data or a faulty log statement. The next example illustrates this case:

Log
```
Online refund
  [MERCHANTNAMEA-123123123123]
    error Online refund [MERCHANTNAMEA-123123123123] failed

Online refund
  [MERCHANTNAMEB-321321321321]
    error Online refund [MERCHANTNAMEB-321321321] failed
```

This is an edge case in which the clustering fails because the message is short and contains a large chunk of parameter data logged in the start and the end of the message. The log statement in this example is superfluous because the merchant name and reference number is logged twice. This issue can be fixed by changing the log statement or by lowering the threshold of our algorithm. Lowering the threshold is not preferred as it increases the probability of missing out on issues.

> Answer to RQ1: The aggregation algorithm presents a precision from [89.0%, 91.4%] and recall from [89.6%, 93.5%]. Therefore, we consider this method to be acceptable in clustering logs. The algorithm is both strong in detecting log clusters and precise in aggregating log data.

5.2 RQ2: Logness in the Real-World

During our research, Logness was able to capture 14 issues that were not captured by the current stack. Table 5.1 is a list of all the noticeable issues we found during our research at Adyen. The importance column indicates how severe the issue was in terms of business for Adyen. In this section, the most important alerts from Logness are listed as well as the cause for the problem and consequences for the Adyen system. Furthermore, the action taken to settle the problem is described and how Logness was instrumental in finding the issue.

#	Description	Date	Importance
1	Concurrency Issue #1	1/5/2016	low
2	Proxy Server Down	18/5/2016	medium
3	Log Issues	8/7/2016	low
4	SysMon Connectivity Problems	29/7/2016	low
5	Long Overflow Credit Card Numbers	5/9/2016	medium
6	Point-of-sale Issues	7/9/2016	high
7	Invalid Card Length Issues	10/9/2016	medium
8	Faulty Merchant Integration	10/9/2016	medium
9	Concurrency Issue #2	1/10/2016	low
10	Invalid Response From Acquirer	10/10/2016	low
11	VPN Tunnel Issues #1	3/11/2016	medium
12	VPN Tunnel Issues #2	3/11/2016	medium
13	Merchant Configuration Errors	15/11/2016	high
14	Firewall Issues	24/11/2016	high

Table 5.1: List of issues detected by Logness

5.2.1 Concurrency Issue #1

Since March 1, 2016 a bug was present in the Adyen system. This error was caused by checking the e-mail address of a shopper. If a large bulk of volume was processed, multiple instances tried to access the same character encoder instance, resulting in an *IllegalStateException*. This problem only occurred at the end of each month due to an increase in volume and happened at most 20 times in a week. This resulted in a retry of the payment which succeeded most of the times.

Time Period

March 1st, 2016 — March 9th, 2016

Result

Daily events:	3
Status:	Fixed on March 9, 2016 by a patch
Importancy:	low

Impact Adyen

The impact on Adyen was minimal because the issue occurred only at the end of the month, and did not significantly cripple the performance of the system. However, issues like these accumulate and the volume of payments handled by Adyen is increasing, meaning that it can occur more often. These risk checks are an important part of the payment, without them a payment cannot be completed and it is therefore important that the issue is fixed.

Logness as Instrument

This was the first issue found through Logness, and at that point in time we had only implemented a very basic version of Logness. However, it was still instrumental in finding the issue because of the sorting mechanism. All the non-custom exceptions are filtered out of the log messages and are then sorted, e.g. *NullPointerException*, *IOException*. The exceptions are sorted by frequency they appeared in the log files. *IllegalStateException* was in the top 10 issues we investigated. This issue has been solved as of March 9, 2016.

5.2.2 Proxy Server Down

On May 17, 2016 there was a short outage on the Adyen datacenter caused by a firewall patch. The logs were flooded with errors from almost all applications that could not reach the datacenter. Adyen is running payments to an external party through a proxy for security reasons. One of the two proxies did not recover successfully after the firewall patch and started producing *SocketTimeoutExceptions*.

Time Period

May 17$^{\text{th}}$, 2016 — June 18$^{\text{th}}$, 2016

Result

Daily events:	3798
Status:	Fixed on June 18, 2016 by rebooting the server
Importancy:	medium

Impact Adyen

This issue had gone unnoticed because the other proxy server was still running. The result was that in one month, 23,000 payments were retried. This is not as catastrophic as it sounds because in the end, the payments were still processed. However, the processing of these payments were done very inefficiently and could have led to disaster if the other proxy server had crashed. This issue was noticed by the acquiring team, but it was dismissed as a temporary trend in the payments processed by the specific acquirer.

Logness as Instrument

Logness picked up a constant stream of errors that was not there before May 17, 2016. The message is logged as a warning because a connection problem does not always have to be severe. However, the warning graph in Kibana was not showing any rising trends because the issue was occurring three times per minute, while the number of warnings are roughly 500 per minute which flooded the issue completely. The issue was also missed by the current monitoring practice because there was no custom alert set for this event. Logness was able to identify the issue because it aggregated the logs and sorted them on most frequent each day. This issue was listed as a top 3 issue during the day and was therefore directly visible

to the developers. No effort was required in detecting the issue using Logness while it was unnoticed for over a month by the current monitoring stack.

5.2.3 Log Issues

Starting Friday 8 July, 2016 a new feature was tested on live that logged over 30,000 warnings every 10 minutes over the whole weekend. It collected 5GB of log data in total. The warnings were caused by continuously retrying to poll a database without the proper configuration.

Time Period
July 8th, 2016 — July 11th, 2016

Result

Daily events:	14 million
Status:	Fixed on July 11 2016 by a patch
Importancy:	low

Impact Adyen

This issue did not influence the live systems although it ran on the live servers. No real harm was done for Adyen, but because of this sloppy error, the warning logs are polluted with log data that is not important. If another issue happened during the weekend on the same machine, it would be much harder to figure out the root cause.

Logness as Instrument

Logness detected the issue as soon as we deployed the application on Monday morning, because not surprisingly, the error was on top of all the other errors by a wide margin. Therefore, the issue was easily spotted using Logness. Although the issue generated a lot of warnings, it was not noticed by the current monitoring practice because there is no custom alarm set for this operation. Secondly, the issue is logged as a warning and therefore not visible in the Kibana dashboard because of the constant noise of log data. The issue was resolved as soon as we reported it to the responsible developer.

5.2.4 SysMon Connectivity Issues

SysMon, one of the system monitors of Adyen started to produce errors every minute because of connectivity issues on July 29, 2016. This issue was not detected because like most other issues, the issue was logged as a warning that was not visible in the current warning trend line. No other monitoring mechanisms were in place to capture this issue.

Time Period
July 29th, 2016 — July 30th, 2016

Result

Daily events:	400
Status:	Fixed on July 30, 2016 by a reboot
Importancy:	low

Impact Adyen

The impact on Adyen was minimal because no other issues rose during the connectivity issue. Furthermore, there are multiple instances of SysMon running so there was no downtime of the monitor. However, it had gone unnoticed by the current monitors and the developers, that one of the nodes was not functioning, which can lead to a potential failure if the other node crashes.

Logness as Instrument

Logness detected the issue because it was registered as a new, frequently occurring issue. The issue was in the top 5 issues during the day reported by Logness. Therefore, no effort was required in finding the issue using Logness.

5.2.5 Long Overflow Credit Card Numbers

Most credit card numbers consist of 16 to 19 digits and start with a number between 1 to 6. However, in some countries, new credit card numbers have been released using a number that starts with "95" and are 19 digits long. Somewhere in the transaction network, the number is converted to a long-type, because that is required by the ISO standard, causing an overflow that resulted into a failed payment.

Time Period Unknown — September 5$^{\text{th}}$, 2016
September 5$^{\text{th}}$ — October 25$^{\text{th}}$, 2016

Result

Monthly events:	4
Status:	Fixed on September 5, 2016 by a patch
	Server rebooted October 25, 2016
Importancy:	medium

Impact Adyen

This case was hard to detect, because it only affected a couple of people every month. However, credit card numbers keep growing and eventually, the problem will unfold itself in an unfavorable option, namely a merchant leaving Adyen because of bad user experience.

Logness as Instrument

Logness detected an unknown issue logged as a *NumberFormatException* which drew our attention. Logness was already locally operational for a couple of weeks, containing trained

data. New exceptions are therefore quite unknown and as soon as this issue popped up, it was investigated. The issue was fixed on September 5, 2016 but re-occured later that month. This time, the message was successfully generated but was passed down to a proxy server that was not patched. The proxy server could not handle the message and failed. The proxy servers are often not patched because they don not require any changes most of the time. Logness detected the "new" issue only after three weeks. This can be explained by the fact that Logness is not running in production at the time of writing. Therefore, some issues are missed because the application is simply offline. The issue was missed by the current monitoring practice because custom alerts are impossible to create for events that are unknown in advance.

5.2.6 Point-of-sale Issues

The patch of September 7, 2016, introduced better support for terminals that are running old libraries and cannot be updated. However, a bug was introduced which caused some payments to not be captured. The result is that the customer can still buy goods, but the merchant will not receive their funds. The warning was logged as an *IOException* because the input was corrupted.

Time Period

September 7th, 2016 — September 9th, 2016

Result

Daily events:	2,4 million
Status:	Fixed on September 7, 2016 by a patch
Importancy:	high

Impact Adyen

The issue was missed by the developers probably because of a couple of reasons:

1. The issue slowly ramped up instead of popping up in the logs instantly.
2. The issue was logged as a warning message and was not noticeable in the warning trend line.
3. The issue was very similar to a non fatal log message that required no action.

The issue caused a massive number of logs, but no alert was sent out by the current monitoring stack because it was logged as a warning message. The issue is only related to terminals from a specific merchant, because these terminals are running old library software that caused the issue. The issue was involved in a couple of hundred of payments but did not affect the shopper.

Logness as Intrument

Logness detected this issue because it picked up a stream of errors that produced significantly more logs than any other issue during the day. The issue was clustered as a new,

unknown issue and was not matched against the non-fatal log message that was already present. Logness identified the issue as the most important event of the day and could therefore easily be spotted by the developers. The issue was resolved after it was fixed by a hotfix.

A very similar issue re-occured a month later when another merchant decided to put old terminals back in order. This issue however was not initially detected as the log message was slightly different and did not contain an exception. This time it did not involve a capture, but only a cancel operation. That is, an action were the shopper cancels a payment on the terminal. Logness captured the issue because it was by accident logged as an exception when the encryption processed failed for very specific input. It was therefore hardly visible. To better detect these issues, we should not only consider warnings that contain exceptions, like we do now, but consider all warning messages.

5.2.7 Invalid Card Length Issues

Credit card numbers that have less than 10 digits cause an exception in some parts of the Adyen system. Instead of handling these creditcard numbers an *IndexOutOfBoundsException* is thrown.

Time Period September 10, 2016 — October, 2016

Result

Daily events:	200
Status:	Fixed October, 2016 by a patch
Importancy:	medium

Impact Adyen

This issue was not directly involved in the payment process, and was therefore not critical. However, this does impact the performance of the system because the error is not gracefully handled. The process is retried several times; putting a strain on the system performance. Initially we believed that this was a fraudulent transaction because there are no credit card numbers with less than 10 digits. However, it appeared that for very specific countries credit card numbers with less than 10 digits do exist. This demonstrates that this scenario was uncommon to some of the developers and Logness was able to point the developers to the right direction.

Logness as Instrument

Logness was instrumental in finding the issue because the issue was listed in the top 5 issues of the day and therefore prominently visible to the developers. The exception was not very critical and therefore not logged as an error, and did not trigger any custom monitor alerts. The issue was not visible in warning trendline because the number of warnings is far below the daily average of 1 million warnings (as of September 2016).

5.2.8 Faulty Merchant Integration

One of Adyen's merchants has been successfully processing payments since August 2016. However, some payments fail because this merchant is encrypting credit card data wrongly. This issue does not always unfold itself because it depends on the content of the encrypted data whether the hash is corrupted or not. This happened to almost 60 payments every day for this merchant.

Time Period August — Undefined

Result

Monthly events:	1800
Status:	Reported to the merchant as of October 4, 2016
Importancy:	medium

Impact Adyen

There is no bug present in Adyen's system. The issue is captured and successfully transmitted to the merchant. However, the merchant is probably not aware of the problem although it does affect quite some payments. By reporting this issue to the merchant both Adyen and the merchant benefit from a fix.

Logness as Instrument

Logness was instrumental in finding the issue because it prominently displayed the issue to the developers. It was listed in the top 10 issues of the day based on frequency. Although it is not something that requires a fix on Adyen' side, we still marked it as an issue because it is important to help the merchants as much as possible in fixing small but critical problems like this one. It is also in the best interest of Adyen to process more payments. This demonstrates that we can potentially use Logness to discover bugs on the merchant's side as well. The current monitors did not capture the issue because it was logged as a warning and hardly visible because of the low frequency.

5.2.9 Concurrency Issue #2

Logness discovered *ConcurrentModificationExceptions* which is an indicator of concurrency issues. The issue has been present since the launch of Adyen.

Time Period 2007 — October, 2016

Result

Monthly events:	4
Status:	Fixed October, 2016
Importancy:	low

Impact Adyen

The impact was minimal because it only affected the BackOffice application Adyen employees use internally. It was also hardly noticed because this concurrency issue occurred only 4 times every month. The effect is that the action fails, after which a retry probably succeedd. This issue has been around since the starting days of Adyen but was never noticed.

Logness as Instrument

Logness was instrumental in finding the issue because like the other issues, it was detected as a "unique" error log message. It was therefore prominently visible during the day in Logness. The issue was also noticed because the exception stood out between the standard timeout exceptions.

5.2.10 Invalid Response From Acquirer

Payments that fail because Adyen cannot parse response data from external parties are often noticed quickly; because in the end, no funds are transferred between the different parties. However, in the case that a payment should fail, e.g., when an invalid card number is provided, an invalid path of execution is not always noticed. In this case, when an invalid card number from a prepaid credit card is provided to a specific issuer to check the balance of the card, the response cannot be parsed and the result is a *NullPointerException* resulting in a *Server Error* to the shopper. The transaction fails but the execution path is invalid and should return a meaningful error.

Time Period
July 1st, 2016 — October, 2016

Result

Monthly events:	208
Status:	Fixed, October 2016
Importancy:	low

Impact Adyen

The impact for Adyen was minimal because this only involved balance checks for prepaid credit card numbers, for a specific issuer. The balance check fails because an invalid card number is provided. However, statistics on these transactions are skewed because instead of storing the payment details as a failed payment with the cause: "invalid card number", Adyen stores the payment status as a failure due to an error on the side of the issuing bank.

Logness as Instrument

Logness was instrumental in finding this issue mainly because we filtered on known important exceptions such as *NullPointerException*, *IllegalStateException* and *ConcurrentModifi-*

cationException. Even though the message was logged with an "ERROR" severity level, it was not picked out by the developers. We believe that this is because of the poor visibility of small (less than 300) issues in the bulk of warning log messages. Logness was able to filter out important exceptions such as *NullPointerExceptions* and present them to the developers.

5.2.11 VPN Tunnel Issue #1

On November 3, 2016 a small internet outage in the United States caused the Miami datacenter to be unreachable for 3 minutes. The result was that a VPN tunnel to an external party disconnected and did not automatically revive.

Time Period

November 3rd, 2016 — November 4th, 2016

Result

Daily events:	7,000
Status:	Resolved, 4 November 2016
Importancy:	medium

Impact Adyen

There was no direct impact because of two reasons. First of all there was a quick response after we pointed out the issues using Logness. Secondly, there are always two connections available for merchants to connect to the datacenter. Most of the time, the load is balanced between the two datacenters. However, this time only one connection was used. The other one was only used when there is maintenance on the "main" connection. There is only a problem when these events coincide; which we prevented to happen.

Logness as Instrument

Logness was instrumental in finding the issue, because it stood out between the other issues in the tool and was logging a constant stream of *SocketTimeoutExceptions*. The issue was displayed as number one issue of the day by Logness. We have not seen this issue before so it was presented as an "unknown" issue. After the issue had been reported, it was fixed within a couple of hours.

5.2.12 VPN Tunnel Issue #2

Related to the problem in 5.2.11, a similar issue arose during the weekend of 5 November, 2016. Maintenance was done on one of the servers of the external partners. When the maintenance was done, the VPN connection did not automatically revive.

Time Period

November 5th, 2016 — November 5th, 2016

Result

Daily events:	17,300
Status:	Resolved, 4 November 2016
Importancy:	medium

Impact Adyen

As with the other related issue, there was no direct impact as the backup server is still able to process payment requests for the external party. However, if this servers fails, payments cannot be processed through this external party.

Logness as Instrument

The issue was not noticed because it generated quite some warning messages, but only a few error messages. This issue arose during the weekend so there were not many people watching the system performance. The number of errors was below the threshold, so no emergency e-mail was sent out. Logness picked up this error stream because first of all, it was unique, and secondly it was generating much more events than every other issue. It was therefore prominently displayed in the top issues of the day.

5.2.13 Merchant Configuration Error

During the morning of 15 November, 2016 an issue came to our attention that involved a specific merchant. It was quickly noticed that the change was requested by the merchant itself. The support team applied the change, but shortly after this the payments stalled for this merchant.

Time Period

November 15th, 2016 — November 15th, 2016

Result

Daily events:	24
Status:	Resolved, 15 November 2016
Importancy:	high

Impact Adyen

The impact on the merchant is immense because the payments stalled. Although the change was requested by the merchant itself, they did not realize that this small change would stall all payments. The change should not have been done in the first place, but due to a miscommunication, the change was accidentally processed.

Logness as Instrument

This is a strong case for Logness because we detected and fixed the issue before the merchant started complaining. Because the issue was high priority we did not wait to see if the

Adyen developers could notice the issue. We doubt if it would have been detected by the regular monitors because it was logged as a warning message and only occurred once every minute. The log level of this statement should clearly be changed to an “ERROR”-level. But even then, the issue is probably not visible in the current monitoring practice because it is flooded by the other messages. Logness detected the unique, new issue since the tool had been running on trained data for the past month. Unknown events are rare and are therefore easily noticed because they are prominently displayed as top priority issues.

5.2.14 Firewall Issues

Shortly after patching one of the servers, a continuous low frequency warning stream was picked up by Logness. The firewall was not configured correctly which caused some of the applications on the server to be unreachable.

Time Period

November 24th, 2016 — November 24th, 2016

Result

Daily events:	3,600
Status:	Fixed on November 24, 2016 by firewall fix
Importancy:	high

Impact Adyen

This issue caused no direct impact for Adyen because there were no merchants yet configured for this server. However, future merchants could have been configured for this server and would cause major issues as soon as the merchant would go live. It still would have been possible for merchants to process payments, but displaying payments would result in errors on this machine.

Logness as Instrument

Logness was instrumental in finding the issue because it picked up a small error stream in the log data and prominently visualized it as top issue of the day. Logness is able to detect small changes in log messages because the tool is trained with annotated log data. Unique issues are therefore rare and this issue was suspicious, because it appeared right after patching. The issue appeared even though no merchant accounts were active on the server. The error unfolded itself because it was caused by a small component in the application depending on the same firewall rules as the live merchant accounts.

This issue illustrates that Logness can be powerful in detecting issues by prominently visualizing issues for developers. The current visualization of the warning and errors is lacking this capability to capture delicate issues.

5.2.15 Small Remarks

October 26, 2016 a lot of issues stacked during the day because of a large system update. Each fix introduced a new problem. At the end of the day, the developers were tired and missed some of the problems. Logness provided insight in the current state of the logs and captured these mistakes. The issues did not form a direct threat, but could potentially wake up the duty team during the night, if the problem persisted. These problems were missed because the focus was gone at the end of the day. Logness aids by pointing out the important log changes such that constant focus is not required.

5.2.16 Answer to RQ2: Logness in the Real-World

Logness is able to support developers in finding relevant issues. It is able to capture the "state" of the current logs produced by the system and provides an innovative way in visualizing log clusters. The effort required to find issues using Logness is minimal because Logness is able to automatically sort important issues. Logness was able to identify 12 out of 14 issues as something "important" and displayed them as top priority during the day, without search effort from the developers. In the other two cases, we were at a point in time were we had only implemented a very basic version of Logness which did not intelligently sort log clusters. However, Logness was still helpful in finding the issue because it filtered important exceptions.

Answer to RQ2: Logness is able to support developers in finding relevant issues. It is able to capture the "state" of the current logs entering the system and is precise in detecting possible issues without flooding the developers with too many issues.

5.3 RQ3: Logness as Monitor Instrument

To answer RQ3: "How does Logness behave around a release cycle?", we actively monitored a large release cycle on November 8, 2016. During the day, all servers were patched because a new currency was introduced. We successfully captured all issues during the day and we caught some extra issues that were not detected by the regular monitors. The following table contains the details of the issues we encountered during the day.

5.3.1 Release November 8, 2016

The patch described in table 5.2 was a small system patch. We do not expect many issues because the change is small but it is required to patch all servers.

- **9:20 AM** The patching process started around 9:30 AM but we already recorded an issue before the patching process started at 09:20 AM. This issue was not detected by the development team until 09:30 PM and involved a *NullPointerException* crashing the application. Unfortunately, we misjudged the issue and we postponed investigating the issue by asking help from a developer because the patching process was

Time	Event	Logness	Current Practice
9:20 AM	NPE on daily running job	x	-
10:00 AM	Base64 decode failed for mobile payment method	x	x
10:16 AM	Electronic Payment Authorization (EPA) failed	x	x
10:30 AM	Issues with an added test merchant account	x	-
10:52 AM	Missing configuration for specific bank/acquirer	x	x
11:00 AM	BackOffice down	-	x
11:07 AM	Errors for specific merchant skins	x	x
01:48 PM	Increase on timeouts to specific server	x	x
02:00 PM	NPE on application not patch related	x	-
02:30 PM	Patching Finished		
03:50 PM	Skin issue re-occurred	x	x

Table 5.2: Events during the day while doing a small patch on all the servers

getting started. This issue was discovered and solved therefore at 9:30PM instead of being solved instantly.

- **10:00 AM** We detected an issue that was related to mobile payment methods because for this specific method, no local testing method is available. This issue was captured by both monitoring systems.
- **10:16 AM** A specific job failed that was picked up by both monitoring systems because it is monitored by the custom alert mechanism.
- **10:30 AM** An issue came to our attention that was not noticed by the team but was picked up by Logness. After heavy investigation, we found out that this was related to specific servers forgotten to be patched and are therefore running on old software. This issue was not noticed by the development team because it appeared on servers that were not watched by the developers.
- **10:52 AM** A misconfigured server caused some small issues that were quickly resolved and caught by both the team and Logness.
- **11:00 AM** The BackOffice (admin tool for Adyen) experienced downtime because the server was suffering from memory exhaustion. This issue was missed, and could not be caught by Logness because Logness is only monitoring application logs, and therefore misses out on hardware performance issues. It is the responsibility of the hardware monitoring tools to detect these issues and alert on them.
- **11:07 AM** The first skin issues started. Skins are used to generate custom payment pages for merchants. This issue was noticed by both Logness and the monitoring team and involved corrupt settings for a specific merchant. This issue re-occurred later that day and was detected by both the team and Logness.
- **1:50 PM** Timeouts to a specific datacenter increased which was detected by Logness and the DevOps team. Logness was able to detect the issue because *TimeOutExceptions* increased significantly during this time frame.
- **2:00 PM** During the day we decided to point the developers to existing *NullPointerExceptions* that are not critical and patch related. Around 2:00 PM one of these

issue was pointed out to the developers that was fixed within 30 minutes. This demonstrates that problems are probably fixed quickly if developers are aware of these small issues. This issue was present for more than three months.

- **3:50 PM** The patching process finished around 2:30 PM, but the monitoring process was not. A re-occurring issue was detected around 3:50 PM which was fixed quickly.
- **6:00 PM** Monitoring finished.

During the day we caught all important issues and even detected more problems than the current monitoring practice. Logness was able to detect all the issues but do require some attention by skilled developers that are able to make good judgment calls from the events. There is almost no effort in detecting issues using Logness as the developers are automatically notified for new, unique events. When a re-occurring issue appeared, it is prioritized and directly displayed to the developer. Logness created a total of 46 new log clusters during the day which require little attention to annotate. Annotation costs are further discussed in Section 5.4.

The developers used Logness as a safety net during the release. Actively monitoring the system logs after a patch is still preferred, but impossible to to for all servers. Logness aids the developers by providing insights in all the important log events for all the servers.

5.3.2 Release November 29, 2016

We actively monitored the release of November 29, 2016. December is a critical month for Adyen and therefore, no new releases are deployed after this release for a whole month. The following table contains the details of the issues we encountered during the day. The release started around 8:00 AM.

Time	Event	Logness	Current Practice
09:28 AM	Large DB replication issues caused time-outs	x	x
09:37 AM	First server patched	-	-
10:37 AM	Incomplete database changes caused exceptions	x	x
11:10 AM	NPE after DB changes caused by a software bug	x	x
11:30 AM	Sanity check failures for multiple jobs.	x	x
12:40 PM	NPE in software that is not patch related	x	x
01:45 PM	NPE caused by a software bug	-	x
02:00 PM	Server crashed	x	x
03:00 PM	Re-occurring issue that is not harmful	x	x
03:30 PM	Re-occurring issue caused by database replication	x	-
04:00 PM	Long running query causing time-outs	x	x
04:30 PM	Patching process finished	-	-

Table 5.3: Events during the day while doing a small patch on all the servers

- **9:37 AM** Database replications are done before the servers are patched. Operations on large tables take a while to finish and that is why at 9:37 AM the first server was

finally patched. Unfortunately the database replication was not fully completed and that caused a burst of database exceptions around 10:37 AM.

- **11:00 AM** A software bug caused *NullPointerExceptions* because the newly introduced database model was not taken into account. Logness could only detect the issue after 10 minutes because Elasticsearch was running behind. This is an infrastructure problem that can be solved by upgrading the hardware.
- **11:30 AM** Sanity checks are done on every patched server to verify the compatibility between the code and the database. Some of them failed at 11:30 AM and Logness was able to detect this faster than the current monitoring practice because the issues originated from servers that were not patched at the moment, and thus not actively watched by the development team.
- **12:40 PM** *NullPointerExceptions* detected that did exist before the patching process. It was already caught by Logness but not fixed yet because it is a low priority issue for which no resources were available at the time.
- **1:45 PM** The first issue that was not caught by Logness was caused by a small bug in the grouping algorithm. Logness accidentally hid a *NullPointerException* that was caused by a software bug, and unfold itself around 1:45 PM. After fixing this bug Logness should be able to detect similar issues in the future.
- **2:00 PM** Around 2:00 PM one of the servers crashed. Both monitors picked up the issue but for Logness, it was hard to identify the cause because the server was not logging anything after the crash. However, responsibility for hardware crashes are often monitored by other applications that are out of the scope of this research.
- **3:00 PM** A re-occurring issue was suddenly appearing more often than normal. This issue was already annotated as something "non-severe" in Logness and it was caused by the warm-up procedure of the patched server. It was not harmful but it was investigated by the development team because it was unknown to the monitoring developers.
- **3:30 PM** A re-occurring issue started logging a constant stream of errors. This was not picked up by the development team because it was logging just a few errors every second on a server that was already patched in the morning. Logness was able to pick up this stream and we reported it to the monitoring team which quickly fixed the issue.
- **4:00 PM** Long running queries caused a short peak in time-outs to the database. This issue was picked up by both monitors because the issue caused many problems.

Around 4:30 PM the patching process was finished. Logness detected one more issue than the development team, but failed to report on one issue as well. The Adyen developers are enthusiastic about the accuracy of the tool and see Logness as a safety net in the patching process.

Logness generated a total of 71 log clusters during the day which is exceptionally high. However, in Section 5.1.3 we have shown that despite the large number of clusters, Logness does correctly cluster them together. It is also not hard to annotate these log clusters as most of them are caused by connectivity issues when a server was shut down. As with the last release, little effort is required in "finding" the issue because Logness automatically notifies the developers for re-occuring and new events.

To answer RQ3: We conclude that during the two-weekly release cycle, Logness was able to detect at least the same amount of issues, and sometimes detected more issues. Logness is especially helpful in the patching process by giving insight in the state of the log messages for all servers, instead of just the server that is being patched. Issues are not hard to find because Logness is able to prioritize issues automatically using a trained set of log data. A constant focus is not required because Logness is efficient in filtering and clustering logs and produce almost no false positives.

5.4 RQ4: Training Effort

To answer RQ4: "What is the effort of training Logness?", we collect the data generated by Logness from the period 15 October, 2016 to 24 November, 2016. This data set contains a timestamp of each created log cluster and the annotation made by the developers. We exclude the days for which Logness was online for less than 7 hours.

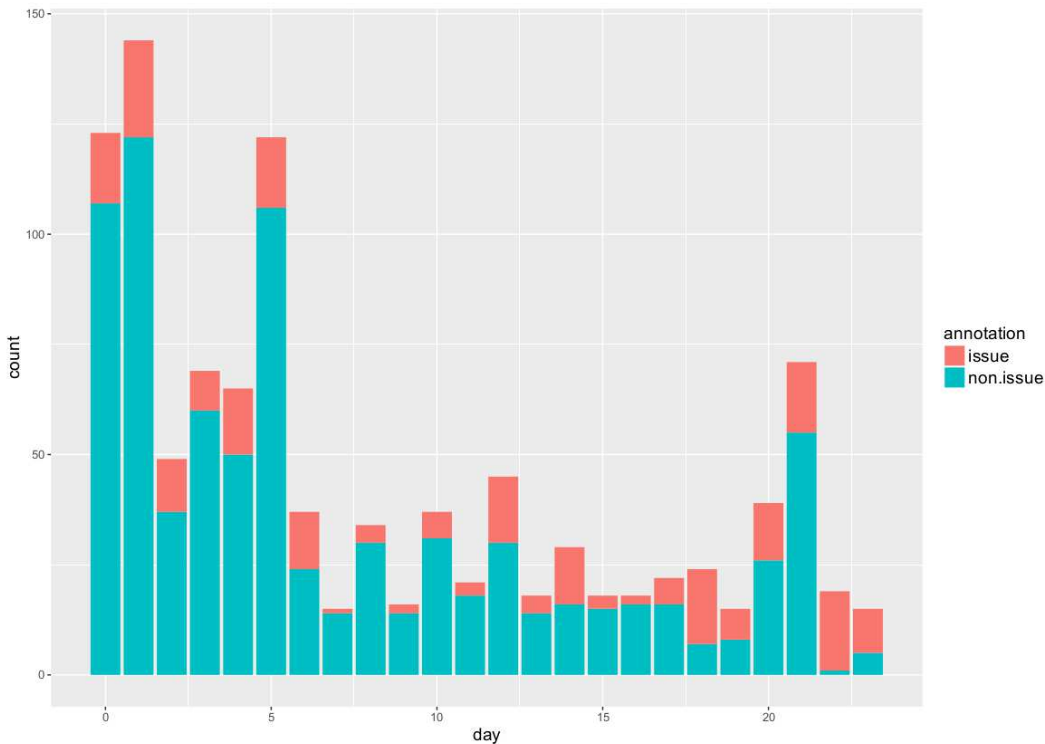

Figure 5.1: Number of issues and annotation type every day recorded by Logness.

In Figure 5.1, bars illustrate the number of new events, and colors represent the classification that we manually performed. It is no surprise that the first few days, many new log clusters are created because no training data is available yet. However, we can see that the number of new events quickly decreases with the exception of the 20th and 21th day. After 5 days we can see a huge drop in the number of log clusters. The median number of issues

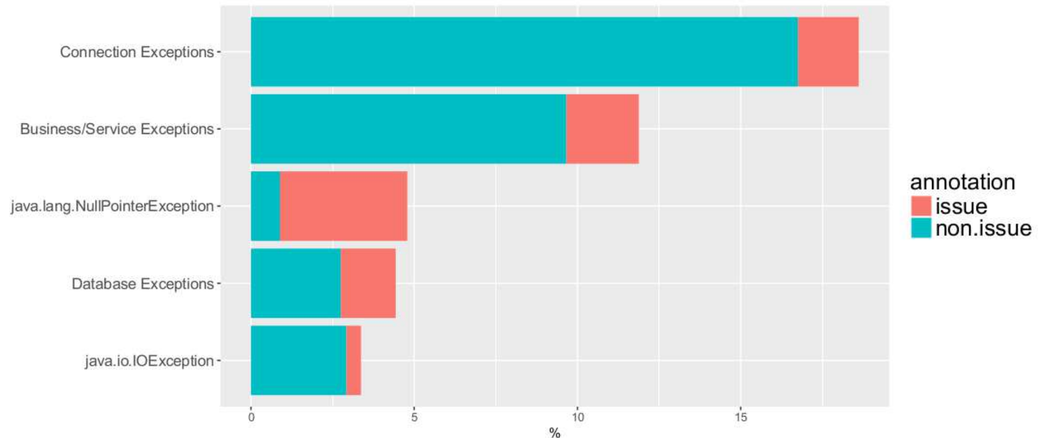

Figure 5.2: Type of exceptions and severity collected by Logness as a percentage of the total.

after the first 6 days are 22 issues every day of which 6 on average are tagged as severe, bug or monitored. This is a reasonable number of issues to manually verify, because most issues are temporary connection problems as we can see in Figure 5.2.

The sudden rise of events on the 20th and the 21th is the effect of patching servers that are heavily used by multiple acquirers. Each time a server was patched, the active connections closed and the fallback mechanism kicked in which reroutes payments to another server. This caused multiple unique *SocketTimeoutExceptions* and *ConnectTimeoutException* for each acquirer on each server. We have shown in Section 5.1.3 that despite the large number of clusters at this day, Logness was still able to correctly cluster most of the log messages.

In practice, we noticed that some exceptions appear frequently and, as soon as a developer understand the reason behind one of them, investigating the remainder ones become easier. Thus, in Figure 5.2, we show the most popular exceptions. Connection exceptions are by far the most common ones. As mentioned before, these exceptions happen frequently when a server is patched and require little time to annotate. In second place, we observe many business/service exceptions, such as *"Not enough balance"* or *"Internal server error"* and can be quickly inspected. On the other hand, most of the *NullPointerExceptions* are tagged as an issue and usually require a deeper investigation.

Answer to RQ4: We conclude that the amount of effort required to use and annotate Logness is low because after the initial training phase, on average 25 issues are generated every day. Most of these issues are connection exceptions that require little effort to annotate.

Chapter 6

Conclusion

Large internet companies can generate over 600GB of log data every day. This log data is a rich source for analyzing the performance of different operations and troubleshooting problems. However, with millions of log messages generated every day, log management tools become crucial. At Adyen, the ELK stack is used as log management tool. Still, too many errors are unnoticed because there is no detection mechanism for outliers in log data.

We built **Logness** at Adyen, a log analysis tool that is able to uncover issues that are hidden in the log data. The tool clusters log messages together based on a couple of heuristics such as filtering digits, hashes, and comparing raw log data using the LCS algorithm. They are then presented to the developers and prioritized based on the frequency in which they appear, and the classification that is given by the developers. By examining a small set of previously unseen log clusters, the accuracy is improved such that only relevant issues are shown to the developers. The tool runs on top of the ELK stack without any change to the infrastructure, source code or deployment process.

The current monitoring practice includes customized alerts for event monitoring. We demonstrated in our research that we found many cases where these alerts fail to capture problems. By tackling this problem in a black box approach, that is, we are not interested in the working of the system, but only consider the output of the application; we have provided a framework that is able to detect subtle errors. The feedback from the developers helps us to filter out false positives.

We evaluate Logness by answering the following research questions:

- **RQ1: How accurate is Logness' classification algorithm?** By manually verifying the created log clusters, we have shown Logness is able to cluster logs with a precision and recall of at least 90%. Most failures in log clustering where caused by cut-off stack traces that can be easily fixed to improve the clustering algorithm.
- **RQ2: Does Logness support developers in finding relevant issues?** We have shown by capturing 14 issues that were gone unnoticed using the current monitoring practice, that Logness is a valuable addition to the current monitoring stack.
- **RQ3: How does Logness behave around a release cycle?** During the two-weekly release cycle, Logness was able to detect issues during the patch missed by the current monitoring practice. Logness did not under perform by detecting at least the same

number of issues in the release as the current practice. The developers found Logness helpful as a safety net in the patching process.

- **RQ4: How many issues appear each day after the initial training phase on a regular basis?** After the initial training phase, on average 25 new issues appear every day. Most of these issues are short-term connectivity issues that can be resolved quickly.

6.1 Implications

In Chapter 3 we have stated our objectives and why we built our own implementation. To summarize:

Better insights. Logness provides more insight in log data by clustering messages and provide valuable information about log statements such as: "*Is this message important?*" and: "*When did it first occur?*". Without Logness these questions are hard to answer, because log aggregation is not easily done using the current practice.

Better awareness. Currently, errors are missed when no patch is on-going because there is a lack of active monitoring. When an overview is presented of possible issues, developers are more willingly to fix issues because they do not have to actively search for problems. This approach is different than the current approach where a graph visualizes the error/warning trendline, but no concrete exception is shown. Developers are more aware of issues and tend to investigate it, if they are presented with a real exception. For example, by showing the cause of the problem such as a many *NullPointerExceptions* or *ConcurrentModificationExceptions*, a developer is triggered to fix an issue more willingly than when he or she is just confronted with a number about the total number of errors in the application.

Security and performance consideration. Logness is atomic and runs on top of any existing solution without changes to any parts of the application. It is therefore save in a security and performance perspective.

6.2 Limitations

Our research shows that Logness is useful in aggregating and visualizing log data. However, there are some limitations to the tool.

Infrastructure The current infrastructure of Logness is polling a server for the latest stream of log messages. This is inefficient but required due to the current logging infrastructure at Adyen. The result is that the amount of logs that can be processed is limited to around 100 errors/warnings every second. This can be improved by setting up a streaming cluster for logs. One of the consumers for this stream should be Logness. Multiple consumers can be setup to process logs in parallel. The clustering algorithm can also be

tweaked to run in parallel. These techniques have already been proposed in literature, for example in a distributed framework for processing logs called Kafka [9]. By improving the infrastructure we can also consider all the warning messages instead of a filtered subset to improve issue detection.

Clustering Improvements An important part of Logness is how it aggregates similar events together and creates clusters. This can be complex and confusing for users when trying to understand information that is not grouped correctly. Future work is required to improve the clustering algorithm such that we can match log messages with a higher precision and extract parameter values.

6.3 Beyond Adyen

Logness has been proven useful for Adyen by making the developers more aware of actual logs Adyen is producing. We visualized the logs and prioritized them in a general fashion such that it can be used by other companies as well. Logness has been designed to run on top of Elasticsearch, but can easily be extended to run on other log frameworks as long as we can continuously stream logs to Logness. We fine-tuned the algorithm to work best for our case, that is detect issues that are hidden in the current monitoring process. The clustering algorithm may be too strict or too loose depending on the company' application log. Therefore, the clustering algorithm can be easily customized.

Logness is creating multiple fingerprints based on log messages. To cut down the amount of issues, we try to combine them by identifying correlations between different log messages based on a payment reference. To leverage this method in other companies, we can find other correlations between log messages. Many companies keep some kind of reference in the log messages to identify that certain actions are involved the same operation, for example, an user that orders a certain product, go through a flow of operations which can log the user-id as a reference. This is leveraged in the work of Lin et al. [10] for Microsoft, in which they used the *taskid* of an operation to correlate messages.

Adyen is enthusiastic about Logness and provide resources to continue after this research. We plan to open source the application as soon as the limitations are solved. We believe that Logness can provide a valuable addition in the jungle of log frameworks because it can extract issues from big log-data, without touching any code or configuration in the existing stack. It is therefore unique in the field of big log-data because Logness is completely atomic.

6.4 Future Work

Multiple steps can be taken to improve the insight in log data. Logness is a first step in aggregating log data and can be improved in the following directions:

Intelligent issue sorting The issues are categorized, based on the annotations made by the developers. The required work to create these annotations is minimal. However, developers

should keep record of the annotated issues. Possible directions are automating the process of creating annotations by applying machine learning on the trained data set.

Parameter value insight As described in Section 6.2, improving the clustering algorithm such that we can extract the exact parameter values can be interesting. These values can be used aggregate log data in different levels. For example, the first level is the sorted table with the exceptions; the second level provides insight in the frequency of the parameter value. For our industry partner Adyen, this means that the message: *"Can not contact acquirer X"* can be easily monitored for every individual acquirer instead of a general "acquirer problem".

IDE integration Integration with the IDE can be helpful for the developer. Improvement of the clustering algorithm is required such that we have a one to one mapping between the log statements and the log messages. This plugin provides detailed information about the log statements live to the developer that is currently editing the class. Challenges lies in the performance and clustering domain to realize this improvement.

Bibliography

[1] Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 39–48. IEEE, 2000.

[2] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[3] Alex Fattal. Facebook: Corporate hackers, a billion users, and the geo-politics of th social grap. *Anthropological Quarterly*, 85(3):927–955, 2012.

[4] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 20–20. USENIX Association, 2007.

[5] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *ICDM*, volume 9, pages 149–158, 2009.

[6] Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

[7] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, May 2000.

[8] Zhen Ming Jiang, Ahmed E Hassan, Parminder Flora, and Gilbert Hamann. Abstracting execution logs to execution events for enterprise applications (short paper). In *2008 The Eighth International Conference on Quality Software*, pages 181–186. IEEE, 2008.

[9] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[10] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. Log clustering based problem identification for online service systems. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 102–111. ACM, 2016.

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

[12] Adetokunbo Makanju, Stephen Brooks, A Nur Zincir-Heywood, and Evangelos E Milios. Logview: Visualizing event log clusters. In *Privacy, Security and Trust, 2008. PST'08. Sixth Annual Conference on*, pages 99–108. IEEE, 2008.

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

[14] Meiyappan Nagappan and Mladen A Vouk. Abstracting log lines to log event types for mining software system logs. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 114–117. IEEE, 2010.

[15] Sivan Sabato, Elad Yom-Tov, Aviad Tsherniak, and Saharon Rosset. Analyzing system logs: A new view of what's important. In *Proceedings of the 2nd USENIX workshop on Tackling computer systems problems with machine learning techniques*, pages 1–7. USENIX Association, 2007.

[16] Risto Vaarandi. A breadth-first algorithm for mining frequent patterns from event logs. In *Intelligence in Communication Systems*, pages 293–308. Springer, 2004.

[17] Ian H Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[18] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132. ACM, 2009.

[19] Wei Xu, Ling Huang, Armando Fox, David A Patterson, and Michael I Jordan. Mining console logs for large-scale system problem detection. *SysML*, 8:4–4, 2008.

[20] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: enhancing failure diagnosis with proactive logging. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 293–306, 2012.

[21] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)*, 30(1):4, 2012.